KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Implicit Code Generation for Polymorphism

Gert-Jan Bottu

Supervisor:
Prof. dr. ir. T. Schrijvers

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

January 2022

Implicit Code Generation for Polymorphism

Gert-Jan BOTTU

Examination committee:
Prof. dr. ir. H. Van Brussel, chair
Prof. dr. ir. T. Schrijvers, supervisor
Prof. dr. ir. G. Janssens
Prof. dr. ir. F. Piessens
Dr. R. Eisenberg
(Tweag Software Innovation Lab, France)
Dr. W. Swierstra
(Universiteit Utrecht, Netherlands)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

January 2022

Uitgegeven in eigen beheer, Gert-Jan Bottu, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Preface

First and foremost, I would like to thank my supervisor, Tom Schrijvers, who offered me the opportunity to start this work, the advice and teaching to guide me through this work, and the support and drive to do better to finish this work. Without him, none of this would have been possible, for which I'm incredibly grateful.

Secondly, I want to thank George Karachalias, who introduced me to the wonderful world of type theory. His unrelenting patience and mentorship taught me so much about science, about work and about life in general. Against better judgement, he even got me started in JavaScript development. He not only made this thesis happen, but made me the person I am today.

Furthermore, I would like to thank the jury for reading this work, for providing their insightful feedback, and for many educating discussions, both during the defense of this thesis and before, and hopefully afterwards as well.

My thanks also goes out to my collegues: Alexander, Amr, Birthe, Cesar, George, Klara, Koen, Ningning, Roger, Ruben and Steven. Together, we made the office both a fun and enlightening place to be, not to mention being in a constant state of amazement. Furthermore, I would like to thank my more junior collegues for our many interesting discussions and offering me the ability to pass on what I learned: Elias, Francisco, Jelle, Jo-Thijs, Lena, Reinert, Rene and Thomas.

My gratitude goes out to the incredible teams at Digital Asset and Tweag.io, who thaught me the real world applications behind the theory. Their knowledge and enthousiasm got me excited for working in industry. A special mention for Richard Eisenberg, for taking the time to introduce me to GHC development.

Of course, I'd like to thank my friends, family and parents for the incredibly important role they play in my life, and for which I'm eternally grateful. None of this would be possible without them. And last but not least, I'm grateful to Ruben for his love, patience and support. You're the best!

Gert-Jan
January 2022

Abstract

As software keeps growing in size and complexity, more sophisticated programming languages are needed to keep software development manageable. One important aspect in a capable programming language is its type system. A sufficiently powerful static type system can not only reduce bugs, but also implicitly generate common boilerplate code. This thesis improves upon on two forms of this implicit type-directed code generation, both in the context of polymorphism: parametric polymorphism and type classes.

The goal of this work is to improve the current state of the art in polymorphism, and we tackle this in three complementary ways:

Firstly, we investigate parametric polymorphism in Haskell by evaluating two design decisions related to type instantiation. For this, we introduce the concept of *stability* as a way of making these decisions. Stability is a measure of whether the meaning of a program alters under small, seemingly innocuous changes in the code (e.g., inlining). We define a type system family, which can be materialised to four different approaches of type instantiation. After defining 11 different stability related properties, and formally verifying them against every variant of the type system family, we conclude that the most stable approach is lazy (instantiate a polytype only when absolutely necessary) and shallow (instantiate only top-level type variables, not variables that appear after explicit arguments).

Secondly, we increase confidence in the type class resolution mechanism, the implicit type-directed code generation used by Haskell for function overloading. While type class resolution is generally nondeterministic (both in Haskell and other languages like Mercury and PureScript), we prove that it still behaves predictably. Indeed, multiple ways can exist to satisfy a wanted constraint in terms of global instances and locally given constraints. However, the property of *coherence* guarantees that every possible outcome of this nondeterministic resolution process behaves indistinguishably from the others in practice. Even though coherence is generally assumed to hold for type class resolution, as far as we know, this work is the first to provide a formal proof of this property in the presence of sources of nondeterminism, like superclasses and flexible contexts. The proof is non-trivial because the semantics elaborates resolution into a target language where different elaborations can be distinguished by contexts that do not have a source language counterpart. Inspired by the notion of full abstraction we present a two-step strategy that first elaborates nondeterministically into an intermediate language that preserves contextual equivalence, and then deterministically elaborates from there into the target language. We use an approach based on logical relations to establish contextual equivalence and thus coherence for the first step of elaboration, while the second step's determinism straightforwardly preserves this coherence property.

Thirdly, we increase expressivity of type classes from Horn clauses to the universal fragment of Hereditary Harrop logic. In fact, *quantified class constraints* have been proposed many years ago for exactly this purpose. Yet, despite being widely asked for over the years, besides a number of stopgap workarounds, quantified constraints had never been formally studied or implemented. We elaborate the idea into a practical language design, and provide a declarative specification of the type system. Furthermore, we design a type inference algorithm that elaborates into System F. While not a direct mapping, this work has provided the necessary kick-start for a GHC implementation of quantified constraints. Finally, we extend the aforementioned coherence proof with quantified class constraints, a non-trivial extension, to show the adaptability of the proof.

In conclusion, we have improved the state of the art of polymorphism, and its type-directed code generation aspects in particular. We have increased (1) the stability of parametric polymorphism, (2) faith in the correctness of type class resolution by formally proving it coherent, and (3) the expressivity of type classes by adopting quantified class constraints. This clears the way for more wide-spread adoption of these features.

Beknopte samenvatting

Met almaar groter en complexer wordende softwareprojecten neemt het belang van geavanceerde programmeertalen toe om de ontwikkeling van software beheersbaar te houden. Een belangrijk aspect van een degelijke programmeertaal is het typesysteem. Een voldoende krachtig statisch typesysteem kan niet enkel fouten vermijden, maar kan ook impliciet veelvoorkomende code genereren. Deze thesis verbetert twee vormen van impliciete type-gestuurde codegeneratie, beide in het kader van polymorfisme: parametrisch polymorfisme en typeklassen.

Het doel van dit werk is om de huidige staat van polymorfisme te verbeteren. Dit bereiken we op drie complementaire manieren:

Ten eerste onderzoeken we parametrisch polymorfisme in Haskell door twee ontwerpkeuzes gerelateerd aan type-instantiatie te evalueren. We introduceren hiervoor het concept *stabiliteit* als een manier om deze beslissingen te maken. Stabiliteit is een maat van hoe de betekenis van een programma verandert onder kleine, schijnbaar onschuldige aanpassingen in de code (e.g., het inlijnen van een variabele). We construeren een familie van typesystemen, waarvan de varianten vier verschillende vormen van type-instantiatie gebruiken. We definiëren 11 verschillende stabiliteitgerelateerde eigenschappen, en verifiëren ze tegen elke variant van onze familie van typesystemen. Onze conclusie is dat lui (een type enkel instantiëren wanneer dit absoluut noodzakelijk is) en oppervlakkig (enkel de bovenste laag variabelen instantiëren, dus geen variabelen die voorkomen na expliciete argumenten) instantiëren het meest stabiel is.

Ten tweede verhogen we het vertrouwen in het resolutiemechanisme van typeklassen, de impliciete type-gestuurde codegeneratie die door Haskell gebruikt wordt voor het overladen van functies. Ondanks dat typeklasseresolutie in het algemeen niet-deterministisch verloopt (zowel in Haskell als in andere talen zoals Mercury en PureScript), bewijzen we dat het zich wel steeds voorspelbaar gedraagt. Inderdaad, er kunnen meerdere manieren bestaan om een gevraagde constraint op te lossen in functie van globale instanties en lokale

gegeven constraints. Desondanks garandeert de *coherentie-eigenschap* dat elke mogelijke uitkomst van dit niet-deterministische proces zich in de praktijk niet te onderscheiden gedraagt van de anderen. Hoewel er algemeen aangenomen wordt dat coherentie geldt voor typeklasseresolutie, is dit werk—voor zover we weten—het eerste dat deze eigenschap formeel bewijst in het bijzijn van bronnen van niet-determinisme zoals superklassen en flexibele contexten. Het bewijs is niet-triviaal omdat de semantiek resolutie vertaalt naar een doeltaal waar een onderscheid gemaakt kan worden tussen verschillende vertalingen in contexten die geen starttaal wederhelft hebben. Gebaseerd op de notie van volledige abstractie presenteren we een twee-staps strategie welke eerst op een niet-deterministische manier vertaalt naar een intermediaire taal, waar contextuele equivalentie gerespecteerd wordt, en vervolgens op een deterministische manier van daar naar de doeltaal. We gebruiken een methode gebaseerd op logische relaties om een contextuele equivalentie te bepalen. Van hieruit kunnen we coherentie van de eerste stap in de vertaling besluiten. Aangezien de tweede stap deterministisch is, behoudt deze triviaal de coherentie-eigenschap.

Ten derde verbeteren we de expressiviteit van typeklassen van Horn clausules tot het universele fragment van Hereditary Harrop logica. In feite zijn *gekwantificeerde klasseconstraints* al jaren geleden voorgesteld voor deze reden. Maar ondanks de grote aanhoudende vraag zijn gekwantificeerde constraints, met uitzondering van een aantal gedeeltelijke noodoplossingen, nooit formeel bestudeerd of geimplementeerd. We breiden dit idee uit tot een praktisch taalontwerp, en presenteren een declaratieve specificatie van het typesysteem. Vervolgens ontwerpen we een type interferentie algoritme dat vertaalt naar System F. Hoewel het geen een-op-een mapping betreft, heeft dit werk het startsein gegeven voor een GHC implementatie van gekwantificeerde constraints. Uiteindelijk breiden we het eerder genoemd coherentiebewijs uit met gekwantificeerde klasseconstraints—een niet triviale extensie—en tonen hiermee de aanpasbaarheid van dit bewijs aan.

We concluderen dat we de staat van polymorfisme, en de type-gestuurde codegeneratie aspecten specifiek, verbeterd hebben op drie manieren. We hebben (1) de stabiliteit van parametrisch polymorfisme verbeterd, (2) het vertrouwen in de correctheid van typeklasseresolutie verhoogd door het formeel coherent te bewijzen, en (3) de expressiviteit van typeklassen verbeterd door gekwantificeerde klasseconstraints toe te voegen. Dit maakt de weg vrij voor een meer wijdverspreid gebruik van deze functionaliteit.

Contents

Abstract iii

Beknopte samenvatting v

List of Symbols vii

Contents vii

List of Figures xiii

1 Introduction 1

1.1 Haskell 2

1.2 Aim of the Thesis 3

1.3 Thesis Overview 4

1.3.1 Part I: Parametric Polymorphism 4

1.3.2 Part II: Ad-Hoc Polymorphism 5

2 Laying the Foundations 7

2.1 Programming Languages 7

2.2 Dynamic Semantics 8

2.3 Static Semantics 8

2.4 Meta-Theory 9

I Parametric Polymorphism 11

3 Polymorphic Types 12

3.1 System F 12

3.2 Hindley Milner 13

4 Type Instantiation 15

4.1 Introduction 15
4.2 Instantiation in GHC 16
4.2.1 Deep vs. Shallow Instantiation 17
4.2.2 Eager vs. Lazy Instantiation 18

5 Meta Theory: Stability 19
5.1 Stability 19
5.1.1 Stability 25
5.2 The Mixed Polymorphic λ-Calculus 26
5.2.1 Syntax 27
5.2.2 Type system overview 29
5.2.3 Instantiation and Skolemisation 33
5.3 Evaluation 35
5.3.1 Contextual Equivalence 35
5.3.2 Properties 37
5.3.3 Conclusion 40
5.4 Instantiation in GHC 40
5.4.1 Eagerness 40
5.4.2 Depth 42
5.4.3 The situation today: Quick Look impredicativity has arrived 42
5.5 Instabilities around instantiation beyond Haskell 42
5.5.1 Explicit Instantiation 43
5.5.2 Idris 43
5.5.3 Agda 43
5.5.4 Explicit Abstraction 44
5.5.5 Implicit Generalisation 46
5.6 Example of Implicit Generalisation in Idris 48
5.7 Related Work 49
5.8 Scientific Output 49

II Ad-hoc Polymorphism 51

6 Type Classes 52
6.1 Introduction 53
6.2 Overview 53
6.2.1 Dictionary-Passing Elaboration 53
6.2.2 Alternatives 56
6.3 Source Language $\lambda_{\mathbf{TC}}$ 56
6.4 Target Language $F_{\{\}}$ 63
6.4.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ 63

7 Meta Theory: Coherence 67

7.1 Introduction . . . 67
7.2 Overview . . . 70
7.2.1 Dictionary-Passing Elaboration . . . 70
7.2.2 Nondeterminism and Coherence . . . 70
7.2.3 Contextual Difference . . . 71
7.2.4 Our Approach to Proving Coherence . . . 72
7.3 Coherence . . . 74
7.3.1 Contextual Equivalence . . . 74
7.3.2 Coherence . . . 75
7.4 Intermediate Language $F_{\mathbf{D}}$ . . . 75
7.4.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ . . . 81
7.4.2 Elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$ . . . 84
7.4.3 Elaboration Decomposition . . . 85
7.5 Coherence Revisited . . . 85
7.5.1 Coherent Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ . . . 86
7.5.2 Deterministic Elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$ . . . 91
7.6 Discussion of Possible Extensions . . . 93
7.7 Related Work . . . 95
7.8 Scientific Output . . . 96

8 Extension: Quantified Constraints 99
8.1 Introduction . . . 99
8.2 Motivation . . . 101
8.2.1 Precise and Succinct Specifications . . . 101
8.2.2 Terminating Corecursive Resolution . . . 103
8.2.3 Summary . . . 104
8.3 Declarative Type System . . . 105
8.3.1 Syntax . . . 105
8.3.2 The Type System . . . 107
8.3.3 Constraint Entailment . . . 109
8.3.4 Remaining Nondeterminism . . . 111
8.4 Type Inference . . . 113
8.4.1 Preliminaries . . . 114
8.4.2 Constraint Generation For Terms . . . 115
8.4.3 Constraint Solving . . . 115
8.4.4 Checking Declarations . . . 120
8.4.5 Program Typing . . . 121
8.5 Translation to System F . . . 121
8.5.1 Target Language: System F . . . 121
8.5.2 Elaboration of Types & Constraints . . . 122
8.5.3 Elaboration of Terms . . . 122
8.5.4 Dictionary Construction . . . 123
8.5.5 Declaration Elaboration . . . 124

8.6 Termination of Resolution 125
8.7 Related Work 127
8.8 Quantified Constraints in GHC 130
8.9 Scientific Output 131

9 Meta Theory: Coherence for Quantified Constraints 133
9.1 Introduction 133
9.2 Calculus Updates 134
9.2.1 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Updates 134
9.2.2 Example Derivation 137
9.2.3 $F_{\mathbf{D}}^{\Rightarrow}$ Updates 138
9.2.4 Example Translation 140
9.3 Meta-Theory 142
9.3.1 $F_{\mathbf{D}}^{\Rightarrow}$ Type Safety 142
9.3.2 Strong Normalisation for $F_{\mathbf{D}}^{\Rightarrow}$ 143
9.3.3 Elaboration from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\mathbf{D}}^{\Rightarrow}$ 143
9.3.4 Elaboration from $F_{\mathbf{D}}^{\Rightarrow}$ to $F_{\{\}}$ 143
9.4 Coherence 145
9.4.1 Logical Relations 145
9.4.2 Coherence Theorem Updates 146
9.5 Conclusion 150

10 Conclusion 151
10.1 Parametric Polymorphism 151
10.2 Ad-Hoc Polymorphism 154

A Additional Relations 159
A.1 MPLC Additional Definitions 159
A.2 MPLC Core Language Definitions 161
A.2.1 Translation from the Mixed Polymorphic λ-calculus . . 165
A.3 $\lambda_{\mathbf{TC}}$ Additional Definitions 170
A.3.1 Syntax 170
A.3.2 $\lambda_{\mathbf{TC}}$ Judgments and Elaboration 170
A.3.3 $\lambda_{\mathbf{TC}}$ Judgments and Elaboration through $F_{\mathbf{D}}$ 177
A.4 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Declarative Type System Additional Judgments 188
A.4.1 Well-formedness of Types & Constraints 188
A.4.2 Program Typing 188
A.4.3 Elaboration of Programs 189
A.5 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Additional Definitions 190
A.5.1 Syntax 190
A.5.2 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Judgments and Elaboration 190
A.5.3 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Judgments and Elaboration through $F_{\mathbf{D}}^{\Rightarrow}$ 197
A.5.4 Unification Algorithm 208

A.6 $F_{\mathbf{D}}$ Additional Definitions 209
A.6.1 Syntax 209
A.6.2 $F_{\mathbf{D}}$ Judgments and Elaboration 210
A.7 $F_{\mathbf{D}}^{\Rightarrow}$ Additional Definitions 216
A.7.1 Syntax 216
A.7.2 $F_{\mathbf{D}}$ Judgments and Elaboration 217
A.8 $F_{\{\}}$ Additional Definitions 224
A.8.1 Syntax 224
A.8.2 $F_{\{\}}$ Judgments 225
A.9 System F with Data Types Definitions 227
A.9.1 Term Typing 228
A.9.2 Well-formedness of Types 228
A.9.3 Program Typing 228
A.9.4 Value Binding Typing 229
A.9.5 Datatype Declaration Typing 229
A.9.6 Call-by-name Operational Semantics 229

B Stability Proofs **231**
B.1 Let-Inlining and Extraction 231
B.2 Contextual Equivalence 234
B.3 Let-Inlining and Extraction, Continued 238
B.4 Type Signatures 240
B.5 Pattern Inlining and Extraction 247
B.6 Single vs. Multiple Equations 250
B.7 η-expansion 251

C Coherence Proofs **253**
C.1 Logical Relations 253
C.1.1 Dictionary Relation 253
C.1.2 Expression Relation 255
C.1.3 Environment Relation 257
C.2 Strong Normalization Relations 258
C.2.1 Dictionary Relation 259
C.2.2 Expression Relation 259
C.3 Equivalence Relations 261
C.3.1 Kleene Equivalence Relations 261
C.3.2 Contextual Equivalence Relations 261
C.4 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Theorems 262
C.4.1 Conjectures 262
C.4.2 Lemmas 263
C.4.3 Typing Preservation 268
C.5 $F_{\mathbf{D}}$ Theorems 291
C.5.1 Conjectures 291

C.5.2 Lemmas 293
C.5.3 Type Safety 296
C.5.4 Strong Normalization 307
C.6 Elaboration Equivalence Theorems 329
C.7 Coherence Theorems 354
C.7.1 Compatibility Lemmas 354
C.7.2 Helper Theorems 378
C.7.3 Partial Coherence Theorems 412
C.7.4 Main Coherence Theorems 428
C.8 $F_{\mathbf{D}}$-to-$F_{\{\}}$ Theorems 434
C.8.1 Lemmas 434
C.8.2 Soundness 437
C.8.3 Determinism 448
C.8.4 Semantic Preservation 458

Bibliography **471**

List of Figures

2.1 Grammar for the λ calculus 8
2.2 Operational Semantics for the λ calculus 8
2.3 Typing Rules for the STLC 9
2.4 Grammar for the STLC, Extension of Figure 2.1 9

3.1 Grammar for System F, Extension of Figure 2.4 13
3.2 Typing Rules for System F, Extension of Figure 2.3 13
3.3 Grammar for HM, Extension of Figure 2.4 14
3.4 Typing Rules for HM, Extension of Figure 2.3 14

5.1 Mixed Polymorphic λ-Calculus (MPLC) Syntax 26
5.2 Term Typing for Mixed Polymorphic λ-Calculus 30
5.3 Argument and Declaration Typing for Mixed Polymorphic λ-Calculus 31
5.4 Pattern Typing for Mixed Polymorphic λ-Calculus 32
5.5 Type Instantiation and Skolemisation 33

6.1 Overview of the different calculi of Part II. 52
6.2 $\lambda_{\mathbf{TC}}$ syntax 57
6.3 $\lambda_{\mathbf{TC}}$ typing, selected rules 60
6.4 Closure and unambiguity relations 61
6.5 $\lambda_{\mathbf{TC}}$ constraint entailment 62
6.6 $\lambda_{\mathbf{TC}}$ instance declaration typing 62
6.7 Target language syntax 63

7.1 The different calculi with elaborations 73
7.2 $F_{\mathbf{D}}$, selected syntax 76
7.3 $F_{\mathbf{D}}$ typing and operational semantics, selected rules 78
7.4 $F_{\mathbf{D}}$ environment well-formedness, selected rules 79
7.5 $F_{\mathbf{D}}$ dictionary typing 79
7.6 $\lambda_{\mathbf{TC}}$ typing with elaboration to $F_{\mathbf{D}}$, selected rules 81

7.7 $\lambda_{\mathbf{TC}}$ constraint entailment with elaboration to $F_{\mathbf{D}}$ 82

8.1 Source Syntax 105
8.2 Declarative Type System (Selected Rules) 106
8.3 Tractable Constraint Entailment 110
8.4 Unambiguity 113
8.5 Constraint Generation for Terms with Elaboration 114
8.6 Constraint Entailment with Dictionary Construction 117
8.7 Declaration Elaboration 118
8.8 Subsumption Rule 120
8.9 System F Syntax 121

9.1 Updated Grammar for $\lambda^{\Rightarrow}_{\mathbf{TC}}$ with Type Annotations, Extension of Figure 8.1 135
9.2 Constraint resolution for $\lambda^{\Rightarrow}_{\mathbf{TC}}$ 136
9.3 Example Constraint Entailment Derivation 139
9.4 Grammar for $F^{\Rightarrow}_{\mathbf{D}}$, extension of Figure 7.2 140
9.5 Dictionary typing relation for $F^{\Rightarrow}_{\mathbf{D}}$ 141
9.6 Operational Semantics for $F^{\Rightarrow}_{\mathbf{D}}$, Extension of Figure 7.3 142
9.7 Strong Normalisation Relation for Dictionaries 144
9.8 Updated Logical Relations for Dictionaries 147
9.9 Updated Logical Relations for Expressions, Extension of Definitions 6 and 7 148

B.1 Relation dependencies 232
B.2 Counting Explicit Arguments 251

C.1 Dependency graph for Typing Preservation Theorems 269
C.2 Dependency graph for Strong Normalization Theorems 316
C.3 Dependency graph for Equivalence Theorems 329
C.4 Dependency graph for Coherence Theorems 409

Chapter 1

Introduction

> “He was determined to discover the underlying logic behind the universe. Which was going to be hard, because there wasn’t one.”
>
> Mort
> Terry Pratchett

The exponential growth of software development, that we have seen over the past few decades, is posing ever more complex problems for software developers [57]. The market demands increasingly complicated software, with more features, more performance, fewer bugs and with minimal development cost and time. As an example, consider the on-board software on space missions, which has steadily increased by a factor of 10, every 10 years, for the past 50 years [2].

Increasing the scope of software complexity that programmers can effectively manage, can be partly achieved by improving the programming languages and the tools at their disposal.

Static Typing One well-established approach of making software development more manageable, is through the use of a static type system. Conceptually, a type is a property of an expression or value, that denotes how the programmer intends to use this object. For example the value 42 (an Integer) supports different operations than the text *"hello world"* (a String). The type system consists of a set of judgements assigning types to expressions, and restricting which operations are allowed on which expressions. Static type systems detect certain kinds of errors early on, at compile time. For this reason, they are used extensively in many mainstream languages like C#, C++ and Java. Empirical research [78] shows a noticable, yet modest effect of type systems on the total number of bugs, but more large-scale studies are needed to measure their impact.

Implicit Type-Directed Code Generation More recently, type systems have expanded their scope beyond reducing the number of bugs, and into increasing programmer productivity by automatically generating parts of the software. In particular, repetitive code (so-called boilerplate) whose definition depends on the structure of the types, can often be automatically generated. This text focusses on such code that is generated implicitly at compile time, invisible to the programmer.

Polymorphism Since the early days of computer programming, developers have been looking for ways to stop reinventing the wheel and to reuse their existing code. For example, generic programming [63] was introduced as a way of reusing implementations. This text focusses on *polymorphism*, where a function can be reused on arguments of different types. As customary [90], we differentiate between two different forms of polymorphism: Firstly, *parametric polymorphism* allows a function to abstract over any type, and behave uniformly, independently of its type. A common example is computing the length of a list, which acts uniformly over lists of Integers, and over lists of Strings. Secondly, *ad-hoc polymorphism* allows a function to differentiate its behaviour, depending on the type of its arguments. A common example is computing the equality of two objects.

Stability We consider a language *stable* when it is robust to small, seeminingly-innocuous changes to the program code. In other words, applying common code transformations should not have a dramatic impact on the meaning of the program. This property turns out to be an import metric in languages with a mix of both implicit and explicit features. Chapter 5 explains the concept in more detail.

1.1 Haskell

This thesis text largely focusses on Haskell, as it features a state-of-the-art, powerful static type system, and takes a pioneering role with new compiler and type system features. Furthermore, Haskell features advanced systems for both parametric and ad-hoc polymorphism (through type classes).

Parametric Polymorphism Consider the function $\mathit{pair} :: \forall\; a\; b.\, a \rightarrow b \rightarrow (a, b)$ which takes two arguments and constructs a tuple containing both. The implementation is straightforward $\mathit{pair}\; x\; y = (x, y)$. Note that this function is parametrically polymorphic as it works on any two arguments, of any two types: pair performs the exact same operation in both $\mathit{pair}\; 5$ `'x'` and $\mathit{pair}\; \mathit{True}\; \mathit{id}$.

Ad-Hoc Polymorphism As a second example, consider a function $\mathit{show} :: \forall\; a.\, a \rightarrow \mathit{String}$, which serialises an argument into a String value. As serialisation depends on the type of its argument (e.g., serializing an Integer is quite different from serializing a boolean), we need to provide several distinct implementations. We do this using Haskells *type classes*, as follows:

```
class Show a where
  show :: a → String
instance Show Bool where
  show True = "True"
  show False = "False"
instance Show Int where
  show = showInt
```

Note that—unlike with parametric polymorphism, which operates on any type—we have now only defined *show* for a specific number of types.

While both parametric and ad-hoc polymorphism features in Haskell are used in academia and industry alike, the meta-theory behind these features—both in terms of correctness and stability—can certainly be explored further.

1.2 Aim of the Thesis

The goal of this thesis is to improve the current state of the art in polymorphism. In particular, we aim to answer the following research questions:

> **Question 1:** How can we improve the stability of polymorphism features?
> **Question 2:** How can we increase the expressivity of polymorphism features?
> **Question 3:** How can we increase confidence in implicit programming features for polymorphism?

We answer these questions by formally evaluating and proving important correctness and stability properties of implicit programming features. Concretely, Part I of the thesis focusses on stability properties of parametric polymorphism in Haskell. Part II focusses on a property of type classes, related to its predictability and non-ambiguity. This increases confidence in this form of implicit code generation for mission-critical software, and consequently may increase its usage in industry. Furthermore, the thesis introduces a number of new features as a case study of the aforementioned evaluations.

While the examples and use cases presented in this thesis are in the Haskell language, we focus on the evaluation of more general properties. We are thus confident that the results from this work are more broadly applicable.

1.3 Thesis Overview

This thesis consists of two main parts: Parametric Polymorphism and Ad-Hoc Polymorphism.

1.3.1 Part I: Parametric Polymorphism

The first part is concerned with the stability of parametric polymorphism in Haskell. We formalize the concept of *stability* as a way of evaluating user-facing design decisions. We then apply it to improve type instantiation with both implicit and explicit arguments.

The material found in this part is largely taken from the following publication:

> Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking stability by being lazy and shallow: lazy and shallow instantiation is user friendly. In Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021). Association for Computing Machinery, New York, NY, USA, 85–97. DOI:https://doi.org/10.1145/3471874.3472985

Chapters 2, 3 and 4 provide the necessary background knowledge. Chapter 5 introduces and evaluates the concept of stability. The formal proofs can be found in Appendix B.

1.3.2 Part II: Ad-Hoc Polymorphism

The second part focuses on the concept of *coherence* as a correctness and predictability property of type classes in Haskell. Furthermore, we discuss *quantified class constraints* as a case study.

The material found in this part is partially taken from the following publications:

> Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. Proc. ACM Program. Lang. 3, ICFP, Article 91 (August 2019), 28 pages. DOI:https://doi.org/10.1145/3341695

> Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017). Association for Computing Machinery, New York, NY, USA, 148–161. DOI:https://doi.org/10.1145/3122955.3122967

Chapter 6 provides the necessary background knowledge on ad-hoc polymorphism in Haskell. Chapter 7 introduces and evaluates the concept of coherence. Quantified class constraints are introduced in Chapter 8. Chapter 9 evaluates the impact of this extension on our proof of coherence. The adapted proof of coherence with quantified constraints can be found in Appendix C. The specific contributions of the author of this thesis are enumerated at the end of each chapter.

Chapter 2

Laying the Foundations

> "Prepare to be amazed!"
>
> George Karachalias

This chapter introduces a number of concepts used extensively throughout the thesis. Note that this text only provides a very brief taste of these fascinating topics. If you're interested to explore them in more detail, I heartily recommend the bible of programming languages: Pierce [75]. Alternatively, if you already have a basis in programming language and type theory, you can freely skip ahead to Chapter 4.

2.1 Programming Languages

While programmable computers have only existed for about 80 years, a plethora of different programming languages exist. These range dramatically in their level of expressiveness (general purpose languages vs. domain specific languages), level of abstraction (high level vs. low level languages), verification of programmer code (static typing vs. dynamic types vs. ...) etc. When working with programming languages, and even more so when investigating the meta-theory of these languages, it is often useful to have a formal specification.

The first step in constructing a formal specification for a programming language is to describe a grammar for the possible syntactical constructs. For instance, Figure 2.1 shows an inductive definition for a simple programming language (the

$$e \quad ::= \quad x \mid e_1\ e_2 \mid \lambda x.e \qquad \textit{Expression}$$

Figure 2.1: Grammar for the λ calculus

$$\boxed{e \longrightarrow e'} \qquad \text{(Operational Semantics)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \textsc{TmApp} \qquad \frac{}{(\lambda x.e_1)\ e_2 \longrightarrow [e_2/x]e_1}\ \textsc{TmAppAbs}$$

Figure 2.2: Operational Semantics for the λ calculus

untyped λ calculus). Expressions e in this language consist of variables—which, throughout this text, we will denote with x and y—, applications $e_1\ e_2$ and abstractions $\lambda x.e$.

2.2 Dynamic Semantics

Being able to execute programs is a crucial aspect of any programming language. Computation in functional programming languages can often be represented using *β-reduction*: $(\lambda x.e_1)\ e_2 \rightsquigarrow [e_2/x]e_1$. This states that applying a function $\lambda x.e_1$ to an expression e_2 is equivalent to replacing every occurrence of x in e_1 with e_2. For example, Figure 2.2 shows the operational semantics for the toy language showcased above. This example describes call-by-name semantics, where evaluation is postponed for as long as possible.

2.3 Static Semantics

As introduced in Chapter 1, static type systems are used extensively as a way of avoiding computer bugs. A sufficiently powerful type system can differentiate between a 'valid' or *well-typed* program, and a program that might potentially get stuck or crash. For illustration purposes, Figure 2.3 shows a declarative specification of a type system for the λ-calculus described in the previous section. This system is known as the *Simply Typed λ-Calculus* (STLC).

The relation $\Gamma \vdash_{\texttt{tm}} e : \tau$ denotes that, under typing environment Γ, the expression e is well-formed and is assigned the type τ. A type τ in the STLC is either

$\boxed{\Gamma \vdash_{\mathtt{tm}} e : \tau}$ (Term Typing)

$$\frac{}{\Gamma \vdash_{\mathtt{tm}} \mathit{true} : \mathit{Bool}}\ \textsc{TmTrue} \qquad \frac{}{\Gamma \vdash_{\mathtt{tm}} \mathit{false} : \mathit{Bool}}\ \textsc{TmFalse} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash_{\mathtt{tm}} x : \tau}\ \textsc{TmVar}$$

$$\frac{\begin{array}{c} x \notin \mathit{dom}(\Gamma) \\ \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e : \tau_2 \end{array}}{\Gamma \vdash_{\mathtt{tm}} \lambda(x : \tau_1).e : \tau_1 \to \tau_2}\ \textsc{TmAbs} \qquad \frac{\Gamma \vdash_{\mathtt{tm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathtt{tm}} e_2 : \tau_1}{\Gamma \vdash_{\mathtt{tm}} e_1\ e_2 : \tau_2}\ \textsc{TmApp}$$

Figure 2.3: Typing Rules for the STLC

$$\begin{array}{lll r} e & ::= & \mathit{true} \mid \mathit{false} \mid \lambda(x : \tau).e \mid \ldots & \mathit{Expression} \\ \tau & ::= & \mathit{Bool} \mid \tau_1 \to \tau_2 & \mathit{Type} \\ \Gamma & ::= & \bullet \mid \Gamma, x : \tau & \mathit{Typing\ Environment} \end{array}$$

Figure 2.4: Grammar for the STLC, Extension of Figure 2.1

a base type like *Bool*, or a function type $\tau_1 \to \tau_2$. The typing environment Γ consists of a list of term variables x that may appear free in e. *Free variables* are variables that are not introduced by a lambda binder in e, and are thus available to be substituted. We say that $\lambda(x : \tau).e$ *binds* the variable x in e. The definitions of these symbols are given in Figure 2.4. Applications $e_1\ e_2$ are typed (TmApp) by verifying that e_1 has a function type, and that the type of e_2 corresponds to the expected argument type of e_1. Abstractions $\lambda(x : \tau_1).e$ are typed (TmAbs) by extending the typing environment Γ with the newly bound variable x and its accompanying type τ_1. The abstraction itself is then assigned a function type $\tau_1 \to \tau_2$. Finally, a variable is typed (TmVar) by looking up its type in the typing environment Γ.

2.4 Meta-Theory

While including a static type system has the potential to reduce bugs, generate boilerplate code, and all-round improve developer productivity, it does come with an up-front cost. Many programming languages require the programmer to write type annotations (either on every declaration like in Java, or selectively like in Haskell), which takes additional development time.

It is thus not unreasonable to demand formal guarantees that the type system itself is in fact 'correct', in exchange for the additional effort. This is even more important in the context of mission critical software, for banks, hospitals, etc. Defining correctness is not straightforward, and often involves multiple different correctness properties.

These correctness properties can be proven using the formal specification of the language, such as the one described in this chapter. Common properties for a type system include

- *progress*: Any well-typed expression is either already a value (in our case, a boolean value or a function), or can be evaluated further. In other words, its evaluation is not stuck.
- *preservation*: When a well-typed expression takes an evaluation step, the resulting expression remains well-typed with the same type.

Commonly investigated properties for a typing algorithm include

- *soundness*: Any type inferred by the algorithm, is in fact a valid type for the given expression in the declarative specification of the type system.
- *completeness*: Any valid type for the declarative specification can also be derived by the algorithm.
- *termination*: The type system halts for every possible input program.

Part I

Parametric Polymorphism

Chapter 3

Polymorphic Types

> "Let W be a classical watch ; a mustard watch derived from W is any W' obtained from W by adding a certain amount of mustard in the mechanism."
>
> An integrated approach to time and food [81]
> Jean-Yves Girard

3.1 System F

While the STLC described in Chapter 2 is provably type safe, it is not very expressive. In fact, the calculus does not support polymorphic functions - the topic of this thesis - in any form. In order to solve this issue, Girard and Reynolds [29, 79] created an extension of the STLC with type-level abstraction: *System F*.

This extension to the syntax of the language is shown in Figure 3.1. Similarly to term variables, System F additionally features type variables, which we denote with a. For System F, we will denote types using σ, which feature abstractions $\forall a.\sigma$. Expressions e now support abstracting over a type variable as $\Lambda a.e$, as well as instantiating a type variables with a given type in $e\ \sigma$.

$$\begin{array}{lcll}
e & ::= & \Lambda a.e \mid e\ \sigma \mid \ldots & \textit{Expression} \\
\sigma & ::= & \mathit{Bool} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\sigma & \textit{Type} \\
\Gamma & ::= & \bullet \mid \Gamma, x : \tau \mid \Gamma, a & \textit{Typing Environment}
\end{array}$$

Figure 3.1: Grammar for System F, Extension of Figure 2.4

$\boxed{\Gamma \vdash_{\mathtt{ty}} \sigma}$ (Type Well-Formedness)

$$\frac{}{\Gamma \vdash_{\mathtt{ty}} \mathit{Bool}}\ \textsc{TyBool} \qquad \frac{a \in \Gamma}{\Gamma \vdash_{\mathtt{ty}} a}\ \textsc{TyVar}$$

$$\frac{\Gamma \vdash_{\mathtt{ty}} \sigma_1 \qquad \Gamma \vdash_{\mathtt{ty}} \sigma_2}{\Gamma \vdash_{\mathtt{ty}} \sigma_1 \to \sigma_2}\ \textsc{TyArrow} \qquad \frac{\Gamma, a \vdash_{\mathtt{ty}} \sigma}{\Gamma \vdash_{\mathtt{ty}} \forall a.\sigma}\ \textsc{TyForall}$$

$\boxed{\Gamma \vdash_{\mathtt{tm}} e : \sigma}$ (Term Typing)

$$\frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\mathtt{tm}} e : \sigma}{\Gamma \vdash_{\mathtt{tm}} \Lambda a.e : \forall a.\sigma}\ \textsc{TmTyAbs} \qquad \frac{\Gamma \vdash_{\mathtt{tm}} e : \forall a.\sigma \qquad \Gamma \vdash_{\mathtt{ty}} \sigma_1}{\Gamma \vdash_{\mathtt{tm}} e\ \sigma_1 : [\sigma_1/a]\sigma}\ \textsc{TmTyApp}$$

Figure 3.2: Typing Rules for System F, Extension of Figure 2.3

The new typing rules are shown in Figure 3.2. Similarly to term abstractions, typing a type abstraction (TmTyAbs) entails binding the newly declared type variable a in the typing environment Γ, and checking the remaining expression under this extended environment. Correspondingly, type applications provide a type σ with which to substitute the type variable a. Furthermore, as types now contain variables, a well-formedness relation $\Gamma \vdash_{\mathtt{ty}} \sigma$ is included to check the well-scopedness of variables.

3.2 Hindley Milner

As discussed previously, while static type systems have great potential to increase programmer productivity, they are not free, and manually writing type annotations poses a significant up-front cost. Thankfully, this is not always required. A compiler like GHC can infer most common types by itself. Under the hood, this *type inference* process is based on the well-known *Hindley-Milner* (HM) algorithm [17]. Type inference requires making a difficult trade-off between decidability and expressiveness: On one side of the spectrum, type inference for

$$\begin{array}{lcll}
e & ::= & \lambda x.e \mid \ldots & \textit{Expression} \\
\tau & ::= & \mathit{Bool} \mid \tau_1 \rightarrow \tau_2 & \textit{Monotype} \\
\sigma & ::= & \tau \mid \forall a.\sigma & \textit{Polytype}
\end{array}$$

Figure 3.3: Grammar for HM, Extension of Figure 2.4

$\boxed{\Gamma \vdash_{\texttt{tm}} e : \sigma}$ (Term Typing)

$$\frac{\begin{array}{c} x \notin \mathit{dom}(\Gamma) \\ \Gamma, x : \tau_1 \vdash_{\texttt{tm}} e : \tau_2 \qquad \Gamma \vdash_{\texttt{ty}} \tau_1 \end{array}}{\Gamma \vdash_{\texttt{tm}} \lambda x.e : \tau_1 \rightarrow \tau_2}\ \textsc{TmAbs} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\texttt{tm}} e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash_{\texttt{tm}} e_2 : \tau_1 \end{array}}{\Gamma \vdash_{\texttt{tm}} e_1\ e_2 : \tau_2}\ \textsc{TmApp}$$

$$\frac{\begin{array}{c} a \notin \Gamma \\ \Gamma, a \vdash_{\texttt{tm}} e : \sigma \end{array}}{\Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma}\ \textsc{TmGen} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma \\ \Gamma \vdash_{\texttt{ty}} \tau \end{array}}{\Gamma \vdash_{\texttt{tm}} e : [\tau/a]\sigma}\ \textsc{TmInst} \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash_{\texttt{tm}} x : \sigma}\ \textsc{TmVar}$$

Figure 3.4: Typing Rules for HM, Extension of Figure 2.3

a language like STLC is trivial, but the language is too limiting to be usable in practice. On the other side of the spectrum, full type inference for an expressive language like System F has been proven to be undecidable. This is the main reason behind the success of the HM algorithm, as it strikes a great balance, providing decidable inference for a highly expressive language.

Striking this balance thus means making a small restriction on the System F language. While HM does support type abstraction, it only allows these forall abstractions on front of the type. Figure 3.3 shows the updated grammar for the language, which shows this new distinction between types. *Polytypes* or *type schemes* σ allow for type abstraction, while *monotypes* τ are the simple types from Section 2.3.

Figure 3.4 presents a declarative specification of the HM typing system. The actual type inference algorithm, *Algorithm W* is out of scope for this text, and can be found in [59]. Note that this specification is no longer syntax directed, and allows for generalisation (TmGen) and instantiation (TmInst) of polytypes when needed. Also note that, since both function arguments and results are restricted to monotypes, TmInst might have to applied before rules TmAbs and TmApp. Type well-formedness operates identically to the System F relation (Figure 3.2), but is now split in separate relations for monotypes and polytypes. The relation is omitted due to space constraints.

Chapter 4

Type Instantiation

> This idea that there is generality in the specific is of far-reaching importance.
>
> Gödel, Escher, Bach
> Douglas R. Hofstadter

4.1 Introduction

Programmers naturally wish to get the greatest possible utility from their work. They thus embrace *polymorphism*: the idea that one function can work with potentially many types. A simple example is $\mathit{const} :: \forall\ a\ b.\ a \rightarrow b \rightarrow a$, which returns its first argument, ignoring its second. The question then becomes: what concrete types should const work with at a given call site? For example, if we say const True `'x'`, then a compiler needs to figure out that a should become Bool and b should become Char. The process of taking a type variable and substituting in a concrete type is called *instantiation*. Choosing a correct instantiation is important; for const, the choice of $a \mapsto \mathit{Bool}$ means that the return type of const True `'x'` is Bool. A context expecting a different type would lead to a type error.

In the above example, the choices for a and b in the type of const were inferred. Haskell, among other languages, also gives programmers the opportunity to *specify* the instantiation for these arguments [24]. For example, we might say *const* @*Bool* @*Char* *True* 'x' (choosing the instantiations for both *a* and *b*) or *const* @*Bool* *True* 'x' (still allowing inference for *b*). However, once we start allowing user-directed instantiation, many thorny design issues arise. For example, will **let** *f* = *const* **in** *f* @*Bool* *True* 'x' be accepted?

Our concerns are rooted in concrete design questions in Haskell, as embodied by the Glasgow Haskell Compiler (GHC). Specifically, as Haskell increasingly has features in support of type-level programming, how should its instantiation behave? Should instantiating a type like $Int \to \forall\, a.\, a \to a$ yield $Int \to \alpha \to \alpha$ (where α is a unification variable), or should instantiation stop at the regular argument of type *Int*, thus resulting in an unchanged type? This is a question of the *depth* of instantiation. Suppose now $f :: Int \to \forall\, a.\, a \to a$. Should *f* 5 have type $\forall\, a.\, a \to a$ or $\alpha \to \alpha$? This is a question of the *eagerness* of instantiation. As we explore in Section 5.1, these questions have real impact on our users.

Unlike much type-system research, our goal is *not* simply to make a type-safe and expressive language. Type-safe instantiation is well understood [e.g., 17, 79]. Instead, we wish to examine the *stability* of a design around instantiation. Intuitively, a language is stable if small, seemingly-innocuous changes to the source code of a program do not cause large changes to the program's behaviour; we expand on this definition in Section 5.1. We use stability as our metric for evaluating instantiation schemes in GHC.

Though we apply stability as the mechanism of studying instantiation within Haskell, we believe our approach is more widely applicable, both to other user-facing design questions within Haskell and in the context of other languages.

4.2 Instantiation in GHC

Visible type application and variable specificity are fixed attributes of the designs we are considering.

Visible type application Since GHC 8.0, Haskell has supported visible instantiation of type variables, based on the order in which those variables occur [24]. Given $const :: \forall\, a\ b.\, a \to b \to a$, we can write *const* @*Int* @*Bool*, which instantiates the type variables, giving us an expression of type $Int \to Bool \to Int$. If a user wants to visibly instantiate a later type parameter (say, *b*) without choosing an earlier one, they can write @_ to skip a parameter. The expression *const* @_ @*Bool* has type $\alpha \to Bool \to \alpha$, for any type α.

Specificity Eisenberg et al. [24, Section 3.1] introduce the notion of type variable *specificity*. The key idea is that quantified type variables are either written by the user (these are called *specified*) or invented by the compiler (these are called *inferred*). A specified variable is available for explicit instantiation using, e.g., $@\mathit{Int}$; an inferred variable may not be explicitly instantiated.

To understand the need for this distinction, consider the following Haskell program

$$\mathit{myPair}\ x\ y = (x, y)$$

We expect the compiler to infer a polymorphic type, but the programmer never specified the order of the type variables. Allowing explicit type instantiation for these compiler infered variables could thus result in fragile and hard to predict behaviour.

Following GHC, we use braces to denote inferred variables. Thus, if we have the program

$$\begin{aligned}&\mathit{id}_1 :: a \to a\\&\mathit{id}_1\ x = x\\&\mathit{id}_2\ x = x\end{aligned}$$

then we would write that $\mathit{id}_1 :: \forall\ a.\ a \to a$ (with a specified a) and $\mathit{id}_2 :: \forall\ \{a\}.\ a \to a$ (with an inferred a). Accordingly, $\mathit{id}_1\ @\mathit{Int}$ is a function of type $\mathit{Int} \to \mathit{Int}$, while $\mathit{id}_2\ @\mathit{Int}$ is a type error.

4.2.1 Deep vs. Shallow Instantiation

The first aspect of instantiation we seek to vary is its *depth*, which type variables get instantiated. Concretely, shallow instantiation affects only the type variables bound before any explicit arguments. Deep instantiation, on the other hand, also instantiates all variables bound after any number of explicit arguments. For example, consider a function $f :: \forall\ a.\ a \to (\forall\ b.\ b \to b) \to \forall\ c.\ c \to c$. A shallow instantiation of f's type instantiates only a, whereas deep instantiation also affects c, despite c's deep binding site. Neither instantiation flavour touches b however, as b is not an argument of f.

Versions of GHC up to 8.10 perform deep instantiation, as originally introduced by Peyton Jones et al. [73], but GHC 9.0 changes this design, as proposed by Peyton Jones [70] and inspired by Serrano et al. [85]. In this chapter, we study this change through the lens of stability.

4.2.2 Eager vs. Lazy Instantiation

Our work also studies the *eagerness* of instantiation, which determines the location in the code where instantiation happens. Eager instantiation immediately instantiates a polymorphic type variable as soon as it is mentioned. In contrast, lazy instantiation holds off instantiation as long as possible until instantiation is necessary in order to, say, allow a variable to be applied to an argument.

For example, consider these functions:

$$\begin{aligned}&\mathit{pair} :: \forall\ a.\ a \to \forall\ b.\ b \to (a, b)\\&\mathit{pair}\ x\ y = (x, y)\\&\mathit{myPairX}\ x = \mathit{pair}\ x\end{aligned}$$

What type do we expect to infer for $\mathit{myPairX}$? With eager instantiation, the type of a polymorphic expression is instantiated as soon as it occurs. Thus, $\mathit{pair}\ x$ will have a type $\beta \to (\alpha, \beta)$, assuming we have guessed $x :: \alpha$. (We use Greek letters to denote unification variables.) With neither α nor β constrained, we will generalise both, and infer $\forall\ \{a\}\ \{b\}.\ a \to b \to (a, b)$ for $\mathit{myPairX}$. Crucially, this type is *different* than the type of pair.

Let us now replay this process with lazy instantiation. The variable pair has type $\forall\ a.\ a \to \forall\ b.\ b \to (a, b)$. In order to apply pair of that type to x, we must instantiate the first quantified type variable a to a fresh unification variable α, yielding the type $\alpha \to \forall\ b.\ b \to (\alpha, b)$. This is indeed a function type, so we can consume the argument x, yielding $\mathit{pair}\ x :: \forall\ b.\ b \to (\alpha, b)$. We have now type-checked the expression $\mathit{pair}\ x$, and thus we take the parameter x into account and generalise this type to produce the inferred type $\mathit{myPairX} :: \forall\ \{a\}.\ a \to \forall\ b.\ b \to (a, b)$. This is the same as the type given for pair, modulo the specificity of a.

As we have seen, thus, the choice of eager or lazy instantiation can change the inferred types of definitions. In a language that allows visible instantiation of type variables, the difference between these types is user-visible. With lazy instantiation, $\mathit{myPairX}\ \mathit{True}\ @\mathit{Char}$ `'x'` is accepted, whereas with eager instantiation, it would be rejected.

Chapter 5

Meta Theory: Stability

5.1 Stability

We have described stability as a measure of how small transformations—call them *similarities*—in user-written code might drastically change the behaviour of a program. This section lays out the specific similarities we will consider with respect to our instantiation flavours. There are naturally *many* transformations one might think of applying to a source program. We have chosen ones that relate best to instantiation; others (e.g. does a function behave differently in curried form as opposed to uncurried form?) do not distinguish among our flavours and are thus less interesting in our concrete context. We include examples demonstrating each of these, showing how instantiation can become muddled. While these examples are described in terms of types inferred for definitions that have no top-level signature, many of the examples can easily be adapted to include a signature. After presenting our formal model of Haskell instantiation, we check our instantiation flavours against these similarities in Section 5.3, with proofs in Appendix B.

We must first define what we mean by the "behaviour" of a program. We consider two different notions of behaviour, both the *compile time* semantics of a program (that is, whether the program is accepted and what types are assigned to its variables) and its *runtime* semantics (that is, what the program executes to, assuming it is still well typed). We write, for example, $\overset{C+R}{\Longleftrightarrow}$ to denote a similarity that we expect to respect both compile and runtime semantics. Concretely, this means that applying the transformation preserves both the type and the resulting value of the expression. Similarly, $\overset{R}{\Leftrightarrow}$ is one that we expect only to respect runtime semantics, but may change compile time semantics. Thirdly, $\overset{C+R}{\Longrightarrow}$ denotes a one-directional similarity that we expect to respect both compile and runtime semantics.

Similarity 1: Let-Inlining and Extraction

A key concern for us is around **let** -inlining and -extraction. That is, if we bind an expression to a new variable and use that variable instead of the original expression, does our program change meaning? Or if we inline a definition, does our program change meaning? These notions are captured in Similarity 1:[1]

$$\textbf{let } x = e_1 \textbf{ in } e_2 \overset{C+R}{\Longleftrightarrow} [e_1/x]\, e_2$$

Example 1: *myId* The Haskell standard library defines $\mathit{id} :: \forall\, a.\, a \rightarrow a$ as the identity function. Suppose we made a synonym of this (using the implicit top-level **let** of Haskell files), with the following:

$$\mathit{myId} = \mathit{id}$$

Note that there is no type signature. Even in this simple example, our choice of instantiation eagerness changes the type we infer:

myId	eager	lazy
deep or shallow	$\forall\,\{a\}.\, a \rightarrow a$	$\forall\, a.\, a \rightarrow a$

Under eager instantiation, the mention of *id* is immediately instantiated, and thus we must re-generalise in order to get a polymorphic type for *myId*. Generalising always produces inferred variables, and so the inferred type for *myId* starts with $\forall\,\{a\}$, meaning that *myId* cannot be a drop-in replacement for *id*, which might be used with explicit type instantiation. On the other hand, lazy instantiation faithfully replicates the type of *id* and uses it as the type of *myId*.

Example 2: *myPair* This problem gets even worse if the original function has a non-prenex type, like our *pair*, above. Our definition is now:

$$\mathit{myPair} = \mathit{pair}$$

With this example, both design axes around instantiation matter:

[1] A language with a strict **let** construct will observe a runtime difference between a let binding and its expansion, but this similarity would still hold with respect to type-checking.

myPair	eager	lazy
deep	$\forall\,\{a\}\,\{b\}.\,a \to b \to (a, b)$	$\forall\,a.\,a \to \forall\,b.\,b \to (a, b)$
shallow	$\forall\,\{a\}.\,a \to \forall\,b.\,b \to (a, b)$	$\forall\,a.\,a \to \forall\,b.\,b \to (a, b)$

All we want is to define a simple synonym, and yet reasoning about the types requires us to consider both depth and eagerness of instantiation.

Example 3: *myPairX* The *myPairX* example above acquires a new entanglement once we account for specificity. We define *myPairX* with this:

$$\mathit{myPairX}\ x = \mathit{pair}\ x$$

We infer these types:

myPairX	eager	lazy
deep or shallow	$\forall\,\{a\}\,\{b\}.\,a \to b \to (a, b)$	$\forall\,\{a\}.\,a \to \forall\,b.\,b \to (a, b)$

Unsurprisingly, the generalised variables end up as inferred, instead of specified.

Similarity 2: Signature Property

The second similarity annotates a let binding with the inferred type σ of the bound expression e_1. We expect this similarity to be one-directional, as dropping a type annotation may indeed change the compile time semantics of a program, as we hope programmers expect.

$$\overline{f\,\overline{\pi}_i = e_i}^{\,i} \xRightarrow{C+R} f : \sigma; \overline{f\,\overline{\pi}_i = e_i}^{\,i}, \text{ where } \sigma \text{ is the inferred type of } f$$

where $\overline{f\,\overline{\pi}_i = e_i}^{\,i}$ denotes a declaration for f, consisting of i equations, with patterns $\overline{\pi}_i$ and definitions e_i. The syntax is explained in greater detail in Section 5.2.

Example 4: *infer* Though not yet implemented, we consider a version of Haskell that includes the ability to abstract over type variables, the subject of an approved proposal for GHC [22]. With this addition, we can imagine writing *infer*:

$$\mathit{infer} = \lambda\ @a\ (x :: a) \to x$$

We would infer these types:

infer	eager	lazy
deep or shallow	$\forall \{a\}.\, a \to a$	$\forall a.\, a \to a$

Note that the eager variant will infer a type containing an inferred quantified variable $\{a\}$. this is because the expression $\lambda\ @a\ (x :: a) \to x$ is instantly instantiated; it is then **let**-generalised to get the type in the table above.

If we change our program to include these types as annotations, the eager type, with its inferred variable, will be rejected. The problem is that we cannot check an abstraction $\lambda\ @a \to \ldots$ against an expected type $\forall\ \{a\}.\ \ldots$: the whole point of having an inferred specificity is to prevent such behaviour, as an inferred variable should not correspond to either abstractions or applications in the term.

Similarity 3: Type Signatures

Changing a type signature should not affect runtime semantics—except in the case of type classes (or other feature that interrupts parametricity). Because our work elides type classes, we can state this similarity quite generally; more fleshed-out settings would require a caveat around the lack of type-class constraints.

$$f : \sigma_1; \overline{f\ \overline{\pi}_i = e_i}^{\,i} \overset{R}{\Leftrightarrow} f : \sigma_2; \overline{f\ \overline{\pi}_i = e_i}^{\,i}$$

Example 5: *swizzle* Suppose we have this function defined[2]:

$$\begin{aligned}&\mathit{undef} :: \forall\ a.\ \mathit{Int} \to a \to a\\ &\mathit{undef} = \mathit{undefined}\end{aligned}$$

Now, we write a synonym but with a slightly different type:

$$\begin{aligned}&\mathit{swizzle} :: \mathit{Int} \to \forall\ a.\ a \to a\\ &\mathit{swizzle} = \mathit{undef}\end{aligned}$$

Shockingly, *undef* and *swizzle* have different runtime behaviour: forcing *undef* diverges (unsurprisingly), but forcing *swizzle* has no effect. The reason is that the definition of *swizzle* is not as simple as it looks. In the System-F-based core language used within GHC, we have $\mathit{swizzle} = \lambda(n :: \mathit{Int}) \to \Lambda(a :: \mathit{Type}) \to \mathit{undef}\ @a\ n$. Accordingly, *swizzle* is a function, which is already a value[3].

[2] This example is inspired by Peyton Jones [70].

[3] Similarly to *swizzle*, the definition of *undef* gets translated into $\Lambda(a :: \mathit{Type}) \to \mathit{undefined}\ @(\mathit{Int} \to a \to a)$. However, this is not a value as GHC evaluates underneath the Λ binder. The evaluation relation can be found in Appendix A.2.

Under shallow instantiation, *swizzle* would simply be rejected, as its type is different than *undef*'s. The only way *swizzle* can be accepted is if it is deeply skolemised (see *Application* in Section 5.2), a necessary consequence of deep instantiation.

swizzle	eager or lazy
deep	converges
shallow	rejected

Similarity 4: Pattern-Inlining and Extraction

The fourth similarity represents changing variable patterns (written to the left of the = in a function definition) into λ-binders (written on the right of the =), and vice versa. Here, we assume the patterns $\overline{\pi}$ contain only (expression and type) variables. The three-place *wrap* relation is unsurprising. It denotes that wrapping the patterns $\overline{\pi}$ around the expression e_1 in lambda binders results in e_1'. Its definition can be found in Appendix A.1.

$$\mathbf{let}\ x\,\overline{\pi} = e_1\ \mathbf{in}\ e_2 \overset{C+R}{\Longleftrightarrow} \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2$$

$$\text{where } \mathit{wrap}\,(\overline{\pi}; e_1\ \ e_1')$$

Example 6: *infer2*, again Returning to the *infer* example, we might imagine moving the abstraction to the left of the =, yielding:

$$\mathit{infer2}\ @a\ (x :: a) = x$$

Under all instantiation schemes, *infer2* will be assigned the type $\forall\ a.\, a \rightarrow a$. Accordingly, under eager instantiation, the choice of whether to bind the variables before the = or afterwards matters.

Similarity 5: Single vs. Multiple Equations

Our language model includes the ability to define a function by specifying multiple equations. The type inference algorithm in GHC differentiates between single and multiple equation declarations (see Section 5.3), and we do not want this distinction to affect types. While normally new equations for a function would vary the patterns compared to existing equations, we simply repeat the existing equation twice; after all, the particular choice of (well-typed) pattern should not affect compile time semantics at all.

$$f\,\overline{\pi} = e \overset{C}{\Leftrightarrow} f\,\overline{\pi} = e, f\,\overline{\pi} = e$$

Example 7: *unitId1* and *unitId2* Consider these two definitions:

$$\mathit{unitId1}\ () = \mathit{id}$$

$$\begin{aligned}&\mathit{unitId2}\ () = \mathit{id}\\&\mathit{unitId2}\ () = \mathit{id}\end{aligned}$$

Both of these functions ignore their input and return the polymorphic identity function. Let us look at their types:

		eager	lazy
unitId1	deep or shallow	$\forall\,\{a\}.\,() \to a \to a$	$() \to \forall\, a.\, a \to a$
unitId2	deep or shallow	$\forall\,\{a\}.\,() \to a \to a$	$\forall\,\{a\}.\,() \to a \to a$

The lazy case for *UnitId1* is the odd one out: we see that the definition of *unitId1* has type $\forall\, a.\, a \to a$, do not instantiate it, and then prepend the () parameter. In the eager case, we see that both definitions instantiate *id* and then re-generalise.

However, the most interesting case is the treatment of *unitId2* under lazy instantiation. The reason the type of *unitId2* here differs from that of *unitId1* is that the pattern-match forces the instantiation of *id*. As each branch of a multiple-branch pattern-match must result in the same type, we have to seek the most general type that is still less general than each branch's type. Pattern matching thus performs an instantiation step (regardless of eagerness), in order to find this common type.

In the scenario of *unitId2*, however, this causes trouble: the match instantiates *id*, and then the type of *unitId2* is re-generalised. This causes *unitId2* to have a different inferred type than *unitId1*, leading to an instability.

Similarity 6: η-Expansion

And lastly, we want η-expansion not to affect types. (This change *can* reasonably affect runtime behaviour, so we would never want to assert that η-expansion maintains runtime semantics.)

$$e \overset{C}{\Leftrightarrow} \lambda x. e\, x, \ \text{ where } e \text{ has a function type}$$

Example 8: *eta* Consider these two definitions, where $\mathit{id} :: \forall\, a.\, a \to a$:

$$\begin{aligned}&\mathit{noEta} = \mathit{id}\\&\mathit{eta} \quad\;\, = \lambda x \to \mathit{id}\ x\end{aligned}$$

where we take x to be an unused variable. The two right-hand sides should have identical meaning, as eta is simply the η-expansion of $noEta$. Yet, under lazy instantiation, these two will have different types:

		eager	lazy
$noEta$	deep or shallow	$\forall\,\{a\}.\,a \to a$	$\forall\,a.\,a \to a$
eta	deep or shallow	$\forall\,\{a\}.\,a \to a$	$\forall\,\{a\}.\,a \to a$

The problem is that the η-expansion instantiates the occurrence of id in eta, despite the lazy instantiation strategy. Under eager instantiation, the instantiation happens regardless.

5.1.1 Stability

The examples in this section show that the choice of instantiation scheme matters—and that no individual choice is clearly the best. To summarise, each of our possible schemes runs into trouble with some example; this table lists the numbers of the examples that witness a problem:

	eager	lazy
deep	1, 2, 3, 4, 5, 6	5, 7, 8
shallow	1, 2, 3, 4, 6	7, 8

At this point, the best choice is unclear. Indeed, these examples are essentially where we started our exploration of this issue—with failures in each quadrant of this table, how should we design instantiation in GHC?

To understand this better, Section 5.2 presents a formalisation of GHC's type-checking algorithm, parameterised over the choice of depth and eagerness. Section 5.3 then presents properties derived from the similarities of this section and checks which variants of our type system uphold which properties. The conclusion becomes clear: lazy, shallow instantiation respects the most similarities.

We now fix the definition of stability we will work toward in this chapter:

Definition (Stability)**.** *A language is considered stable when all of the program similarities above are respected.*

We note here that the idea of judging a language by its robustness in the face of small transformations is not new; see, for example, Le Botlan and Rémy [53] or

$$
\begin{array}{llll}
\delta & ::= & \mathcal{S} \mid \mathcal{D} & \textit{Depth} \\
\epsilon & ::= & \mathcal{E} \mid \mathcal{L} & \textit{Eagerness} \\
\tau & ::= & a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} & \textit{Monotype} \\
\rho & ::= & \tau \mid \sigma \to \phi^{\delta} & \textit{Instantiated type} \\
\sigma & ::= & \rho \mid \forall\,\overline{a}.\sigma \mid \forall\,\overline{\{a\}}.\sigma \mid \sigma_1 \to \sigma_2 & \textit{Type scheme} \\
\phi^{\delta} & ::= & \rho \quad (\delta = \mathcal{D}) \quad \mid \sigma \quad (\delta = \mathcal{S}) & \textit{Instantiated result} \\
\eta^{\epsilon} & ::= & \rho \quad (\epsilon = \mathcal{E}) \quad \mid \sigma \quad (\epsilon = \mathcal{L}) & \textit{Synthesised type} \\
e & ::= & h\,\overline{arg} \mid \lambda x.e \mid \Lambda a.e \mid \textbf{let}\ decl\ \textbf{in}\ e & \textit{Expression} \\
h & ::= & x \mid K \mid e : \sigma \mid e & \textit{Application head} \\
arg & ::= & e \mid @\sigma & \textit{Application argument} \\
decl & ::= & x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \mid \overline{x\,\overline{\pi}_i = e_i}^{\,i} & \textit{Declaration} \\
\pi & ::= & x \mid K\,\overline{\pi} \mid @\sigma & \textit{Pattern} \\
\Sigma & ::= & \bullet \mid \Sigma, T\,\overline{a} \mid \Sigma, K : \overline{a}; \overline{\sigma}; T & \textit{Static context} \\
\Gamma, \Delta & ::= & \Sigma \mid \Gamma, x : \sigma \mid \Gamma, a & \textit{Context} \\
\psi & ::= & \tau \mid @a & \textit{Argument descriptor}
\end{array}
$$

Figure 5.1: Mixed Polymorphic λ-Calculus (MPLC) Syntax

Schrijvers et al. [83], who also consider a similar property. However, we believe ours is the first work to focus on it as the primary criterion of evaluation.

Our goal in this work is not to eliminate instability, which would likely be too limiting, leaving us potentially with either the Hindley-Milner implicit type system or a System F explicit one. Both are unsatisfactory. Instead, our goal is to make the consideration of stability a key guiding principle in language design. The rest of this chapter uses the lens of stability to examine design choices around ordered explicit type instantiation. We hope that this treatment serves as an exemplar for other language design tasks and provides a way to translate vague notions of an “intuitive” design into concrete language properties that can be proved or disproved. Furthermore, we believe that instantiation is an interesting subject of study, as any language with polymorphism must consider these issues, making them less esoteric than they might first appear.

5.2 The Mixed Polymorphic λ-Calculus

In order to assess the stability of our different designs, this section develops a polymorphic, stratified λ-calculus with both implicit and explicit polymorphism. We call it the Mixed Polymorphic λ-calculus, or MPLC. Our formalisation (based on Eisenberg et al. [24] and Serrano et al. [85]) features explicit type instantiation and abstraction, as well as type variable specificity. In order to support visible type application, even when instantiating eagerly, we must consider all the arguments to a function before doing our instantiation, lest some arguments be type arguments. Furthermore, type signatures are allowed in the calculus, and the bidirectional type system [76] permits higher-rank [64] functions. Some other features, such as local **let** declarations defining functions with multiple equations, are added to support some of the similarities we wish to study.

We have built this system to support flexibility in both of our axes of instantiation scheme design. That is, the calculus is parameterised over choices of instantiation depth and eagerness. In this way, our calculus is essentially a *family* of type systems: choose your design, and you can instantiate our rules accordingly.

5.2.1 Syntax

The syntax for MPLC is shown in Figure 5.1. We define two meta parameters δ and ϵ denoting the depth and eagerness of instantiation respectively. In the remainder of this chapter, grammar and relations which are affected by one of these parameters will be annotated as such. A good example of this are types ϕ^{δ} and η^{ϵ}, as explained below.

Keeping all the moving pieces straight can be challenging. We thus offer some mnemonics to help the reader: In the remainder of the chapter, aspects pertaining to **e**ager instantiation are highlighted in emerald, while **l**azy features are highlighted in lavender. Similarly, instantiation under the **s**hallow scheme is drawn using a **s**triped line, as in $\Gamma \vdash \sigma \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho$.

Types Our presentation of the MPLC contains several different type categories, used to constrain type inference. Monotypes τ represent simple ground types without any polymorphism, while type schemes σ can be polymorphic, including under arrows. In contrast, instantiated types ρ cannot have any top-level polymorphism. However, depending on the depth δ of instantiation, a ρ-type may or may not feature nested foralls on the right of function arrows. This dependency on the depth δ of type instantiation is denoted using an instantiated result type ϕ^{δ} on the right of the function arrow. Instantiating shallowly, $\phi^{\mathcal{S}}$ is a type scheme σ, but deep instantiation sees $\phi^{\mathcal{D}}$ as an instantiated type ρ. This makes sense: $\textbf{Int} \rightarrow \forall\, a.a \rightarrow a$ *is* a fully instantiated type under shallow instantiation, but not under deep. We also have synthesised types η^{ϵ} to denote the output of the type synthesis judgement $\Gamma \vdash e \Rightarrow \eta^{\epsilon}$, which infers a type from an expression. The shape of this type depends on the eagerness ϵ of type instantiation: under lazy instantiation ($\mathcal{L}$), inference can produce full type schemes σ; but under eager instantiation ($\mathcal{E}$), synthesised types η^ϵ are always instantiated types ρ: any top-level quantified variable would have been instantiated away.

Finally, an argument descriptor ψ represents a type synthesised from analysing a function argument pattern. Descriptors are assembled into type schemes σ with the $\mathit{type}\,(\overline{\psi}; \sigma_0\ \sigma)$ judgement, in Figure 5.5.

Expressions Expressions e are mostly standard; we explain the less common forms here.

As inspired by Serrano et al. [85], applications are modelled as applying a head h to a (maximally long) list of arguments $\overline{\mathit{arg}}$. The main idea is that under eager instantiation, type instantiation for the head is postponed until it has been applied to its arguments. A head h is thus defined to be either a variable x, a data constructor K, an annotated expression $e : \sigma$ or a simple expression e. This last form will not be typed with a type scheme under eager instantiation—that is, we will not be able to use explicit instantiation—but is required to enable application of a lambda expression. As we feature both term and type application, an argument arg is defined to be either an expression e or a type argument $@\sigma$.

Our syntax additionally includes explicit abstractions over type variables, written with Λ. Though the design of this feature (inspired by Eisenberg et al. [25, Appendix B]) is straightforward in our system, its inclusion drives some of the challenge of maintaining stability.

Lastly, **let** -expressions are modelled on the syntax of Haskell. These contain a single (non-recursive) declaration decl, which may optionally have a type signature $x : \sigma$, followed by the definition $\overline{x\,\overline{\pi}_i = e_i}^{\,i}$. The patterns $\overline{\pi}$ on the left of the equals sign can each be either a simple variable x, type $@\sigma$ or a saturated data constructor $K\,\overline{\pi}$.

Contexts Typing contexts Γ are entirely standard, storing both the term variables x with their types and the type variables a in scope; these type variables may appear in both terms (as the calculus features explicit type application) and types. The type constructors and data constructors are stored in a static context Σ, which forms the basis of typing contexts Γ. This static context contains the data type definitions by storing both type constructors $T\,\overline{a}$ and data constructors $K : \overline{a}; \overline{\sigma}; T$. Data constructor types contain the list of quantified variables $\overline{a}$, the argument types $\overline{\sigma}$, and the resulting type T; when

Fig. 5.2	$\Gamma \vdash e \Rightarrow \eta^{\epsilon}$	Synthesise type η^{ϵ} for e
Fig. 5.2	$\Gamma \vdash e \Leftarrow \sigma$	Check e against type σ
Fig. 5.2	$\Gamma \vdash^{H} h \Rightarrow \sigma$	Synthesise type σ for head h
Fig. 5.2	$\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'$	Check $\overline{arg}$ against σ, resulting in type σ'
Fig. 5.3	$\Gamma \vdash \mathit{decl} \Rightarrow \Gamma'$	Extend context with a decl.
Fig. 5.4	$\Gamma \vdash^{P} \overline{\pi} \Rightarrow \overline{\psi}; \Delta$	Synthesise types $\overline{\psi}$ for patterns $\overline{\pi}$, binding context Δ
Fig. 5.4	$\Gamma \vdash^{P} \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$	Check $\overline{\pi}$ against σ, with residual type σ', binding Δ
Fig. 5.5	$\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho$	Instantiate σ to ρ
Fig. 5.5	$\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'$	Skolemise σ to ρ, binding Γ'
App. A.1	$\mathit{binders}^{\delta}(\sigma) = \overline{a}; \rho$	Extract type var. binders $\overline{a}$ and residual ρ from σ
App. A.1	$\mathit{wrap}\,(\overline{\pi}; e_1\ \ e_2)$	Bind patterns $\overline{\pi}$ in e_1 to get e_2

Table 5.1: Relation Overview

$K : \overline{a}; \overline{\sigma}; T$, then the use of K in an expression would have type $\forall\,\overline{a}.\overline{\sigma} \to T\,\overline{a}$, abusing syntax slightly to write a list of types $\overline{\sigma}$ to the left of an arrow.

5.2.2 Type system overview

Table 5.1 provides a high-level overview of the different typing judgements for the MPLC. The detailed rules can be found in Figures 5.2–5.5. The starting place to understand our rules is in Figure 5.2. These judgements implement a bidirectional type system, fairly standard with the exception of their treatment of a list of arguments all at once[4].

Understanding this aspect of the system hinges on rule Tm-InfApp, which synthesises the type of the head h and uses its type to check the arguments $\overline{arg}$. The argument-checking judgement $\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'$ (inspired by Dunfield and Krishnaswami [21]) uses the function's type σ to learn what type is expected of each argument; after checking all arguments, the judgement produces a residual type σ'. The judgement's rules walk down the list, checking term arguments (rule Arg-App), implicitly instantiating specified variables where necessary (rule Arg-Inst, which spots a term-level argument e but does not consume it), uses type arguments for instantiation (rule Arg-TyApp), and eagerly instantiates inferred type arguments (rule Arg-InfInst).

[4] This is a well-known technique to reduce the number of traversals through the applications, known as *spine form* [14].

$\boxed{\Gamma \vdash^{H} h \Rightarrow \sigma}$ *(Head Type Synthesis)*

$$\textsc{H-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash^{H} x \Rightarrow \sigma} \qquad \textsc{H-Con}\ \frac{K : \overline{a} \,;\, \overline{\sigma} \,;\, T \in \Gamma}{\Gamma \vdash^{H} K \Rightarrow \forall \overline{a}.\overline{\sigma} \to T\,\overline{a}} \qquad \textsc{H-Ann}\ \frac{\Gamma \vdash e \Leftarrow \sigma}{\Gamma \vdash^{H} e : \sigma \Rightarrow \sigma}$$

$$\textsc{H-Inf}\ \frac{\Gamma \vdash e \Rightarrow \eta^{\epsilon}}{\Gamma \vdash^{H} e \Rightarrow \eta^{\epsilon}}$$

$\boxed{\Gamma \vdash e \Rightarrow \eta^{\epsilon}}$ *(Term Type Synthesis)*

$$\textsc{Tm-InfAbs}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \eta_2^{\epsilon}}{\Gamma \vdash \lambda x.e \Rightarrow \tau_1 \to \eta_2^{\epsilon}} \qquad \textsc{Tm-InfApp}\ \frac{\Gamma \vdash^{H} h \Rightarrow \sigma \quad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \quad \Gamma \vdash \sigma' \xrightarrow{inst\ \delta} \eta^{\epsilon}}{\Gamma \vdash h\,\overline{arg} \Rightarrow \eta^{\epsilon}} \qquad \textsc{Tm-InfLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \quad \Gamma' \vdash e \Rightarrow \eta^{\epsilon}}{\Gamma \vdash \mathbf{let}\ decl\ \mathbf{in}\ e \Rightarrow \eta^{\epsilon}}$$

$$\textsc{Tm-InfTyAbs}\ \frac{\Gamma, a \vdash e \Rightarrow \eta_1^{\epsilon} \quad \Gamma \vdash \forall a.\eta_1^{\epsilon} \xrightarrow{inst\ \delta} \eta_2^{\epsilon}}{\Gamma \vdash \Lambda a.e \Rightarrow \eta_2^{\epsilon}}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma}$ *(Term Type Checking)*

$$\textsc{Tm-CheckAbs}\ \frac{\Gamma \vdash \sigma \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_1 \to \sigma_2; \Gamma_1 \quad \Gamma_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2}{\Gamma \vdash \lambda x.e \Leftarrow \sigma} \qquad \textsc{Tm-CheckLet}\ \frac{\Gamma \vdash decl \Rightarrow \Gamma' \quad \Gamma' \vdash e \Leftarrow \sigma}{\Gamma \vdash \mathbf{let}\ decl\ \mathbf{in}\ e \Leftarrow \sigma} \qquad \textsc{Tm-CheckInf}\ \frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma_1 \quad \Gamma_1 \vdash e \Rightarrow \eta^{\epsilon} \quad \Gamma_1 \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \rho \quad e \neq \lambda, \Lambda, let}{\Gamma \vdash e \Leftarrow \sigma}$$

$$\textsc{Tm-CheckTyAbs}\ \frac{\sigma = \forall \overline{\{a\}}.\forall a.\sigma' \quad \Gamma, \overline{a}, a \vdash e \Leftarrow \sigma'}{\Gamma \vdash \Lambda a.e \Leftarrow \sigma}$$

Figure 5.2: Term Typing for Mixed Polymorphic λ-Calculus

$\boxed{\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'}$ *(Argument Type Checking)*

$$\textsc{Arg-Empty}\quad \frac{}{\Gamma \vdash^{A} \bullet \Leftarrow \sigma \Rightarrow \sigma}$$

$$\textsc{Arg-App}\quad \frac{\begin{array}{c}\Gamma \vdash e \Leftarrow \sigma_1 \\ \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma'\end{array}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'}$$

$$\textsc{Arg-TyApp}\quad \frac{\Gamma \vdash^{A} \overline{arg} \Leftarrow [\sigma_1/a]\,\sigma_2 \Rightarrow \sigma_3}{\Gamma \vdash^{A} @\sigma_1, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3}$$

$$\textsc{Arg-Inst}\quad \frac{\begin{array}{c}\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_2' \Rightarrow \sigma_3 \\ \sigma_2' = [\tau_1/a]\,\sigma_2\end{array}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \forall\, a.\sigma_2 \Rightarrow \sigma_3}$$

$$\textsc{Arg-InfInst}\quad \frac{\begin{array}{c}\sigma = \forall\, \{a\}.\sigma_2 \\ \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2' \Rightarrow \sigma_3 \\ \sigma_2' = [\tau_1/a]\,\sigma_2\end{array}}{\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma_3}$$

$\boxed{\Gamma \vdash decl \Rightarrow \Gamma'}$ *(Declaration Checking)*

$$\textsc{Decl-NoAnnSingle}\quad \frac{\begin{array}{c}\Gamma \vdash^{P} \overline{\pi} \Rightarrow \overline{\psi}; \Delta \\ \Gamma, \Delta \vdash e \Rightarrow \eta^{\epsilon} \\ type\,(\overline{\psi}; \eta^{\epsilon}\ \sigma) \quad \overline{a} = \mathbf{fv}\,(\sigma) \setminus \mathbf{dom}\,(\Gamma)\end{array}}{\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \forall\,\overline{\{a\}}.\sigma}$$

$$\textsc{Decl-NoAnnMulti}\quad \frac{\begin{array}{c}j > 1 \quad \overline{\Gamma \vdash^{P} \overline{\pi}_j \Rightarrow \overline{\psi}; \Delta_j}^{\,j} \\ \overline{\Gamma, \Delta_j \vdash e_j \Rightarrow \eta_j^{\epsilon}}^{\,j} \\ \overline{\Gamma, \Delta_j \vdash \eta_j^{\epsilon} \xrightarrow{inst\ \delta} \rho}^{\,j} \quad type\,(\overline{\psi}; \rho\ \sigma) \\ \overline{a} = \mathbf{fv}\,(\sigma) \setminus \mathbf{dom}\,(\Gamma) \quad \sigma' = \forall\,\overline{\{a\}}.\sigma\end{array}}{\Gamma \vdash \overline{x\,\overline{\pi}_j = e_j}^{\,j} \Rightarrow \Gamma, x : \sigma'}$$

$$\textsc{Decl-Ann}\quad \frac{\overline{\Gamma \vdash^{P} \overline{\pi}_j \Leftarrow \sigma \Rightarrow \sigma_j'; \Delta_j}^{\,j} \quad \overline{\Gamma, \Delta_j \vdash e_j \Leftarrow \sigma_j'}^{\,j}}{\Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_j = e_j}^{\,j} \Rightarrow \Gamma, x : \sigma}$$

Figure 5.3: Argument and Declaration Typing for Mixed Polymorphic λ-Calculus

$\boxed{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}$ *(Pattern Synthesis)*

$$\textsc{Pat-InfEmpty}\quad \frac{}{\Gamma \vdash^P \bullet \Rightarrow \bullet; \bullet}$$

$$\textsc{Pat-InfVar}\quad \frac{\Gamma, x : \tau_1 \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}{\Gamma \vdash^P x, \overline{\pi} \Rightarrow \tau_1, \overline{\psi}; x : \tau_1, \Delta}$$

$$\textsc{Pat-InfCon}\quad \frac{\begin{array}{c} K \ : \ \overline{a}_0 \ ; \ \overline{\sigma}_0 \ ; \ T \ \in \ \Gamma \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0 / \overline{a}_0]\, (\overline{\sigma}_0 \to T\, \overline{a}_0) \Rightarrow T\, \overline{\tau}; \Delta_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}; \Delta_2 \end{array}}{\Gamma \vdash^P (K\, @\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Rightarrow T\, \overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2}$$

$$\textsc{Pat-InfTyVar}\quad \frac{\Gamma, a \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta}{\Gamma \vdash^P @a, \overline{\pi} \Rightarrow @a, \overline{\psi}; a, \Delta}$$

$\boxed{\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta}$ *(Pattern Checking)*

$$\textsc{Pat-CheckEmpty}\quad \frac{}{\Gamma \vdash^P \bullet \Leftarrow \sigma \Rightarrow \sigma; \bullet}$$

$$\textsc{Pat-CheckVar}\quad \frac{\Gamma, x : \sigma_1 \vdash^P \overline{\pi} \Leftarrow \sigma_2 \Rightarrow \sigma'; \Delta}{\Gamma \vdash^P x, \overline{\pi} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma'; x : \sigma_1, \Delta}$$

$$\textsc{Pat-CheckCon}\quad \frac{\begin{array}{c} K \ : \ \overline{a}_0 \ ; \ \overline{\sigma}_0 \ ; \ T \ \in \ \Gamma \quad \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_1 \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0 / \overline{a}_0]\, (\overline{\sigma}_0 \to T\, \overline{a}_0) \Rightarrow \rho_1; \Delta_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2 \end{array}}{\Gamma \vdash^P (K\, @\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2}$$

$$\textsc{Pat-CheckForall}\quad \frac{\begin{array}{c} \Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \\ \overline{\pi} \neq \cdot \ \mathit{and}\ \overline{\pi} \neq @\sigma, \overline{\pi}' \end{array}}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta}$$

$$\textsc{Pat-CheckTyVar}\quad \frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow [a/b]\, \sigma_1 \Rightarrow \sigma_2; \Delta}{\Gamma \vdash^P @a, \overline{\pi} \Leftarrow \forall\, b.\sigma_1 \Rightarrow \sigma_2; a, \Delta}$$

$$\textsc{Pat-CheckInfForall}\quad \frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \quad \overline{\pi} \neq \cdot}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; a, \Delta}$$

Figure 5.4: Pattern Typing for Mixed Polymorphic λ-Calculus

$\boxed{\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho}$ *(Type instantiation)*

$$\textsc{Inst-Inst}\quad \frac{binders^{\delta}(\sigma) = \overline{a}; \rho}{\Gamma \vdash \sigma \xrightarrow{inst\ \delta} [\overline{\tau}/\overline{a}]\, \rho}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'}$ *(Type skolemisation)*

$$\textsc{Skol-Skol}\quad \frac{binders^{\delta}(\sigma) = \overline{a}; \rho}{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma, \overline{a}}$$

$\boxed{type\,(\overline{\psi}; \sigma\ \sigma')}$ *(Telescope Type Construction)*

$$\textsc{Type-Empty}\quad \frac{}{type\,(\bullet; \sigma\ \sigma)} \qquad \textsc{Type-Var}\quad \frac{type\,(\overline{\psi}; \sigma_2\ \sigma_2')}{type\,(\tau_1, \overline{\psi}; \sigma_2\ \tau_1 \to \sigma_2')} \qquad \textsc{Type-TyVar}\quad \frac{type\,(\overline{\psi}; \sigma\ \sigma')}{type\,(@a, \overline{\psi}; \sigma\ \forall\, a.\sigma')}$$

Figure 5.5: Type Instantiation and Skolemisation

Our type system also includes **let** -declarations, which allow for the definition of functions, with or without type signatures, and supporting multiple equations defined by pattern-matching. Checking declarations and dealing with patterns is accomplished by the judgements in Figures 5.3 and 5.4, respectively, although the details may be skipped on a first reading: we include these rules for completeness and as the basis of our stability-oriented evaluation (Section 5.3). These rules do not directly offer insight into our treatment of instantiation.

Instead, the interesting aspects of our formulation are in the instantiation and skolemisation judgements.

5.2.3 Instantiation and Skolemisation

When we are type-checking the application of a polymorphic function, we must *instantiate* its type variables: this changes a function *id* $:: \forall\ a.\, a \to a$ into *id* $:: \tau \to \tau$, where τ is any monotype. On the other hand, when we are type-checking the body of a polymorphic definition, we must *skolemise* its type variables: this changes a definition $(\lambda x \to x) :: \forall\ a.\, a \to a$ so that we assign x to have type a, where a is a *skolem constant*—a fresh type, unequal to any other. These constants are bound in the context returned from the skolemisation judgement.

Naturally, the behaviour of both instantiation and skolemisation depend on the instantiation depth; see Figure 5.5. Both rule Inst-Inst and rule Skol-Skol use the *binders* helper function: $\mathit{binders}^{\delta}(\sigma) = \overline{a}; \rho$ extracts out bound type variables $\overline{a}$ and a residual type ρ from a polytype σ. The depth, though, is key: the shallow ($\mathcal{S}$) version of our type system, *binders* gathers only type variables bound at the top, while the deep ($\mathcal{D}$) version looks to the right past arrows. As examples, we have $\mathit{binders}^{\mathcal{S}}(\forall\, a.a \to \forall\, b.b \to b) = a; a \to \forall\, b.b \to b$ and $\mathit{binders}^{\mathcal{D}}(\forall\, a.a \to \forall\, b.b \to b) = a, b; a \to b \to b$. The full definition (inspired by Peyton Jones et al. [73, Section 4.6.2]) is in Appendix A.1.

Some usages of these relations happen only for certain choices of instantiation flavour. For example, see rule Tm-InfApp. We see the last premise instantiates the result of the application—but its emerald colour tells us that this instantiation happens only under the eager flavour[5]. Indeed, this particular use of instantiation is the essence of eager instantiation: even after a function has been applied to all of its arguments, the eager scheme continues to instantiate. Similarly, rule Tm-InfTyAbs instantiates eagerly in the eager flavour.

The lazy counterpart to the eager instantiation in rule Tm-InfApp is the instantiation in rule Tm-CheckInf. This rule is the catch-all case in the checking judgement, and it is used when we are checking an application against an expected type, as in the expression $f\ a\ b\ c :: T\ \mathit{Int}\ \mathit{Bool}$. In this example, if $f\ a\ b\ c$ still has a polymorphic type, then we will need to instantiate it in order to check the type against the monomorphic $T\ \mathit{Int}\ \mathit{Bool}$. This extra instantiation would always be redundant in the eager flavour (the application is instantiated eagerly when inferring its type) but is vital in the lazy flavour.

Several other rules interact with instantiation in interesting ways:

λ-expressions Rule Tm-CheckAbs checks a λ-expression against an expected type σ. However, this expected type may be a polytype. We thus must first skolemise it, revealing a function type $\sigma_1 \to \sigma_2$ underneath (if this is not possible, type checking fails). In order to support explicit type abstraction inside a lambda binder $\lambda x.\Lambda a.e$, rule Tm-CheckAbs never skolemises under an arrow: note the fixed $\mathcal{S}$ visible in the rule. As an example, this is necessary in

[5] We can also spot this fact by examining the metavariables. Instantiation takes us from a σ-type to a ρ-type, but the result in rule Tm-InfApp is a η^{ϵ}-type: a ρ-type in the eager flavour, but a σ-type in the lazy flavour.

order to accept $(\lambda x\ @b\ (y :: b) \to y) :: \forall\ a.\ a \to \forall\ b.\ b \to b$, where it would be disastrous to deeply skolemise the expected type when examining the outer λ.

Declarations without a type annotation Rule Decl-NoAnnMulti is used for synthesising a type for a multiple-equation function definition that is not given a type signature. When we have multiple equations for a function, we might imagine synthesising different polytypes for each equation. We could then imagine trying to find some type that each equation's type could instantiate to, while still retaining as much polymorphism as possible. This would seem to be hard for users to predict, and hard for a compiler to implement. Our type system here follows GHC in instantiating the types of all equations to be a monotype, which is then re-generalised. This extra instantiation is not necessary under eager instantiation, which is why it is coloured in lavender.

For a single equation (rule Decl-NoAnnSingle), synthesising the original polytype, without instantiation and regeneralisation is straightforward, and so that is what we do (also following GHC).

5.3 Evaluation

This section evaluates the impact of the type instantiation flavour on the stability of the programming language. To this end, we define a set of eleven properties, based on the informal definition of stability from Section 5.1. Every property is analysed against the four instantiation flavours, the results of which are shown in Table 5.2, which also references the proof appendix for each of the properties, in the column labeled App.

We do not investigate the type safety of our formalism, as the MPLC is a subset of System F. We can thus be confident that programs in our language can be assigned a sensible runtime semantics without going wrong.

5.3.1 Contextual Equivalence

Following the approach of GHC, rather than providing an operational semantics of our type system directly, we instead define an elaboration of the surface language presented in this chapter to explicit System F, our core language. It is important to remark that elaborating deep instantiation into this core language involves semantics-changing η-expansion. This allows us to understand the behaviour of Example 5, *swizzle*, which demonstrates a change in runtime

					$\mathcal{E}$		$\mathcal{L}$	
Sim.	Property		Phase	App.	$\mathcal{S}$	$\mathcal{D}$	$\mathcal{S}$	$\mathcal{D}$
1	1	Let inlining	C	B.1	✓	✓	✓	✓
	2	Let extraction	C	B.1	✗	✗	✓	✓
	3		R	B.3	✓	✗	✓	✗
2	4	Signature prop.	C		✗	✗	✗	✗
	4b	*restricted*		B.4	✗	✗	✓	✓
	5		R	B.4	✓	✗	✓	✗
3	6	Type signatures	R	B.4	✓	✗	✓	✗
4	7	Pattern inlining	C	B.5	✗	✗	✓	✓
	8		R	B.5	✓	✗	✓	✓
	9	Pattern extraction	C	B.5	✗	✗	✓	✓
5	10	Single/multi	C	B.6	✓	✓	✗	✗
6	11	η-expansion	C		✗	✗	✗	✗
	11b	*restricted*		B.7	✗	✓	✗	✗

Table 5.2: Property Overview

semantics arising from a type signature. This change is caused by η-expansion, observable only in the core language.

The definition of this core language and the elaboration from MPLC to core are in Appendix A.2. The meta variable e refers to core terms, and $\rightsquigarrow$ denotes elaboration. In the core language, η-expansion is expressed through the use of an expression wrapper $\dot{t}$, an expression with a hole, which retypes the expression that gets filled in. The full details can be found in Appendix A.2. We now provide an intuitive definition of contextual equivalence in order to describe what it means for runtime semantics to remain unchanged.

Definition 1 (Contextual Equivalence)**.** *Two core expressions e_1 and e_2 are contextually equivalent, written $e_1 \simeq e_2$, if there does not exist a context that can distinguish them. That is, e_1 and e_2 behave identically in all contexts.*

Here, we understand a context to be a core expression with a hole, similar to an expression wrapper, which instantiates the free variables of the expression that gets filled in. More concretely, the expression built by inserting e_1 and e_2 to the context should either both evaluate to the same value, or both diverge. A formal definition of contextual equivalence can be found in Appendix B.2.

5.3.2 Properties

let -inlining and extraction We begin by analysing Similarity 1, which expands to the three properties described in this subsection.

Property 1 (Let Inlining is Type Preserving).

- $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Rightarrow \eta^{\epsilon}\ \supset \Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta^{\epsilon}$
- $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Leftarrow \sigma\ \supset \Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma$

Property 2 (Let Extraction is Type Preserving).

- $\Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta_2^{\epsilon}\ \wedge\ \Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon} \supset \Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Rightarrow \eta_2^{\epsilon}$
- $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma_2\ \wedge\ \Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon} \supset \Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Leftarrow \sigma_2$

Property 3 (Let Inlining is Runtime Semantics Preserving).

- $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow e_1 \wedge\ \Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow e_2\ \supset e_1 \simeq e_2$
- $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Leftarrow \sigma \rightsquigarrow e_1 \wedge\ \Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma \rightsquigarrow e_2\ \supset e_1 \simeq e_2$

As an example for why Property 2 does not hold under eager instantiation, consider $id\,@\mathbf{Int}$. Extracting the id function into a new **let** -binder fails to type check, because id will be instantiated and then re-generalised. This means that explicit type instantiation can no longer work on the extracted definition.

The runtime semantics properties (both these and later ones) struggle under deep instantiation. This is demonstrated by Example 5, *swizzle*, where we see that non-prenex quantification can cause η-expansion during elaboration and thus change runtime semantics.

Signature Property Similarity 2 gives rise to these properties about signatures.

Property 4 (Signature Property is Type Preserving).
$\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma'\ \wedge\ x\ :\ \sigma\ \in\ \Gamma' \supset \Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma'$

As an example of how this goes wrong under eager instantiation, consider the definition $x = \Lambda a.\lambda y.(y : a)$. Annotating x with its inferred type $\forall\,\{a\}.a \rightarrow a$ is rejected, because rule Tm-CheckTyAbs requires a *specified* quantified variable, not an *inferred* one.

However, similarly to eager evaluation, even lazy instantiation needs to instantiate the types at some point. In order to type a multi-equation declaration, a single type needs to be constructed that subsumes the types of every branch. In our type system, rule Decl-NoAnnMulti simplifies this process by first instantiating every branch type (following the example set by GHC), thus breaking Property 4. We thus introduce a simplified version of this property, limited to single equation declarations. This raises a possible avenue of future work: parameterising the type system over the handling of multi-equation declarations.

Property 4b (Signature Property is Type Preserving (Single Equation))**.**
$\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma' \;\wedge\; x \;:\; \sigma \;\in\; \Gamma' \;\supset \Gamma \vdash x : \sigma; x\,\overline{\pi} = e \Rightarrow \Gamma'$

Property 5 (Signature Property is Runtime Semantics Preserving)**.**
$\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_1$
$\wedge\;\; \Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_2 \;\;\supset\;\; e_1 \simeq e_2$

Type Signatures Similarity 3 gives rise to the following property about runtime semantics.

Property 6 (Type Signatures are Runtime Semantics Preserving)**.**
$\Gamma \vdash x : \sigma_1; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_1 = e_1$
$\wedge\;\; \Gamma \vdash x : \sigma_2; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_2 = e_2$
$\wedge\;\; \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1 \;\;\wedge\;\; \Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
$\supset\;\; \dot{t}_1[e_1] \simeq \dot{t}_2[e_2]$

Consider **let** $x : \forall\, a.\mathbf{Int} \rightarrow a \rightarrow a; x = \mathit{undefined}$ **in** x ‘e_q‘ (), which diverges. Yet under deep instantiation, this version terminates: **let** $x : \mathbf{Int} \rightarrow \forall\, a.a \rightarrow a; x = \mathit{undefined}$ **in** x ‘e_q‘ (). Under shallow instantiation, the second program is rejected, because *undefined* cannot be instantiated to the type $\mathbf{Int} \rightarrow \forall\, a.a \rightarrow a$, as that would be impredicative. You can find the typing rules for *undefined* and e_q in Appendix A.2.1.

Pattern Inlining and Extraction The properties in this section come from Similarity 4. Like in that similarity, we assume that the patterns are just variables (either implicit type variables or explicit term variables).

Property 7 (Pattern Inlining is Type Preserving)**.**
$\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma' \;\;\wedge\;\; \mathit{wrap}\,(\overline{\pi}; e_1\;\; e_2) \;\;\supset\;\; \Gamma \vdash x = e_2 \Rightarrow \Gamma'$

The failure of pattern inlining under eager instantiation will feel similar: if we take $\mathit{id}\,@a\;\; x = x : a$, we will infer a type $\forall\, a.a \rightarrow a$. Yet if we write

$id = \Lambda a.\lambda x.(x : a)$, then eager instantiation will give us the different type $\forall \{a\}.a \rightarrow a$.

Property 8 (Pattern Inlining / Extraction is Runtime Semantics Preserving).
$\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_1 \ \wedge\ \mathit{wrap}\,(\overline{\pi}; e_1\ \ e_2)$
$\wedge\ \ \Gamma \vdash x = e_2 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_2\ \ \supset\ \ e_1 \simeq e_2$

Property 9 (Pattern Extraction is Type Preserving).
$\Gamma \vdash x = e_2 \Rightarrow \Gamma'\ \ \wedge\ \ \mathit{wrap}\,(\overline{\pi}; e_1\ \ e_2)\ \ \supset\ \ \Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$

Single vs. multiple equations Similarity 5 says that there should be no observable change between the case for a single equation and multiple (redundant) equations with the same right-hand side. That gets formulated into the following property.

Property 10 (Single/multiple Equations is Type Preserving).
$\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \sigma\ \ \supset\ \ \Gamma \vdash x\,\overline{\pi} = e, x\,\overline{\pi} = e \Rightarrow \Gamma'$

This property favours the otherwise-unloved eager flavour. Imagine *f* _ = *pair*. Under eager instantiation, this definition is accepted as type synthesis produces an instantiated type. Yet if we simply duplicate this equation under lazy instantiation (realistic scenarios would vary the patterns on the left-hand side, but duplication is simpler to state and addresses the property we want), then rule Decl-NoAnnMulti will reject as it requires the type to be instantiated.

η-expansion Similarity 6 leads to the following property.

Property 11 (η-expansion is Type Preserving).

- $\Gamma \vdash e \Rightarrow \eta^{\epsilon}\ \ \wedge\ \ \mathit{numargs}(\eta^{\epsilon}) = n\ \ \supset\ \ \Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Rightarrow \eta^{\epsilon}$
- $\Gamma \vdash e \Leftarrow \sigma\ \ \wedge\ \ \mathit{numargs}(\rho) = n\ \ \supset\ \ \Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Leftarrow \sigma$

Here, $\overline{x}^{n}$ represents n variables. We use $\mathit{numargs}(\sigma)$ to count the number of explicit arguments an expression can take, possibly instantiating any intervening implicit arguments. A formal definition can be found in Figure B.2 in the appendix. However, in synthesis mode this property fails for every flavour: η^{ϵ} might be a function type $\sigma_1 \rightarrow \sigma_2$ taking a type scheme σ_1 as an argument, while we only synthesise monotype arguments. We thus introduce a restricted version of Property 11, with the additional premise that η^{ϵ} can not contain any binders to the left of an arrow.

Property 11b (η-expansion is Type Preserving (Monotype Restriction)).

- $\Gamma \vdash e \Rightarrow \eta^{\epsilon} \ \wedge \ \mathit{numargs}(\eta^{\epsilon}) = n \ \wedge \ \Gamma \vdash \eta^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \tau$
  $\supset \ \Gamma \vdash \lambda \overline{x}^{n}.e\,\overline{x}^{n} \Rightarrow \eta^{\epsilon}$
- $\Gamma \vdash e \Leftarrow \sigma \ \wedge \ \mathit{numargs}(\rho) = n \ \supset \ \Gamma \vdash \lambda \overline{x}^{n}.e\,\overline{x}^{n} \Leftarrow \sigma$

This (restricted) property fails for all but the eager/deep flavour as η-expansion forces other flavours to instantiate arguments they otherwise would not have.

5.3.3 Conclusion

A brief inspection of Table 5.2 suggests how we should proceed: choose *lazy*, *shallow* instantiation. While this configuration does not respect all properties, it is the clear winner—even more so when we consider that Property 11b (one of only two that favour another mode) must be artificially restricted in order for any of our flavours to support the property.

We should note here that we authors were surprised by this result. This work arose from the practical challenge of designing instantiation in GHC. After considerable debate among the authors of GHC, we were unable to remain comfortable with any one decision—as we see here, no choice is perfect, and so any suggestion was met with counter-examples showing how that suggestion was incorrect. Yet we had a hunch that eager instantiation was the right design. We thus formulated the similarities of Section 5.1 and went about building a formalisation and proving properties. Crucially, we did *not* select the similarities to favour a particular result, though we did choose to avoid reasonable similarities that would not show any difference between instantiation flavours. At an early stage of this work, we continued to believe that eager instantiation was superior. It was only through careful analysis, guided by our proofs and counter-examples, that we realised that lazy instantiation was winning. We are now convinced by our own analysis.

5.4 Instantiation in GHC

Given the connection between this work and GHC, we now turn to examine some practicalities of how lazy instantiation might impact the implementation.

5.4.1 Eagerness

GHC used eager instantiation from the beginning, echoing Damas and Milner [17]. However, the GHC 8 series, which contains support for explicit type application, implements an uneasy truce, sometimes using lazy instantiation (as advocated by Eisenberg et al. [24]), and sometimes eager. In contrast, GHC 9.0 uses eager instantiation everywhere. This change was made for practical reasons: eager instantiation simplifies the code somewhat. If we went back to using lazy instantiation, the recent experience in going from lazy to eager suggests we will have to combat these challenges:

Displaying inferred types The types inferred for functions are more exotic with lazy instantiation. For example, defining $f = \lambda\_ \rightarrow id$ would infer $f :: \forall\ \{a\}.\, a \rightarrow \forall\ b.\, b \rightarrow b$. These types, which could be reported by tools (including GHCi), might be confusing for users.

Monomorphism restriction Eager instantiation makes the monomorphism restriction easier to implement, because relevant constraints are instantiated.

The monomorphism restriction is a peculiarity of Haskell, introduced to avoid unexpected runtime evaluation[6]. It potentially applies whenever a variable is defined without a type annotation and without any arguments to the left of the =: such a definition is not allowed to infer a type constraint.

Eager instantiation is helpful in implementing the monomorphism restriction, as the implementation of **let**-generalisation can look for unsolved constraints and default the type if necessary. With lazy instantiation, on the other hand, we would have to infer the type and then make a check to see whether it is constrained, instantiating it if necessary. Of course, the monomorphism restriction itself introduces instability in the language (note that *plus* and $(+)$ have different types), and so perhaps revisiting this design choice is worthwhile.

Type application with un-annotated variables For simplicity, we want all variables without type signatures not to work with explicit type instantiation. ([24, Section 3.1] expands on this point.) Eager instantiation accomplishes this, because variables without type signatures would get their polymorphism via re-generalisation. On the other hand, lazy instantiation would mean that some user-written variables might remain in a variable's type, like in the type of f, just above.

Yet even with eager instantiation, if instantiation is shallow, we can *still* get the possibility of visible type application on un-annotated variables: the specified variables might simply be hiding under a visible argument. Consider *myPair* from Example 2: under eager shallow instantiation, it gets assigned the type

[6] The full description is in the Haskell Report, Section 4.5.5 [55].

$\forall\, \{a\}.\, a \rightarrow \forall\, b.\, b \rightarrow (a, b)$. This allows for visible type application despite the lack of a signature: *myPair True* @*Char*.

5.4.2 Depth

From the introduction of support for higher-rank types in GHC 6.8, GHC has done deep instantiation, as outlined by Peyton Jones et al. [73], the paper describing the higher-rank types feature. However, deep instantiation has never respected the runtime semantics of a program; Peyton Jones [70] has the details. In addition, deep instantiation is required in order to support covariance of result types in the type subsumption judgement ([73, Figure 7]). This subsumption judgement, though, weakens the ability to do impredicative type inference, as described by Serrano et al. [84] and Serrano et al. [85]. GHC has thus, for GHC 9.0, changed to use shallow subsumption and shallow instantiation.

5.4.3 The situation today: Quick Look impredicativity has arrived

A recent innovation within GHC (due for release in the next version, GHC 9.2) is the implementation of the Quick Look algorithm for impredicative type inference [85]. The design of that algorithm walks a delicate balance between expressiveness and stability. It introduces new instabilities: for example, if *f x y* requires impredicative instantiation, (**let** *unused* = 5 **in** *f*) *x y* will fail. Given that users who opt into impredicative type inference are choosing to lose stability properties, we deemed it more important to study type inference without impredicativity in analysing stability. While our formulation of the inference algorithm is easily integrated with the Quick Look algorithm, we leave an analysis of the stability of the combination as future work.

5.5 Instabilities around instantiation beyond Haskell

The concept of stability is important in languages that have a mix of implicit and explicit features—a very common combination, appearing in Coq, Agda, Idris, modern Haskell, C++, Java, C#, Scala, F#, and Rust, among others. This section walks through how a mixing of implicit and explicit features in Idris[7] and Agda[8] causes instability, alongside the features of Haskell we describe in the

[7] We work with Idris 2, as available from `https://github.com/idris-lang/Idris2`, at commit `a7d5a9a7fdfbc3e7ee8995a07b90e6a454209cd8`.

[8] We work with Agda 2.6.0.1.

main chapter. We use these languages to show how the issues we describe are likely going to arise in any language mixing implicit and explicit features—and how stability is a worthwhile metric in examining these features—not to critique these languages in particular.

5.5.1 Explicit Instantiation

Our example languages feature explicit instantiation of implicit arguments, allowing the programmer to manually instantiate a polymorphic type, for example. Explicit instantiation broadly comes in two flavours: ordered or named parameters.

5.5.2 Idris

Idris supports named parameters. If we define $\mathit{const} : \{a, b : \mathit{Type}\} \rightarrow a \rightarrow b \rightarrow a$ (this syntax is the Idris equivalent of the Haskell type $\forall\ a\ b.\ a \rightarrow b \rightarrow a$), then we can write $\mathit{const}\ \{b = \mathit{Bool}\}$ to instantiate only the second type parameter or $\mathit{const}\ \{a = \mathit{Int}\}\ \{b = \mathit{Bool}\}$ to instantiate both. Order does not matter; $\mathit{const}\ \{b = \mathit{Bool}\}\ \{a = \mathit{Int}\}$ works as well as the previous example. Named parameters may be easier to read than ordered parameters and are robust to the addition of new type variables.

Idris's approach suffers from an instability inherent with named parameters. Unlike Haskell, the order of quantified variables does not matter. Yet now, the choice of *names* of the parameters naturally *does* matter. Thus $\mathit{const} : c \rightarrow d \rightarrow c$ (taking advantage of the possibility of omitting explicit quantification in Idris) has a different interface than $\mathit{const} : a \rightarrow b \rightarrow a$, despite the fact that the type variables scope over only the type signature they appear in.

5.5.3 Agda

Agda accepts both ordered and named parameters. After defining $\mathit{const} : \{a\ b : \mathit{Set}\} \rightarrow a \rightarrow b \rightarrow a$, we can write expressions like $\mathit{const}\ \{\mathit{Int}\}$ (instantiating only a), $\mathit{const}\ \{b = \mathit{Bool}\}$, or $\mathit{const}\ \{\_\}\ \{\mathit{Bool}\}$. Despite using named parameters, order *does* matter: we cannot instantiate earlier parameters after later ones. Naming is useful for skipping parameters that the user does not wish to instantiate. Because Agda requires explicit quantification of variables used in types (except as allowed for in implicit generalisation, below), the ordering of variables must be fixed by the programmer. However, like Idris, Agda suffers from the fact that the choice of name of these local variables leaks to clients.

5.5.4 Explicit Abstraction

Binding implicit variables in named function definitions If we sometimes want to explicitly instantiate an implicit argument, we will also sometimes want to explicitly abstract over an implicit argument. A classic example of why this is useful is in the *replicate* function for length-indexed vectors, here written in Idris:

$$\begin{array}{l}
\mathit{replicate} : \{n : \mathit{Nat}\} \to a \to \mathit{Vect}\ n\ a \\
\mathit{replicate}\ \{n = Z\} \quad \_ = [\,] \\
\mathit{replicate}\ \{n = S\ \_\}\ x = x :: \mathit{replicate}\ x
\end{array}$$

Because a length-indexed vector *Vect* includes its length in its type, we need not always pass the desired length of a vector into the *replicate* function: type inference can figure it out. We thus decide here to make the $n : \mathit{Nat}$ parameter to be implicit, putting it in braces. However, in the definition of *replicate*, we must pattern-match on the length to decide what to return. The solution is to use an explicit pattern, in braces, to match against the argument n.

Idris and Agda both support explicit abstraction in parallel to their support of explicit instantiation: when writing equations for a function, the user can use braces to denote the abstraction over an implicit parameter. Idris requires such parameters to be named, while Agda supports both named and ordered parameters, just as the languages do for instantiation. The challenges around stability are the same here as they are for explicit instantiation.

Haskell has no implemented feature analogous to this. Its closest support is that for scoped type variables, where a type variable introduced in a type signature becomes available in a function body. For example:

$$\begin{array}{l}
\mathit{const} :: \forall\ a\ b.\ a \to b \to a \\
\mathit{const}\ x\ y = (x :: a)
\end{array}$$

The $\forall\ a\ b$ brings a and b into scope both in the type signature *and* in the function body. This feature in Haskell means that, like in Idris and Agda, changing the name of an apparently local variable in a type signature may affect code beyond that type signature. It also means that the top-level $\forall$ in a type signature is treated specially. For example, neither of the following examples are accepted by GHC:

$$\begin{array}{l}
\mathit{const}_1 :: \forall.\ \forall\ a\ b.\ a \to b \to a \\
\mathit{const}_1\ x\ y = (x :: a) \\
\mathit{const}_2 :: (\forall\ a\ b.\ a \to b \to a) \\
\mathit{const}_2\ x\ y = (x :: a)
\end{array}$$

In const_1, the vacuous $\forall.$ (which is, generally, allowed) stops the scoped-type variables mechanism from bringing a into scope; in const_2, the parentheses around the type serve the same function. Once again, we see how Haskell is unstable: programmers might reasonably think that syntax like $\forall\ a\ b.$ is shorthand for $\forall\ a. \forall\ b.$ or that outermost parentheses would be redundant, yet neither of these facts is true.

Binding implicit variables in an anonymous function Sometimes, binding a type variable only in a function declaration is not expressive enough, however—we might want to do this in an anonymous function in the middle of some other expression.

Here is a (contrived) example of this in Agda, where ∋ allows for prefix type annotations:

```
_∋_ : (A : Set) → A → A
A ∋ x = x

ChurchBool  :  Set₁
ChurchBool = {A : Set} → A → A → A

churchBoolToBit : ChurchBool → ℕ
churchBoolToBit b = b 1 0

one  :  ℕ
one = churchBoolToBit (λ{A} x₁ x₂ → A ∋ x₁)
```

Here, we bind the implicit variable A in the argument to $\mathit{churchBoolToBit}$. (Less contrived examples are possible; see the Motivation section of Eisenberg [22].)

Binding an implicit variable in a λ-expression is subtler than doing it in a function clause. Idris does not support this feature at all, requiring a named function to bind an implicit variable. Agda supports this feature, as written above, but with caveats: the construct only works sometimes. For example, the following is rejected:

```
id : {A : Set} → A → A
id = λ{A} x → A ∋ x
```

The fact that this example is rejected, but $\mathit{id}\ \{A\}\ x = A \ni x$ is accepted is another example of apparent instability—we might naïvely expect that writing a function with an explicit λ and using patterns to the left of an $=$ are equivalent. Another interesting aspect of binding an implicit variable in a λ-abstraction is that the name of the variable is utterly arbitrary: instead of writing $(\lambda\{A\}\ x_1\ x_2 \rightarrow A \ni x_1)$, we can write $(\lambda\{\mathit{anything} = A\}\ x_1\ x_2 \rightarrow A \ni x_1)$. This is an attempt to use Agda's support for named implicits, but the name can be, well, $\mathit{anything}$. This would appear to be a concession to the fact that the proper name for this variable, A as written in the definition of $\mathit{ChurchBool}$, can be arbitrarily far away from the usage of the name, so Agda is liberal in accepting any replacement for it.

An accepted proposal [22] adds this feature to Haskell, though it has not been implemented as of this writing. That proposal describes that the feature would be available only when we are *checking* a term against a known type, taking advantage of GHC's bidirectional type system [24, 73]. One of the motivations that inspired this work was to figure out whether we could relax this restriction. After all, it would seem plausible that we should accept a definition like $\mathit{id} = \lambda\ @a\ (x :: a) \to a$ without a type signature. (Here, the $@a$ syntax binds a to an otherwise-implicit type argument.) It will turn out that, in the end, we can do this only when we instantiate lazily—see Section 5.3.

5.5.5 Implicit Generalisation

All three languages support some form of implicit generalisation, despite the fact that the designers of Haskell famously declared that **let** should not be generalised [94] and that both Idris and Agda require type signatures on all declarations.

Haskell Haskell's **let**-generalisation is the most active, as type signatures are optional.[9] Suppose we have defined $\mathit{const}\ x\ y = x$, without a signature. What type do we infer? It could be $\forall\ a\ b.\, a \to b \to a$ or $\forall\ b\ a.\, a \to b \to a$. This choice matters, because it affects the meaning of explicit type instantiations. A natural reaction is to suggest choosing the former inferred type, following the left-to-right scheme described above. However, in a language with a type system as rich as Haskell's, this guideline does not always work. Haskell supports type synonyms (which can reorder the occurrence of variables), class constraints (whose ordering is arbitrary) [95], functional dependencies (which mean that a type variable might be mentioned *only* in constraints and not in the main body of a type) [46], and arbitrary type-level computation through type families [15, 23].

[9] Though not relevant for our analysis, some readers may want the details: Without any language extensions enabled, all declarations without signatures are generalised, meaning that defining $\mathit{id}\ x = x$ will give id the type $\forall\ a.\, a \to a$. With the `MonoLocalBinds` extension enabled, which is activated by either of `GADTs` or `TypeFamilies`, local definitions that capture variables from an outer scope are not generalised—this is the effect of the dictum that **let** should not be generalised. As an example, the g in $f\ x = \textbf{let}\ g\ y = (y, x)\ \textbf{in}\ (g\ \texttt{'a'}, g\ \mathit{True})$ is not generalised, because its body mentions the captured x. Accordingly, f is rejected, as it uses g at two different types (Char and Bool). Adding a type signature to g can fix the problem.

With all of these features potentially in play, it is unclear how to order the type variables. Thus, in a concession to language stability, Haskell brutally forbids explicit type instantiation on any function whose type is inferred; we discuss the precise mechanism in the next section.

Since GHC 8.0, Haskell allows dependency within type signatures [96], meaning that the straightforward left-to-right ordering of variables—even in a user-written type signature—might not be well-scoped. As a simple example, consider $tr :: TypeRep\ (a :: k)$, where $TypeRep :: \forall\ k.\, k \to Type$ allows runtime type representation and is part of GHC's standard library. A naive left-to-right extraction of type variables would yield $\forall\ a\ k.\, TypeRep\ (a :: k)$, which is ill-scoped when we consider that a depends on k. Instead, we must reorder to $\forall\ k\ a.\, TypeRep\ (a :: k)$. In order to support stability when instantiating explicitly, GHC thus defines a concrete sorting algorithm, called "ScopedSort", that reorders the variables; it has become part of GHC's user-facing specification. Any change to this algorithm may break user programs, and it is specified in GHC's user manual.

Idris Idris's support for implicit generalisation is harder to trigger; see Appendix 5.6 for an example of how to do it. The problem that arises in Idris is predictable: if the compiler performs the quantification, then it must choose the name of the quantified type variable. How will clients know what this name is, necessary in order to instantiate the parameter? They cannot. Accordingly, in order to support stability, Idris uses a special name for generalised variables: the variable name itself includes braces (for example, it might be $\{k:265\}$) and thus can never be parsed[10].

Agda Recent versions of Agda support a new **variable** keyword[11]. Here is an example of it in action:

```
variable
  A : Set
  l₁ l₂ : List A
```

The declaration says that an out-of-scope use of, say, A is a hint to Agda to implicitly quantify over $A : Set$. The order of declarations in a **variable** block is significant: note that l_1 and l_2 depend on A. However, because explicit instantiation by order is possible in Agda, we must specify the order

[10] Idris 1 does not use an exotic name, but still prevents explicit instantiation, using a mechanism similar to Haskell's specificity mechanism.

[11] See `https://agda.readthedocs.io/en/v2.6.0.1/language/generalization-of-declared-variables.html` in the Agda manual for an description of the feature.

of quantification when Agda does generalisation. Often, this order is derived directly from the **variable** block—but not always. Consider this (contrived) declaration:

$$\mathit{property} : \mathit{length}\ l_2 + \mathit{length}\ l_1 \equiv \mathit{length}\ l_1 + \mathit{length}\ l_2$$

What is the full, elaborated type of *property*? Note that the two lists l_1 and l_2 can have *different* element types A. The Agda manual calls this *nested* implicit generalisation, and it specifies an algorithm—similar to GHC's ScopedSort—to specify the ordering of variables. Indeed it must offer this specification, as leaving this part out would lead to instability; that is, it would lead to the inability for a client of *property* to know how to order their type instantiations.

5.6 Example of Implicit Generalisation in Idris

It is easy to believe that a language that requires type signatures on all definitions will not have implicit generalisation. However, Idris does allow generalisation to creep in, with just the right definitions.

We start with this:

$$\begin{array}{l} \textbf{data}\ \mathit{Proxy} : \{k : \mathit{Type}\} \to k \to \mathit{Type}\ \textbf{where} \\ \quad P : \mathit{Proxy}\ a \end{array}$$

The datatype *Proxy* here is polymorphic; its one explicit argument can be of any type.

Now, we define *poly*:

$$\begin{array}{l} \mathit{poly} : \mathit{Proxy}\ a \\ \mathit{poly} = P \end{array}$$

We have not given an explicit type to the type variable a in *poly*'s type. Because *Proxy*'s argument can be of any type, a's type is unconstrained. Idris *generalises* this type, giving *poly* the type $\{k : \mathit{Type}\} \to \{a : k\} \to \mathit{Proxy}\ a$.

At a use site of *poly*, we must then distinguish between the possibility of instantiating the user-written a and the possibility of instantiating the compiler-generated k. This is done by giving the k variable an unusual name, `{k:446}` in our running Idris session.

5.7 Related Work

The type systems in this work build most directly from Peyton Jones et al. [73], Eisenberg et al. [24], and Serrano et al. [85]. Each of these papers adds new capabilities to Haskell, and each also decreases the stability of the language. While these papers do consider properties we would consider to be components of stability, stability is not a key criterion in those authors' evaluation. By contrast, our work focuses squarely on stability as a believable proxy for the quality of the user experience.

Many other works on designing type inference algorithms also introduce stability properties, but these properties exist among others—such as completeness—and do not seem to guide the design of the algorithm. We do call out one such work, that by Schrijvers et al. [83], which revolves around implicit programming systems, and describes a property they call *stability*. In the context of that work, stability is about Haskell's class-instance selection mechanism: we would like the choice of instance to remain stable under substitutions. That is, if $f :: C\ a \Rightarrow a \rightarrow Int$ is called at an argument of type $Maybe\ b$ (for a type variable b), the instance selected for $C\ (Maybe\ b)$ should be the same as the one that would be selected if f were called on an argument of type $Maybe\ Int$. After all $Maybe\ b$ can be substituted to become $Maybe\ Int$; perhaps a small change to the program would indeed cause this substitution, and we would not want a change in runtime behaviour. Accordingly, the stability property, as used by Schrijvers et al. [83], is what we would also call a stability property, but it is much narrower than the definition we give the term.

In comparison to these other papers on type systems and type inference, the angle of this work is somewhat different: we are not introducing a new language or type system feature, proving a language type safe, or proving an inference algorithm sound and complete to its declarative specification. Instead, we introduce the concept of *stability* as a new metric for evaluating (new or existing) type systems, and then apply this metric to a system featuring both implicit and explicit instantiation. Because of this novel, and somewhat unconventional topic, we are unable to find further related work.

5.8 Scientific Output

This chapter introduces the concept of stability, and constructs the MPLC to evaluate the different design decisions discussed in Chapter 4.

The material found in this chapter is largely taken from the following publication:

> Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking stability by being lazy and shallow: lazy and shallow instantiation is user friendly. In Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021). Association for Computing Machinery, New York, NY, USA, 85–97. DOI:https://doi.org/10.1145/3471874.3472985

This work was largely performed while the auther of this thesis was interning at Tweag, under the supervision of Richard Eisenberg. The contributions of the different authors are as follows:

- The author of this thesis worked on all aspects of the chapter, including designing the typing rules, writing all of the proofs, and composing the text.
- Richard Eisenberg organized and guided this project, critiqued the typing rules, provided GHC/Haskell expertise, and substantially contributed to writing.

Part II

Ad-hoc Polymorphism

Chapter 6

Type Classes

Throughout Part II of this thesis, we will use no less than five different calculi. As a guide to the reader, we present these languages in different colours, and thus encourage the reader to view / print this text in colour. Chapter 6 presents our source language $\lambda_{\mathbf{TC}}$ (marked in blue), with a translation to the target language $F_{\{\}}$ (marked in yellow). Chapter 7 introduces a third language $F_{\mathbf{D}}$, as an intermediate step in this translation. This language will thus be marked in green. Chapter 8 defines quantified class constraints as a new language extension. We will mark this extension with a ⇒ and by adding a red colour, transforming $\lambda_{\mathbf{TC}}$ into $\lambda^{\Rightarrow}_{\mathbf{TC}}$ (marked in purple) and $F_{\mathbf{D}}$ into $F^{\Rightarrow}_{\mathbf{D}}$ (marked in red). These relations are shown in Figure 6.1, where an arrow represents a translation.

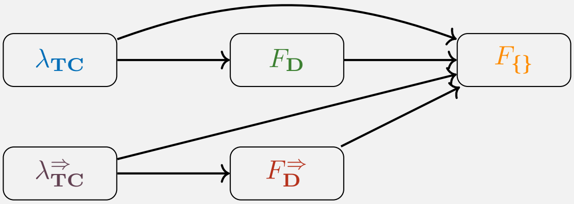
$\lambda_{\mathbf{TC}}$
$F_{\mathbf{D}}$
$F_{\{\}}$
$\lambda^{\Rightarrow}_{\mathbf{TC}}$
$F^{\Rightarrow}_{\mathbf{D}}$

Figure 6.1: Overview of the different calculi of Part II.

6.1 Introduction

Type classes were initially introduced in Haskell [69] by Wadler and Blott [95] to make ad-hoc overloading less ad hoc, and they have since become one of Haskell's core abstraction features. Moreover, their resounding success has spread far beyond Haskell: several languages have adopted them (e.g., Mercury [35], Coq [87], PureScript [26], Lean [18]), and they have inspired various alternative language features (e.g., Scala's implicits [56, 65], Rust's traits [62], C++'s concepts [31], Agda's instance arguments [19]).

Type classes have also received a lot of attention from researchers with many proposals for extensions and improvements, including functional dependencies [46], associated types [15], quantified constraints [11] among other extensions.

As described in the original work by Wadler and Blott [95], this thesis text employs an indirect, elaboration-based approach for giving meaning to programs with type classes. Indeed, the meaning of such programs is commonly given in terms of their translation to a core language [32], like System F, the meaning of which is defined in the form of an operational semantics. In this translation process, type classes are elaborated into explicitly passed function dictionaries.

6.2 Overview

This section provides some background on dictionary-passing elaboration of type class resolution. We then briefly introduce our calculi. Throughout the section we use Haskell-like syntax as the source language for examples, and to simplify our informal discussion we use the same syntax without type classes as the target language.

6.2.1 Dictionary-Passing Elaboration

The dynamic semantics for type classes are not expressed directly but rather by type-directed elaboration into a simpler language without type classes such as System F. Thus the dynamic semantics of type classes are given indirectly as the dynamic semantics of their elaborated forms.

Basic Elaboration. Consider the small program with type classes in Example 1. We declare a type class *Eq* and instances for the *Int* and pair types. The function

```
class Eq a where
  (==) :: a → a → Bool
instance Eq Int where
  (==) = primEqInt
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x1, y1) == (x2, y2) = x1 == x2 && y1 == y2
refl :: Eq a ⇒ a → Bool
refl x = x == x
main :: Bool
main = refl (5, 42)
```

Example 1: Program with type classes.

refl trivially tests whether an expression is equivalent to itself, which is called in *main*.

The dictionary-passing elaboration translates this program into a System F-like core language that does not feature type classes. The main idea of the elaboration is to map a type class declaration onto a datatype that contains the method implementations, a so-called *(function) dictionary*.

```
data EqD a = EqD {(==) :: a → a → Bool}
```

Then simple instances give rise to dictionary values:

```
eqInt :: EqD Int
eqInt = EqD {(==) = primEqInt}
```

Instances with a non-empty context are translated to functions that take context dictionaries to the instance dictionary.

```
eqPair :: (EqD a, EqD b) → Eq (a, b)
eqPair (da, db) =
  EqD {(==) = λ(x1, y1) (x2, y2) → (==) da x1 x2 &&(==) db y1 y2}
```

Functions with qualified types, like *refl*, are translated to functions that take explicit dictionaries as arguments.

```
refl :: EqD a → a → Bool
refl d x = (==) d x x
```

Finally, calls to functions with a qualified type are mapped to calls that explicitly pass the appropriate dictionary.

```
class Base a where
  base :: a → Bool
class Base a ⇒ Sub1 a where
  sub1 :: a → Bool
test1 :: Sub1 a ⇒ a → Bool
test1 x = sub1 x && base x
```

Example 2: Program with superclasses.

```
main :: Bool
main = refl (eqPair eqInt eqInt) (5, 42)
```

Elaboration of Superclasses. Superclasses require a small extension to the above elaboration scheme. Consider the small program in Example 2 where *Sub1* is a subclass of *Base*. The function *test1* has *Sub1 a* in the context and calls *sub1* and *base* in its definition.

The standard approach to encode superclass is to embed the superclass dictionary in that of the subclass. For this case, *Sub1D a* contains a field *super1* that points to the superclass:

```
data BaseD a = BaseD {base :: a → Bool}
data Sub1D a = Sub1D {super1 :: BaseD a
  , sub1 :: a → Bool}
```

This way we can extract the superclass from the subclass when needed. The function *test1* is then encoded as:

```
test1 :: Sub1 a → a → Bool
test1 d x = sub1 d x && base (super1 d) x
```

Resolution. Calls to functions with a qualified type generate type class constraints. The process for checking whether these constraints can be satisfied, is known as *resolution*. For the sake of dictionary-passing elaboration, this resolution process is augmented with the construction of the appropriate dictionary that witnesses the satisfiability of the constraint.

6.2.2 Alternatives

Morris [60] presents a specialization-based approach to type Haskell type classes. Rather than expressing the semantics of the program through elaboration into a more explicit target language, the paper represents a class as a type-indexed collection of all its ground instance types. The main advantage of this approach is that it makes reasoning over classes and properties of the type system easier, as it avoids a translation step.

However, this work sticks to the classic elaboration-based semantics for type classes, as it more closely relates to GHC, the de facto Haskell compiler. A specialization-based approach would likely simplify a proof for coherence of type class resolution (Chapter 7), as it makes the uniqueness of instances more explicit. However, an additional proof would have to be included to claim that this approach is equivalent to the familiar elaboration-based approach.

6.3 Source Language $\lambda_{\mathbf{TC}}$

This section presents our source language $\lambda_{\mathbf{TC}}$, a basic calculus which only supports features that are essential for type class resolution.

Consequently, the language is strongly normalizing, and thus does not support recursive let expressions, mutual recursion or recursive methods. This calculus will form the basis for a formal proof of coherence of the type class resolution mechanism in Chapter 7. This is a sensible choice, as recursion does not affect the fundamentals of the coherence proof. The proof could include recursion through step indexing [3], a well-known technique, but this would significantly clutter the proof. Recursion is discussed in more detail in Section 7.6.

Furthermore, two notable design decisions were made in the support of superclasses in $\lambda_{\mathbf{TC}}$. Firstly, similar to GHC, $\lambda_{\mathbf{TC}}$ derives all possible superclass constraints from their subclass constraints in advance, instead of deriving them "just-in-time" during resolution. The advantage of this approach is that it streamlines the actual resolution process.

Secondly, similar to Coq [87] and unlike Wadler and Blott [95], we pass superclass dictionaries alongside their subclass dictionaries, i.e., in a *flattened* form, instead of nesting them inside their subclass dictionaries. This design decision was taken to considerably simplify the coherence proof in Chapter 7. Neither our type class resolution mechanism, nor the intermediate language $F_{\mathbf{D}}$ (Section 7.4) need to have any support for superclasses and can treat them as regular local constraints. As it is not too difficult to see that both approaches are isomorphic,

$$
\begin{array}{llll}
\tau & ::= & \mathit{Bool} \mid a \mid \tau_1 \rightarrow \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid Q \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme} \\
 & & & \\
Q & ::= & \mathit{TC}\,\tau & \textit{class constraint} \\
C & ::= & \forall \overline{a}.\overline{Q} \Rightarrow Q & \textit{constraint scheme} \\
 & & & \\
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid m \mid \lambda x.e \mid e_1\,e_2 & \textit{term} \\
 & \mid & \mathbf{let}\ \ x : \sigma = e_1\,\mathbf{in}\ \ e_2 \mid e :: \tau & \\
 & & & \\
\mathit{pgm} & ::= & e \mid \mathit{cls}; \mathit{pgm} \mid \mathit{inst}; \mathit{pgm} & \lambda_{\mathbf{TC}}\ \textit{program} \\
\mathit{cls} & ::= & \mathbf{class}\ \ \overline{\mathit{TC}_i\ a} \Rightarrow \mathit{TC}\,a\,\mathbf{where}\ \ \{m : \sigma\} & \textit{class decl.} \\
\mathit{inst} & ::= & \mathbf{instance}\ \ \overline{Q} \Rightarrow \mathit{TC}\,\tau\,\mathbf{where}\ \ \{m = e\} & \textit{instance decl.} \\
 & & & \\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \overline{\mathit{TC}_i\ a} \Rightarrow \mathit{TC}\,a : \sigma & \textit{class environment} \\
P & ::= & \bullet \mid P, (D : C).m \mapsto \Gamma : e & \textit{program context} \\
M & ::= & [\bullet] \mid \lambda x.M \mid M\,e \mid e\,M \mid M :: \tau & \textit{evaluation context} \\
 & & \mid \mathbf{let}\ \ x : \sigma = M\,\mathbf{in}\ \ e \mid \mathbf{let}\ \ x : \sigma = e\,\mathbf{in}\ \ M &
\end{array}
$$

Figure 6.2: $\lambda_{\mathbf{TC}}$ syntax

flattening the superclasses does not impact the coherence of resolution. A more structured representation would give rise to additional complexity, but would not alter the essence of the proof.

Syntax. Figure 6.2 presents the, mostly standard, syntax. Programs consist of a number of class (with superclasses) and instance declarations, and an expression. For the sake of simplicity and well-foundedness, the declarations are ordered and can only refer to previous declarations.

Following Jones [42]'s qualified types framework, we distinguish between three sorts of types: monotypes τ, qualified types ρ which include constraints, and type schemes σ which include type abstractions. Constraints differentiate between full constraint schemes C and simple class constraints Q. Observe that we allow flexible contexts in the qualified types; they are not restricted to constraints on type variables.

The definition of expressions e is standard, but with a few notable exceptions. Firstly, the language differentiates syntactically between regular variables x and method names m, which are introduced in class declarations. Secondly, type annotations $e :: \tau$ allow the programmer to manually assign a monotype to an expression. This is useful for resolving ambiguity—see the *Typing* paragraph below. Finally, let bindings include type annotations with a type scheme σ, allowing the programmer to introduce local constraints—also discussed in the *Typing* paragraph. Note that we use Haskell syntax for class and instance declarations.

There are three $\lambda_{\mathbf{TC}}$ environments: two global ones and one local environment. Firstly, the global class environment Γ_C stores all class declarations. Each entry in Γ_C contains the method name m, any superclasses $\overline{TC_i\ a}$, the class $TC\,a$ itself and the corresponding method type σ.

Secondly, the global program context P contains all instance declarations. Each entry in P consists of a unique dictionary constructor D, its corresponding constraint C, the method name m and its implementation e, together with the context Γ under which e should be interpreted. This context contains the local axioms available in this instance declaration, as well as any axioms which explicitly annotate the method type signature.

Thirdly, the local typing environment Γ, besides containing the default term and type variables x and a, also stores any local axioms Q. As opposed to the program context P, Γ does not contain any type class instances. Instead, the (local) axioms are associated with a dictionary variable δ. Sections 7.4 and 7.4.1 explain the use of these dictionaries.

Typing. Our type system features two design choices to eliminate the possibility of ambiguous type schemes. This decision will simplify the discussion of coherence in Chapter 7, as it allows us to focus on the coherence of type class resolution, by making our proof orthogonal to ambiguous type schemes, the source of ambiguity which has already been studied by Jones [40]. We thus side-step an already solved problem and focus on tackling the full problem of resolution coherence.

Firstly, we require type signatures to be unambiguous (Figure 6.4, right-hand side) to make sure that all newly introduced type variables are bound in the head of the type (the remaining monotype after dropping all type and constraint abstractions). This prevents ambiguous expressions such as:

```
let f : ∀ a. Eq a ⇒ Int → Int   -- ambiguous
   = λx ∘ x + 1 in f 42
```

Secondly, we use a bidirectional type system rather than a fully declarative one. A bidirectional type system distinguishes between two typing modes: *inference* and *check* mode. The former synthesizes a type from the given expression, while the latter checks whether a given expression is of a given type. Special in our setting is that variables can only be typed in check mode, to ensure that only a single instantiation exists. This avoids the ambiguity that can arise when instantiating type variables in inference mode. Consider the following example:

$$\begin{array}{l}\mathbf{let}\ y : \forall\, a.\, Eq\ a \Rightarrow a \rightarrow a = \ldots \\ \mathbf{in}\ const\ 1\ y\end{array}$$

where $const\ x$ is the constant function, which evaluates to x for any input. The instantiation of y's type scheme is not uniquely determined by the context in which it is used. In a declarative type system or in inference mode, this ambiguity would result in multiple distinct typings and corresponding elaborations. While this ambiguity is harmless, it is not the focus of this work. Hence, to focus exclusively on the resolution, we use a bidirectional type system with check mode for variables to eliminate this irrelevant source of ambiguity.

Figure 6.3 [1] shows selected typing rules. The full set of rules can be found in Section 2.2 of the appendix. We ignore the red (elaboration-related) parts for now and explain them in detail in Section 6.4.1. The judgments $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau$ and $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau$ denote *infer*ring a monotype τ for expression e and *check*ing e to have a monotype τ respectively, in environments P, Γ_C and Γ. Note that the constraint and type well-formedness relations $\vdash_Q$ and $\vdash_{ty}$ are omitted, as they are standard well-scopedness checks. They can be found in Section 2.1 of the appendix.

Through a let binding (rule sTm-infT-let), the programmer provides a type scheme for a variable, thus potentially introducing local constraints. As explained above, the unambiguity check from Figure 6.4 (right-hand side) requires the provided type scheme to be unambiguous. In order to flatten the superclasses, the rule takes the closure over the superclass relation (left-hand side of Figure 6.4) of the user provided constraints $\overline{Q}_i$. It then adds the resulting set of constraints $\overline{Q}_k$ to the typing environment, under which to typecheck e_1. Finally, the type of e_2 is inferred under the extended environment.

Rule sTm-checkT-meth types a method call m in check mode, like regular variables, to avoid any ambiguity in the instantiation of the type variables in the method's type scheme. This includes both the type variable a from the class and any additional free variables $\overline{a}_j$ in the method type. Furthermore, the rules uses the unambig-relation to avoid ambiguity in the method type scheme itself, by requiring that both sets of type variables have to occur in the head of the method type. The rule also checks that all required constraints $\overline{Q}_i$ from the method type can be entailed.

[1] Note that lists, such as $\overline{\tau}_i$, are denoted by overlines, whereas collections of predicates are annotated by their range. For instance, $(\Gamma_C; \Gamma \vdash_{ty} \tau_i \rightsquigarrow \sigma_i\ ,\ \forall i)$ iterates over i.

$\boxed{P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}$ *($\lambda_{\mathbf{TC}}$ Term Inference)*

sTm-infT-let

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \qquad \mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C;\overline{Q}_i) = \overline{Q}_k \qquad (\Gamma_C;\Gamma \vdash_Q Q_k \rightsquigarrow \sigma_k \ ,\ \forall k) \\ \Gamma_C;\Gamma \vdash_{ty} \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma \qquad \overline{\delta}_k\ \mathbf{fresh} \\ P;\Gamma_C;\Gamma,\overline{a}_j,\overline{\delta}_k : \overline{Q}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P;\Gamma_C;\Gamma, x : \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \end{array}}{\begin{array}{c} P;\Gamma_C;\Gamma \vdash_{tm} \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = e_1\ \mathbf{in}\ \ e_2 \Rightarrow \tau_2 \\ \rightsquigarrow \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta_k : \sigma_k}^{k}.e_1\ \mathbf{in}\ \ e_2 \end{array}}$$

$\boxed{P;\Gamma_C;\Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$ *($\lambda_{\mathbf{TC}}$ Term Checking)*

sTm-checkT-meth

$$\frac{\begin{array}{c} (m : \overline{Q}'_k \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \qquad P;\Gamma_C;\Gamma \vDash [TC\ \tau] \rightsquigarrow e \\ \Gamma_C;\Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \qquad (P;\Gamma_C;\Gamma \vDash [[\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i] \rightsquigarrow e_i \ ,\ \forall i) \\ (\Gamma_C;\Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j \ ,\ \forall j) \qquad \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma \end{array}}{P;\Gamma_C;\Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow e.m\,\overline{\sigma}_j\,\overline{e}_i}$$

Figure 6.3: $\lambda_{\mathbf{TC}}$ typing, selected rules

The instance typing rule can be found in Figure 6.6. The relation $P;\Gamma_C \vdash_{inst} inst : P'$ denotes that an instance declaration $inst$ results in a $\lambda_{\mathbf{TC}}$ program context P', while being typed under environments P and Γ_C. The unambig-relation for constraints (Figure 6.4, bottom right), similarly to the unambig-relation for types, checks that all free type variables $\overline{b}_k$ in the instance context occur in the instance type τ as well, in order to avoid ambiguity. Like in the sTm-infT-let rule explained above, the superclasses of the instance context $\overline{Q}_p$ are flattened into additional local constraints $\overline{Q}_q$ and added to the environment. The superclasses $\overline{Q}'_i$ of the instantiated type class are then checked to be entailed under this extended environment. The rule checks that no overlapping instance declarations D' have been defined. Finally, the program context is extended with the new instance axiom D, consisting of a constraint scheme that requires the full set of local constraints $\overline{Q}_q$.

Type Class Resolution. The type class resolution rules can be found in Figure 6.5, where $P;\Gamma_C;\Gamma \vDash [Q]$ denotes that a class constraint Q is entailed

$$\boxed{\mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_j} \qquad \textit{(Superclass Closure)}$$

$$\textsc{sClosure-empty}\quad \frac{}{\mathbf{closure}(\Gamma_C; \bullet) = \bullet} \qquad \textsc{sClosure-TC}\quad \frac{\begin{array}{c} TC\,a = \mathit{head}(C) \\ (m : \overline{C}_m \Rightarrow TC\,a : \sigma) \in \Gamma_C \\ \mathbf{closure}(\Gamma_C; \overline{C}_i, \overline{C}_m) = \overline{C}_j \end{array}}{\mathbf{closure}(\Gamma_C; \overline{C}_i, C) = \overline{C}_j, C}$$

$$\boxed{\mathbf{unambig}(\sigma)} \qquad \textit{(Unambiguity for Type Schemes)}$$

$$\textsc{sUnambig-scheme}\quad \frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau)}$$

$$\boxed{\mathbf{unambig}(C)} \qquad \textit{(Unambiguity for Constraints)}$$

$$\textsc{sUnambig-constraint}\quad \frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\tau)}$$

Figure 6.4: Closure and unambiguity relations

under the environments P, Γ_C and Γ. A wanted constraint Q can either be resolved using a locally available constraint δ (sENTAILT-LOCAL) or through a global instance declaration D (sENTAILT-INST). The former is entirely straightforward. The latter is more involved as an instance D may have an instance context $\overline{Q}'_i$, which has to be recursively resolved. Before resolving the context, the type variables $\overline{a}_j$ are instantiated with the corresponding concrete types $\overline{\tau}_j$, originating from the wanted constraint Q.

Note that the type class resolution mechanism does not require any specific support for superclasses, as these have all been flattened into regular local constraints.

$\boxed{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e}$ *(Constraint Entailment)*

sEntailT-local

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow \delta}$$

sEntailT-inst

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{Q}'_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}'_i, \overline{b}_k, \overline{\delta}_y : \overline{Q}_y \qquad Q = [\overline{\tau}_j / \overline{a}_j] Q' \\ P_1; \Gamma_C; \Gamma' \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e \qquad \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ (\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j \;, \; \forall j) \qquad (\Gamma_C; \bullet, \overline{a}_j \vdash_Q Q'_i \rightsquigarrow \sigma'_i \;, \; \forall i) \\ (\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_Q Q_y \rightsquigarrow \sigma''_y \;, \; \forall y) \qquad (P; \Gamma_C; \Gamma \vDash [[\overline{\tau}_j / \overline{a}_j] Q'_i] \rightsquigarrow e_i \;, \; \forall i) \end{array}}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow (\Lambda \overline{a}_j . \lambda \, \overline{\delta'_i : \sigma'_i}^{\,i} . \{m = \Lambda \overline{b}_k . \lambda \, \overline{\delta_y : \sigma''_y}^{\,y} . e\}) \, \overline{\sigma}_j \, \overline{e}_i}$$

Figure 6.5: $\lambda_{\mathbf{TC}}$ constraint entailment

$\boxed{P; \Gamma_C \vdash_{inst} inst : P'}$ *(Instance Decl Typing)*

sInstT-inst

$$\frac{\begin{array}{c} (m : \overline{Q}'_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{Q}'_y \Rightarrow \tau_1) \in \Gamma_C \qquad \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau \rightsquigarrow \sigma \qquad \mathbf{closure}(\Gamma_C; \overline{Q}_p) = \overline{Q}_q \\ \mathbf{unambig}(\forall \overline{b}_k . \overline{Q}_q \Rightarrow TC\, \tau) \qquad (\Gamma_C; \bullet, \overline{b}_k \vdash_Q Q_q \rightsquigarrow \sigma_q \;, \; \forall q) \\ D \;\mathbf{fresh} \qquad \overline{\delta}_q \;\mathbf{fresh} \qquad \overline{\delta}'_y \;\mathbf{fresh} \\ (P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q \vDash [[\tau / a] Q'_i] \rightsquigarrow e_i \;, \; \forall i) \\ \Gamma' = \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}'_y : [\tau / a] \overline{Q}'_y \qquad P; \Gamma_C; \Gamma' \vdash_{tm} e \Leftarrow [\tau / a] \tau_1 \rightsquigarrow e \\ (D' : \forall \overline{b}'_m . \overline{Q}'_n \Rightarrow TC\, \tau_2).m' \mapsto \Gamma'' : e' \notin P \\ \mathit{where}\; [\overline{\tau}'_m / \overline{b}'_m] \tau_2 = [\overline{\tau}'_k / \overline{b}_k] \tau \qquad P' = (D : \forall \overline{b}_k . \overline{Q}_q \Rightarrow TC\, \tau).m \mapsto \Gamma' : e \end{array}}{P; \Gamma_C \vdash_{inst} \mathbf{instance} \;\; \overline{Q}_p \Rightarrow TC\, \tau \;\mathbf{where}\;\; \{m = e\} : P'}$$

Figure 6.6: $\lambda_{\mathbf{TC}}$ instance declaration typing

$$\begin{array}{lcll}
\sigma & ::= & \mathit{Bool} \mid a \mid \forall a.\sigma \mid \sigma_1 \to \sigma_2 \mid \{\, \overline{m_i : \sigma_i}^{\,i<n} \,\} & F_{\{\}}\ \textit{type} \\
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma.e \mid e_1\, e_2 \mid \Lambda a.e \mid e\, \sigma & F_{\{\}}\ \textit{term} \\
 & & \mid \{\, \overline{m_i = e_i}^{\,i<n} \,\} \mid e.m \mid \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e_2 & \\
\Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & F_{\{\}}\ \textit{context}
\end{array}$$

Figure 6.7: Target language syntax

6.4 Target Language $F_{\{\}}$

This section covers our target language $F_{\{\}}$, and the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$.

The target language is System F with records, which we consider a reasonable subcalculus of those used by Haskell compilers. Its syntax is shown in Figure 6.7. We omit its standard typing rules and call-by-name operational semantics and refer the reader to Pierce [75, Chapter 23], or Section 5 of the appendix.

6.4.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$

The red aspects in Figure 6.3 denote the elaboration of $\lambda_{\mathbf{TC}}$ terms to $F_{\{\}}$. We have adopted the convention that any red $F_{\{\}}$ types are the elaborated forms of their identically named blue $\lambda_{\mathbf{TC}}$ counterparts. This elaboration maps most $\lambda_{\mathbf{TC}}$ forms on identical $F_{\{\}}$ terms, with the exception of a few notable cases: (a) The interesting aspect of elaborating let expressions (sTm-infT-let) is that, as mentioned previously, superclasses are flattened into additional local constraints. The elaborated expression thus explicitly requires both the type variables and the full closure of the local constraints. (b) As opposed to $\lambda_{\mathbf{TC}}$, dictionary and type application are made explicit in $F_{\{\}}$. When elaborating variables x and method references m (sTm-checkT-meth), all previously substituted types $\overline{\tau}_j$ are now explicitly applied, together with the dictionary expressions $\overline{e}_i$. Furthermore, method names m are elaborated to $F_{\{\}}$ record labels m and therefore cannot appear by themselves, but must be applied to a record expression e, which originates from resolving the class constraint.

Type class resolution (Figure 6.5) of a $\lambda_{\mathbf{TC}}$ constraint Q results in a $F_{\{\}}$ expression e. When resolving the wanted constraint using a locally available constraint δ (sEntailT-local), this results in a regular term variable δ (which keeps the name of its $\lambda_{\mathbf{TC}}$ counterpart for readability). On the other hand, when resolving with the use of a global instance declaration D (sEntailT-inst), a record expression is constructed, containing the method name m and its corresponding implementation e. This method implementation now explicitly abstracts over the type variables $\overline{b}_k$ and term variables $\overline{\delta}_y$ originating from the method types's class constraints $\overline{Q}_y$, which annotate the class declaration. Furthermore, the record expression is nested in abstractions over the type variables $\overline{a}_j$ and term variables $\overline{\delta}'_i$ arising from the corresponding instance constraints $\overline{Q}'_i$. These abstractions are immediately instantiated by applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint Q to the instance declaration Q' and (b) the expressions $\overline{e}_i$ constructed by resolving the instance context constraints $\overline{Q}'_i$.

Example 1 $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$. Typing the Example 1 program results in the following environments:

$$\Gamma_C = (==) : Eq\, a : a \to a \to Bool$$

$$\begin{aligned} P = {} & (D_1 : Eq\, Int).(==) \mapsto \bullet : primEqInt \\ & , (D_2 : \forall a, b. Eq\, a \Rightarrow Eq\, b \Rightarrow Eq\, (a, b)).(==) \mapsto a, b, \delta_1 : Eq\, a, \delta_2 : Eq\, b : \\ & \quad \lambda(x_1, y_1).\lambda(x_2, y_2).\ (\&\&)\ ((==)\, x_1\, x_2)\ ((==)\, y_1\, y_2) \end{aligned}$$

The Eq class straightforwardly gets stored in the class environment Γ_C. Instances are stored in the $\lambda_{\mathbf{TC}}$ program context P (containing the dictionary constructor, the corresponding constraint, the method implementation and the environment under which to interpret this expression). Storing the instance declaration for $Eq\, Int$ is clear-cut. The instance for tuples on the other hand is somewhat more complex, since it requires an instance context, containing the local constraints $Eq\, a$ and $Eq\, b$. These constraints are made explicit, that is, the corresponding dictionaries are required by the elaborated implementation.

Elaborating the $\lambda_{\mathbf{TC}}$ expression results in the following $F_{\{\}}$ expression:

$$
\begin{aligned}
&\mathbf{let}\ \ \mathit{refl}\colon \forall a.\{(==) : a \to a \to \mathit{Bool}\} \to a \to \mathit{Bool}\\
&\qquad\qquad = \Lambda a.\lambda \delta_3 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda x : a.\delta_3.(==)\, x\, x\\
&\mathbf{in}\ \ \mathbf{let}\ \ \mathit{main}\colon \mathit{Bool}\\
&\qquad\qquad\qquad = \mathit{refl}\,(\mathit{Int}, \mathit{Int})\\
&\qquad\qquad\qquad\quad (\Lambda a.\Lambda b.\lambda \delta_4 : \{(==) : a \to a \to \mathit{Bool}\}.\\
&\qquad\qquad\qquad\qquad \lambda \delta_5 : \{(==) : b \to b \to \mathit{Bool}\}.\\
&\qquad\qquad\qquad\quad\ \ \{(==) = \lambda(x_1, y_1) : (a, b).\lambda(x_2, y_2) : (a, b).\\
&\qquad\qquad\qquad\qquad (\&\&)\,(\delta_4.(==)\, x_1\, x_2)\,(\delta_5.(==)\, y_1\, y_2)\})\\
&\qquad\qquad\qquad\quad \mathit{Int}\,\mathit{Int}\,\{(==) = \mathit{primEqInt}\}\,\{(==) = \mathit{primEqInt}\}\,(5, 42)\\
&\mathbf{in}\ \ \mathit{main}
\end{aligned}
$$

Note that the $\mathit{Eq}\, a$ constraint is made explicit in the implementation of refl, by abstracting over the constraint (elaborated to $F_{\{\}}$ as the record type $\{(==) : a \to a \to \mathit{Bool}\}$, which stores the method name and its corresponding type) with the use of the record variable δ_3. When this function is called in main, both the type and the dictionary variable are instantiated. The latter is performed by (recursively) constructing a dictionary expression, using the type class resolution mechanism, as explained above in Section 6.4.1.

Example 2 $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$. Below is the environment generated by typing the Example 2 program (including the Section 7.2.2 extension), which features superclasses.

$$
\begin{aligned}
\Gamma_C &= \mathit{base} : \mathit{Base}\, a : a \to \mathit{Bool}\\
&\quad , \mathit{sub}_1 : \mathit{Base}\, a \Rightarrow \mathit{Sub}_1\, a : a \to \mathit{Bool}\\
&\quad , \mathit{sub}_2 : \mathit{Base}\, a \Rightarrow \mathit{Sub}_2\, a : a \to \mathit{Bool}
\end{aligned}
$$

The class environment Γ_C contains three classes, two of which have superclasses. However, since the example does not contain any instance declarations, the resulting program context P is empty.

For space reasons, we focus solely on elaborating *test2*, which results in the following $F_{\{\}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \ \mathit{test}_2 \colon \forall a.\{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_1 : a \to \mathit{Bool}\} \\
&\qquad\qquad\quad \to \{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_2 : a \to \mathit{Bool}\} \to a \to \mathit{Bool} \\
&\qquad\qquad = \Lambda a.\lambda\delta_1 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda\delta_2 : \{\mathit{sub}_1 : a \to \mathit{Bool}\}. \\
&\qquad\qquad\quad \lambda\delta_3 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda\delta_4 : \{\mathit{sub}_2 : a \to \mathit{Bool}\}.\lambda x : a.\ \delta_1.\mathit{base}\, x \\
&\textbf{in}\ \ \mathit{True}
\end{aligned}
$$

Note that the $\lambda_{\mathbf{TC}}$ expression requires two local constraints: $\mathit{Sub}_1\, a$ and $\mathit{Sub}_2\, a$. However, after flattening the superclasses and adding them to the local constraints, the elaborated $F_{\{\}}$ expression requires (the elaborated form of) the $\mathit{Base}\, a$, $\mathit{Sub}_1\, a$, $\mathit{Base}\, a$ and $\mathit{Sub}_2\, a$ constraints. Notice the duplicate $\mathit{Base}\, a$ entry. Either of these two entries can be used for calling the method base. We have arbitrarily selected the first here. The next section proves that both options are equivalent and can be used interchangeably.

Chapter 7

Meta Theory: Coherence

"Seeing, contrary to popular wisdom, isn't believing. It's where belief stops, because it isn't needed any more."

Pyramids
Terry Pratchett

7.1 Introduction

Given the extensive attention that type classes have received, it may be surprising that the metatheory of their elaboration-based semantics [32] has not yet been exhaustively studied. In particular, as far as we know, while there have been many informal arguments, the formal notion of *coherence* has never been proven. Reynolds [80] has defined coherence as follows:

> *"When a programming language has a sufficiently rich type structure, there can be more than one proof of the same typing judgment; potentially this can lead to semantic ambiguity since the semantics of a typed language is a function of such proofs. When no such ambiguity arises, we say that the language is coherent."*

Type classes give rise to two main (potential) sources of incoherence. The first source are *ambiguous type schemes*, such as that of the function *foo*:

$$\mathit{foo} :: (\mathit{Show}\ a, \mathit{Read}\ a) \Rightarrow \mathit{String} \rightarrow \mathit{String}$$
$$\mathit{foo}\ s = \mathit{show}\ (\mathit{read}\ s)$$

The type scheme of *foo* requires that the type with which *a* will be instantiated must have *Show* and *Read* instances. This restriction alone is too permissive, because the type part ($\mathit{String} \rightarrow \mathit{String}$) of *foo*'s type scheme is not sufficient for a deterministic instantiation of *a* from the calling context. *a* can thus be instantiated arbitrarily to any type with *Show* and *Read* instances. Yet, the choice of type may lead to a different behavior of *show* and *read*, and thus of *foo* as a whole. For instance, *foo* `"1"` yields `"1"` when *a* is instantiated to *Int*, and `"1.0"` when it is instantiated to *Float*. To rule out this source of incoherence, Jones [40] requires type schemes to be unambiguous and has formally proven that, for his system, this guarantees coherence.

The second source of ambiguity arises from the type class resolution mechanism itself. Such mechanisms check whether a particular type class constraint holds. Usually, they are styled after resolution-based proof search in logic, where type class instances act as Horn clauses and type scheme constraints as additional facts. Generally, this process is nondeterministic, but languages like Haskell, Mercury or PureScript contain it by requiring that type class instances do not overlap with each other or with locally given constraints. Nevertheless, superclasses remain as a source of nondeterminism; indeed, a superclass constraint can be resolved through any of its subclass constraints. Hence, in the presence of superclasses, type class resolution should properly be considered as a potential source for incoherence. Moreover, overlap between locally wanted constraints and global instances is often allowed (e.g., through GHC's *FlexibleContexts* pragma), but a formal argument for its harmlessness is also lacking. Jones [40] considered neither of these aspects and simply assumed the coherence of resolution as a given. Morris [60] side-steps these issues with a denotational semantics that is disconnected from the original elaboration-based semantics and its implementations (e.g., Hugs and GHC).

This chapter aims to fill this gap in the metatheory of programming languages featuring type classes, including industrial grade languages such as Haskell, by formally establishing that elaboration-based type class resolution is coherent in the presence of superclasses and flexible contexts. The proof of this property is considerably complicated by the indirect, elaboration-based approach that is used to give meaning to programs with type classes. These dictionaries can, however, often be constructed in more than one way, resulting in multiple possible translations for a single program. The problem is that different translations of the same source program actually may have different meanings in the core language. The reason for this discrepancy is that the core language is more expressive than the source language and admits programs — that cannot be expressed in the source language — in which the different dictionaries can be distinguished.

We solve this problem with a new two-step approach that splits the problem into two subproblems. The midway point is an intermediate language that makes type class dictionaries explicit, but—inspired by fully abstract compilation—cannot distinguish between different elaborations from the same source language term [1]. We use a logical-relations approach to show that the nondeterministic elaboration from the source language to this intermediate language is coherent. Showing coherence for the elaboration from the intermediate language to the target language is much simpler, because we can formulate it in a deterministic fashion.

In summary, the contributions of this work are:

- We present a simple calculus $\lambda_{\mathbf{TC}}$ with full-blown type class resolution (incl. superclasses), which isolates nondeterministic resolution. Furthermore, we present an elaboration from $\lambda_{\mathbf{TC}}$ to the target language $F_{\{\}}$, System F with records, which are used to encode dictionaries.
- We present an intermediate language $F_{\mathbf{D}}$ with explicit dictionary-passing. This language enforces the uniqueness of dictionaries, which captures the intention of type class instances. We study its metatheory, and define a logical relation to prove contextual equivalence.
- We present elaborations from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ and from $F_{\mathbf{D}}$ to $F_{\{\}}$, and prove that a direct translation from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ can always be decomposed into an equivalent translation through $F_{\mathbf{D}}$.
- We prove coherence of the elaboration between $\lambda_{\mathbf{TC}}$ and $F_{\mathbf{D}}$, using logical relations.
- We prove that coherence is also preserved through the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$. As a consequence, by combining this with the previous result, we prove that the elaboration between $\lambda_{\mathbf{TC}}$ and $F_{\{\}}$ is coherent. The latter coherence result implies coherence of elaboration-based type class resolution in the presence of superclasses and flexible contexts.

The full formalization and coherence proof are provided in the accompanying 122-page appendix, which can be found at `https://arxiv.org/pdf/1907.00844.pdf`. We will refer to this document as the coherence proof appendix.

The purpose of our work is twofold: 1) To develop a proof technique to establish coherence of type class resolution. Because this result is achieved on a minimal calculus, this work becomes a basis for researchers investigating type class extensions and larger languages, as well as their impact on coherence. 2) To present a formal proof of coherence for language designers considering to adopt type classes. In doing so, we show that the informally trivial argument for the coherence of type class resolution is surprisingly hard to formalize.

7.2 Overview

This section provides some background on dictionary-passing elaboration of type class resolution and discusses the potential nondeterminism introduced by superclasses and local constraints. We then briefly introduce our calculi and discuss the key ideas of the coherence proof. Throughout the section we use Haskell-like syntax as the source language for examples, and to simplify our informal discussion we use the same syntax without type classes as the target language.

7.2.1 Dictionary-Passing Elaboration

A program is coherent if it has exactly one meaning — i.e., its semantics is unambiguously determined. For type classes this is not as straightforward as it seems, because their dynamic semantics are not expressed directly but rather by type-directed elaboration into a simpler language without type classes such as System F. Thus the dynamic semantics of type classes are given indirectly as the dynamic semantics of their elaborated forms.

7.2.2 Nondeterminism and Coherence

For Haskell'98 programs there is usually only one way to construct a dictionary for a type class constraint. Yet, in the presence of superclasses, there may be multiple ways. Suppose we extend Example 2 with an additional subclass and the following function:

```
class Base a ⇒ Sub2 a where
  sub2 :: a → Bool
test2 :: (Sub1 a, Sub2 a) ⇒ a → Bool
test2 x = base x
```

There are two possible ways to resolve the *Base a* constraint that arises from the call to *base* in function *test2*, resulting in the following two translations: we can either establish the desired constraint as the superclass of the given *Sub1 a* constraint or as the superclass of the given *Sub2 a* constraint.

$$\begin{array}{l}
\mathit{test2a}, \mathit{test2b} :: (\mathit{Sub1D}\ a, \mathit{Sub2D}\ a) \to a \to \mathit{Bool} \\
\mathit{test2a}\ (\mathit{d1}, \mathit{d2})\ x = \mathit{base}\ (\mathit{super1}\ \mathit{d1})\ x \\
\mathit{test2b}\ (\mathit{d1}, \mathit{d2})\ x = \mathit{base}\ (\mathit{super2}\ \mathit{d2})\ x
\end{array}$$

Fortunately, this nondeterminism is harmless because the difference between the two elaborations cannot be observed. Indeed, for any given type *A*, Haskell'98 only allows a single instance *Base A*, and it does not matter whether we access its dictionary directly or through one of its subclass instances. More generally, this suggests that type class resolution in Haskell'98 is coherent.

If we relax the Haskell'98 non-overlap condition for locally given constraints and adopt flexible contexts (allowing for arbitrary types in class constraints, rather than simple type variables), another source of nondeterminism arises. Consider:

$$\begin{array}{l}
\mathit{isZero} :: \mathit{Eq}\ \mathit{Int} \Rightarrow \mathit{Int} \to \mathit{Bool} \\
\mathit{isZero}\ n = n \mathbin{==} 0
\end{array}$$

There are two ways to resolve the wanted *Eq Int* constraint that arises from the use of (==). Either we use the global *Eq Int* constraint (in *isZero1*), or we use the locally given *Eq Int* constraint, passed as argument *d* (in *isZero2*):

$$\begin{array}{l}
\mathit{isZero1}, \mathit{isZero2} :: \mathit{EqD}\ \mathit{Int} \to \mathit{Int} \to \mathit{Bool} \\
\mathit{isZero1}\ d\ n = (==)\ \mathit{eqInt}\ n\ 0 \\
\mathit{isZero2}\ d\ n = (==)\ d\ n\ 0
\end{array}$$

Haskell'98 does not allow the *Eq Int* constraint in *isZero*'s signature, which overlaps with the global *Eq Int* instance; it only allows constraints on type variables in function signatures. This prevents the above nondeterminism in the elaboration. Yet, the nondeterminism is, once more, harmless; there is no way that the supplied dictionary *d* can be anything other than the global instance's dictionary *eqInt*. Informally, resolution remains coherent in the presence of flexible contexts.

7.2.3 Contextual Difference

While it is easy to provide an informal argument for the coherence of type class resolution, formally establishing the property is much harder. The indirect, elaboration-based attribution of a dynamic semantics in particular is a complicating factor, since it requires us to reason about two languages simultaneously. Unfortunately, there is another factor that further complicates the proof: different elaborations of the same source program can actually be distinguished in the target language. Consider, for instance, the target program below:

```
discern :: ((Sub1D (), Sub2D ()) → () → Bool) → Bool
discern f =
  let b1 = BaseD {base = λ() → True}
      b2 = BaseD {base = λ() → False}
      d1 = Sub1D {super1 = b1}
      d2 = Sub2D {super2 = b2}
  in f (d1, d2) ()
```

We find that *discern test2a* evaluates to *True* and *discern test2b* evaluates to *False*. Hence, since *discern* can differentiate between them, *test2a* and *test2b* clearly do not have the same meaning in the target language.

The dictionaries for *Sub1* () and *Sub2* () have different implementations for their *Base* () superclass. The source language would never allow this, but the target language has no notion of type classes and happily admits *discern*'s violation of source language rules.

The problem is that the target language is more expressive than the source language. While *test2a* and *test2b* cannot be distinguished in any program context that arises from the source language, we can write target programs like *discern* that are not the image of any source program and thus do not have to play by the source language rules.

7.2.4 Our Approach to Proving Coherence

To avoid the problem with contextual difference in the target language, we employ a novel two-step approach. We prove that any elaboration from a source language program into a dictionary-passing encoding in the target language, can be decomposed in two separate elaborations through an intermediate language. We thus obtain two simpler problems for proving coherence of type class resolution.

The source language, $\lambda_{\mathbf{TC}}$ (presented in blue), features full-fledged type class resolution, and simplifies term typing with a bidirectional type system (a technique popularized by Pierce and Turner [76]) to not distract from the main objective of coherent resolution.

The intermediate language, $F_{\mathbf{D}}$ (presented in green), is an extension of System F that explicitly passes type class dictionaries, and preserves the source language invariant that there is at most one such dictionary value for any combination of class and type. We show $F_{\mathbf{D}}$ is type-safe and strongly normalizing, and define a logical relation that captures the contextual equivalence of two $F_{\mathbf{D}}$ terms.

The target language, $F_{\{\}}$ (presented in red), is a different variant of System F without direct support for type class dictionaries; instead it features records, which can be used to encode dictionaries, but does not enforce uniqueness of instances.

The different calculi are presented in Figure 7.1, where the edges denote possible elaborations.

$$\begin{tikzcd}
\lambda_{\mathbf{TC}} \text{ (Fig. 6.2)} \arrow[rr, "\text{Thm. 1}"] \arrow[dr, "\text{Thm. 14}"'] & & F_{\{\}} \text{ (Fig. 6.7)} \\
& F_{\mathbf{D}} \text{ (Fig. 7.2)} \arrow[ur, "\text{Thm. 9}"'] &
\end{tikzcd}$$

Figure 7.1: The different calculi with elaborations

The coherence proof consists of two main parts:

Coherent Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$. Our elaboration from $\lambda_{\mathbf{TC}}$ into $F_{\mathbf{D}}$ is nondeterministic, but type preserving. Furthermore, we show that any two $F_{\mathbf{D}}$ elaborations of the same $\lambda_{\mathbf{TC}}$ term are logically related, and prove that this logical relation implies contextual equivalence. This establishes that the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ is coherent.

Deterministic Elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$. Because of the syntactic similarity between $F_{\mathbf{D}}$ and $F_{\{\}}$, the elaboration from the former into the latter is a more straightforward affair. In addition to being type preserving, it is also deterministic, and preserves contextual equivalence.

These results are easily combined to show the coherence of the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$, which implies coherence of elaboration-based type class resolution. The full proofs can be found in the coherence proof appendix. Note that the proofs depend on a number of standard boilerplate conjectures (e.g., substitution lemmas), which can be found in Sections J.1 and K.1 of the coherence proof appendix.

7.3 Coherence

This section provides an outline for our coherence proof, and defines the required notions. We first provide a definition of *contextual equivalence* [61], which captures that two expressions have the same meaning.

7.3.1 Contextual Equivalence

In order to formally discuss the concept of contextual equivalence, we first define the notion of an *expression context*.

Expression Contexts. An expression context M is an expression with a single hole, for which another expression e can be filled in, denoted as $M[e]$. The syntax can be found in Figure 6.2.

The typing judgment for an expression context M is of the form $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$. This means that for any expression e such that $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$, we have $P; \Gamma_C; \Gamma' \vdash_{tm} M[e] \Rightarrow \tau' \rightsquigarrow e'$. Following regular $\lambda_{\mathbf{TC}}$ term typing, context typing spans all combinations of type inference and checking mode: $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$, $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$ and $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$.

For example, the simplest expression context is the empty context $[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$.

Now we can formally define contextual equivalence. Note that the small step operational semantics can be found in Section E.4 of the coherence proof appendix. The environment and type well-formedness judgments can be found in Sections B.4 and B.1 of the appendix respectively.

Definition 2 (Kleene Equivalence)**.**
Two $F_{\{\}}$ *expressions* e_1 *and* e_2 *are Kleene equivalent, written* $e_1 \simeq e_2$, *if there exists a value* v *such that* $e_1 \longrightarrow^* v$, *and* $e_2 \longrightarrow^* v$.

Definition 3 (Contextual Equivalence)**.**
Two expressions $\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Gamma \vdash_{tm} e_2 : \sigma$,
where $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma$,
are contextually equivalent, written $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$,
if forall $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_2$
implies $M_1[e_1] \simeq M_2[e_2]$.

The definition is adapted from Harper [33, Chapter 46]. Intuitively, contextual equivalence means that two open expressions are observationally indistinguishable, when used in any program that instantiates the expressions' free variables.

7.3.2 Coherence

We can now make a first attempt to prove that different translations of the same source program are contextually equivalent. The program typing judgment can be found in Section B.2 of the coherence proof appendix.

> **Theorem 1** (Coherence).
> *If* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$
> *then* $\Gamma_{C1} = \Gamma_{C2}$, $P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

We first set out to prove the simpler variant, which only considers expressions [1].

> **Theorem 2** (Expression Coherence).
> *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
> *then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

The main requirement which makes type class resolution coherent is that type class instances do not overlap. However, since $F_{\{\}}$ uses records to encode dictionaries, the $F_{\{\}}$ language does not enforce this crucial uniqueness property. In order to prove Theorem 1, we introduce an additional intermediate language $F_{\mathbf{D}}$, which captures the invariant that type class instances do not overlap, and makes it explicit.

7.4 Intermediate Language $F_{\mathbf{D}}$

This section presents our intermediate language $F_{\mathbf{D}}$. The language is modeled with three main goals in mind: (a) $F_{\mathbf{D}}$ should explicitly pass type class dictionaries, which are implicit in $\lambda_{\mathbf{TC}}$; (b) the $F_{\mathbf{D}}$ type system should capture the uniqueness of dictionaries, thus enforcing the elaboration from $\lambda_{\mathbf{TC}}$ to preserve full abstraction; and (c) $F_{\mathbf{D}}$ expressions should elaborate

[1] Theorem 2 also has a type checking mode counterpart, which has been omitted here for space reasons.

$$
\begin{array}{lcll}
\sigma, \tau & ::= & \ldots \mid Q \Rightarrow \sigma & \textit{type} \\
Q & ::= & \mathit{TC}\,\sigma & \textit{class constraint} \\
C & ::= & \forall \overline{a}.\overline{Q} \Rightarrow Q & \textit{constraint} \\
 & & & \\
d & ::= & \delta \mid D\,\overline{\sigma}\,\overline{d} & \textit{dictionary} \\
dv & ::= & D\,\overline{\sigma}\,\overline{dv} & \textit{dictionary value} \\
e & ::= & \ldots \mid \lambda\delta : Q.e \mid e\,d \mid d.m & \textit{expression} \\
 & & & \\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \mathit{TC}\,a : \sigma & \textit{class environment} \\
\Sigma & ::= & \bullet \mid \Sigma, (D : C).m \mapsto e & \textit{method environment}
\end{array}
$$

Figure 7.2: $F_{\mathbf{D}}$, selected syntax

straightforwardly and deterministically to the target language $F_{\{\}}$ (System F with records, see Section 6.4).

To this end, $F_{\mathbf{D}}$ is an extension of System F, with built-in support for dictionaries. These dictionaries differ from those commonly used in Haskell compilers in that they are special constants rather than a record of method implementations. A separate global map Σ from dictionaries to method implementations gives access to the latter. Note that this setup does not allow programs to introduce new (and possibly overlapping) dictionaries dynamically. All dictionaries have to be provided upfront, where uniqueness is easily enforced.

Syntax. Figure 7.2 shows selected syntax of $F_{\mathbf{D}}$; the basic System F constructs are omitted and can be found in Section A.2 of the coherence proof appendix.

$F_{\mathbf{D}}$ introduces a new syntactic sort of dictionaries d that can either be a dictionary variable δ or a dictionary constructor D. A dictionary constructor has a number (possibly zero) of type and dictionary parameters and always appears in fully-applied form. Each constructor corresponds to a unique instance declaration, and is mapped to its method implementation by the global environment Σ.

Expressions have explicit application and abstraction forms for dictionaries. Furthermore, similarly to $F_{\{\}}$, method names can no longer be used on their own. Instead, they have to be applied explicitly to a dictionary, in the form $d.m$.

$F_{\mathbf{D}}$ types σ or τ are identical to the well-known System F types, with the addition of a special function type $Q \Rightarrow \sigma$ for dictionary abstractions.

Similarly to $\lambda_{\mathbf{TC}}$, $F_{\mathbf{D}}$ features two global and a single local environment Γ. The latter is similar to the $\lambda_{\mathbf{TC}}$ typing environment Γ. However, there are notable differences between the global environments. The $F_{\mathbf{D}}$ class environment Γ_C does not contain any superclass information. The reason for this is that, as previously mentioned in Section 6.3, superclass constraints in the source language $\lambda_{\mathbf{TC}}$ are flattened into local constraints, and stored in the typing environment Γ. The analog to the $\lambda_{\mathbf{TC}}$ program context P is the $F_{\mathbf{D}}$ method environment Σ, storing information about all dictionary constructors D. Each constructor corresponds to a unique instance declaration, and stores the accompanying method implementations.

Typing. Figure 7.3 (left-hand side) shows selected typing rules for $F_{\mathbf{D}}$ expressions. The red parts can be safely ignored for now, as they will be explained in detail in Section 7.4.2. The judgment $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ expresses that the $F_{\mathbf{D}}$ term e of type σ is well-typed under environments Σ, Γ_C and Γ. As shown by rule iTm-method, the type of a method variable applied to a dictionary is simply the corresponding method type (as stored in the static class environment), where the type variable has been substituted for the corresponding dictionary type.

Figure 7.5 shows the typing rules for dictionaries. The relation $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q$ denotes that dictionary d of dictionary type Q is well-formed under environments Σ, Γ_C and Γ. Similarly to regular term variables x (iTm-var), the type of a dictionary variable δ (D-var) is obtained from the typing environment Γ. The type of a dictionary constructor D (D-con), on the other hand, is obtained by finding the corresponding entry in the method environment Σ and substituting any types $\overline{\sigma}_j$ applied to it in the corresponding class constraint $TC\,\sigma_q$. All applied dictionaries $\overline{d}_i$ have to be well-typed with the corresponding constraint. Finally, the corresponding method implementation has to be well-typed in the reduced method environment Σ_1, which only contains the instances declared before D. As mentioned in Section 6.3, this reduced environment disallows recursive method implementations, as this would significantly clutter the coherence proof while, as a feature, recursion is completely orthogonal to the desired property.

Non-Overlapping Instances. The main requirement for achieving coherence of type class resolution, is that type class instances do not overlap. This requirement is common in Haskell and is for example enforced in GHC (though strongly discouraged, the *OverlappingInstances* pragma disables it). By storing all method implementations (with their corresponding instances) in a single environment Σ, this invariant can easily be made explicit.

$$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e} \qquad \textit{($F_{\mathbf{D}}$ Term Typing)}$$

iTm-method

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma \rightsquigarrow e \\ (m : TC\,a : \sigma') \in \Gamma_C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma' \rightsquigarrow e.m}$$

iTm-constrI

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : Q \vdash_{tm} e : \sigma \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda\delta : Q.e : Q \Rightarrow \sigma \rightsquigarrow \lambda\delta : \sigma.e}$$

iTm-constrE

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : Q \Rightarrow \sigma \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1\, e_2}$$

$$\boxed{\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma} \qquad \textit{(Constr. Well-Formedness)}$$

iQ-TC

$$\frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ \Gamma_C = \Gamma_{C1}, m : TC\,a : \sigma', \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma' \rightsquigarrow \sigma'\end{array}}{\Gamma_C; \Gamma \vdash_Q TC\,\sigma \rightsquigarrow [\sigma/a]\{m : \sigma'\}}$$

$$\boxed{\Sigma \vdash e \longrightarrow e'} \qquad \textit{($F_{\mathbf{D}}$ Evaluation)}$$

iEval-DApp

$$\frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, d \longrightarrow e'\, d}$$

iEval-method

$$\frac{d \longrightarrow d'}{\Sigma \vdash d.m \longrightarrow d'.m}$$

iEval-DAppAbs

$$\frac{}{\Sigma \vdash (\lambda\delta : Q.e)\, d \longrightarrow [d/\delta]e}$$

Figure 7.3: $F_{\mathbf{D}}$ typing and operational semantics, selected rules

$\boxed{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}$ *($F_{\mathbf{D}}$ Environment Well-Formedness)*

iCtx-MEnv

$$\frac{\begin{array}{c}\mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma) \qquad \Gamma_C; \bullet \vdash_C \forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma \rightsquigarrow \sigma \\ (m : TC\,a : \sigma') \in \Gamma_C \qquad \Sigma; \Gamma_C; \bullet \vdash_{tm} e : \forall \overline{a}_j . \overline{Q}_i \Rightarrow [\sigma/a]\sigma' \rightsquigarrow e \\ D \notin \mathbf{dom}(\Sigma) \qquad (D' : \forall \overline{a}'_m . \overline{Q}'_n \Rightarrow TC\,\sigma').m' \mapsto e' \notin \Sigma \\ \mathbf{where}[\overline{\sigma}_j/\overline{a}_j]\sigma = [\overline{\sigma}'_m/\overline{a}'_m]\sigma' \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\vdash_{ctx} \Sigma, (D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma).m \mapsto e; \Gamma_C; \Gamma}$$

Figure 7.4: $F_{\mathbf{D}}$ environment well-formedness, selected rules

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e}$ *(Dictionary Typing)*

D-var

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \rightsquigarrow \delta}$$

D-con

$$\frac{\begin{array}{c}(D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{Q}_i . e \in \Sigma \\ (\Gamma_C; \bullet, \overline{a}_j \vdash_Q Q_i \rightsquigarrow \sigma'_i\ ,\ \forall i) \qquad (\Gamma_C; \Gamma \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j\ ,\ \forall j) \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \rightsquigarrow e \\ (\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_j/\overline{a}_j]Q_i \rightsquigarrow e_i\ ,\ \forall i) \\ \Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{Q}_i . e, \Sigma_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D\,\overline{\sigma}_j\,\overline{d}_i : TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q \rightsquigarrow (\Lambda \overline{a}_j . \lambda\,\overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\})\,\overline{\sigma}_j\,\overline{e}_i}$$

Figure 7.5: $F_{\mathbf{D}}$ dictionary typing

Figure 7.4 shows the environment well-formedness condition $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$ for the method environment. Besides stating well-scopedness, it denotes that the method environment Σ cannot contain a second instance, for which the head of the constraint overlaps with $TC\,\sigma$, up to renaming. This key property will be exploited in our coherence proof.

Operational Semantics. As $F_{\mathbf{D}}$ is an extension of System F, its call-by-name operational semantics are mostly standard. The non-standard rules can be found in Figure 7.3 (bottom right), where $\Sigma \vdash e \longrightarrow e'$ denotes expression e evaluating to e' in a single step, under method environment Σ.

The evaluation rules for dictionary application (iEval-DApp and iEval-DAppAbs) are identical to those for term and type application. More interesting, however, is the evaluation for methods (iEval-method). A method name applied to a dictionary evaluates in one step to the method implementation, as stored in the environment Σ.

Metatheory. $F_{\mathbf{D}}$ is type safe. That is, the common progress and preservation properties hold:

> **Theorem 3** (Progress).
> *If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$, then either e is a value, or there exists e' such that $\Sigma \vdash e \longrightarrow e'$.*

> **Theorem 4** (Preservation).
> *If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$, and $\Sigma \vdash e \longrightarrow e'$, then $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e' : \sigma$.*

Analogously to $\lambda_{\mathbf{TC}}$, $F_{\mathbf{D}}$ rejects recursive expressions (including mutual recursion and recursive methods). This allows for a normalizing language, that is, any well-typed expression evaluates to a value, after a finite number of steps. The reason for working with a normalizing language is explained in Section C.1. Note that since the small step operational semantics are deterministic, normalization implies strong normalization.

> **Theorem 5** (Strong Normalization).
> *If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$ then all possible evaluation derivations for e terminate : $\exists v : \Sigma \vdash e \longrightarrow^* v$.*

$$\boxed{P;\Gamma_C;\Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e} \qquad \textit{(Source Term Checking)}$$

sTm-check-meth

$$\frac{\begin{array}{c}(m : \overline{Q}'_k \Rightarrow TC\,a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \qquad \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \\ P;\Gamma_C;\Gamma \vDash^M [TC\,\tau] \rightsquigarrow d \qquad \Gamma_C;\Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma \\ (P;\Gamma_C;\Gamma \vDash^M [[\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i] \rightsquigarrow d_i \ , \ \forall i) \\ (\Gamma_C;\Gamma \vdash^M_{ty} \tau_j \rightsquigarrow \sigma_j \ , \ \forall j) \qquad \vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma\end{array}}{P;\Gamma_C;\Gamma \vdash^M_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\,\overline{\tau}_j\,\overline{d}_i}$$

Figure 7.6: $\lambda_{\mathbf{TC}}$ typing with elaboration to $F_{\mathbf{D}}$, selected rules

The proof follows the familiar structure for proving normalization using logical relations, as presented by Ahmed [4], and can be found in Section K.4 of the coherence proof appendix.

7.4.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$

The green aspects in Figure 7.6 denote the elaboration of $\lambda_{\mathbf{TC}}$ terms to $F_{\mathbf{D}}$. Similarly to the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$, we have adopted the convention that any green $F_{\mathbf{D}}$ types or constraints are the elaborated forms of their identically named blue $\lambda_{\mathbf{TC}}$ counterparts. This elaboration works analogously to the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$, as shown in Figure 6.3. The full set of rules can be found in Section C.2 of the coherence proof appendix.

The only notable case is sTm-check-meth, where the entailment relation for solving the type class constraint $TC\,\tau$ now results in a dictionary d. As explained at the start of Section 7.4, unlike $F_{\{\}}$, $F_{\mathbf{D}}$ differentiates syntactically between dictionaries and normal expressions.

Type class resolution (Figure 7.7) of a $\lambda_{\mathbf{TC}}$ constraint Q results in a $F_{\mathbf{D}}$ dictionary d. When using a locally available constraint to resolve the wanted constraint (sEntail-local), the corresponding dictionary variable δ is returned. On the other hand, when resolving using a global instance declaration (sEntail-inst), a dictionary is constructed by taking the corresponding constructor D and applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint to the instance declaration and (b) the dictionaries $\overline{d}_i$, constructed by resolving the instance context constraints.

Metatheory. We discuss the coherence of the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ in detail in Section 7.5, and mention here that it is type preserving:

$$\boxed{P; \Gamma_C; \Gamma \vDash^{M} [Q] \rightsquigarrow d}$$ *(Constraint Entailment)*

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vDash^{M} [Q] \rightsquigarrow \delta} \textsc{sEntail-local}$$

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i, \overline{b}_k, \overline{\delta}_y : \overline{Q}_y \qquad Q = [\overline{\tau}_j / \overline{a}_j] Q' \\ (\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma_j \ , \ \forall j) \qquad \vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \\ (P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j / \overline{a}_j] Q_i] \rightsquigarrow d_i \ , \ \forall i) \end{array}}{P; \Gamma_C; \Gamma \vDash^{M} [Q] \rightsquigarrow D \, \overline{\sigma}_j \, \overline{d}_i} \textsc{sEntail-inst}$$

Figure 7.7: $\lambda_{\mathbf{TC}}$ constraint entailment with elaboration to $F_{\mathbf{D}}$

> **Theorem 6** (Typing Preservation - Expressions)**.**
> *If* $P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e$*, and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*, and* $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$*, then* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$*.*

The same theorem holds for check mode, but is omitted for space reasons. The full proofs can be found in Section J.3 of the coherence proof appendix.

Example 1 $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$. Elaborating the $\lambda_{\mathbf{TC}}$ environments that originate from Example 1, results in the following $F_{\mathbf{D}}$ environments:

$$\Gamma_C = (==) : Eq \, a : a \rightarrow a \rightarrow Bool$$

$$\begin{aligned} \Sigma = {} & (D_1 : Eq \, Int).(==) \mapsto primEqInt \\ & , (D_2 : \forall a, b. Eq \, a \Rightarrow Eq \, b \Rightarrow Eq \, (a, b)).(==) \mapsto \\ & \quad \Lambda a. \Lambda b. \lambda \delta_1 : Eq \, a. \lambda \delta_2 : Eq \, b. \\ & \quad \lambda (x_1, y_1) : (a, b). \lambda (x_2, y_2) : (a, b). (\&\&) \, (\delta_1.(==) \, x_1 \, x_2) \, (\delta_2.(==) \, y_1 \, y_2) \end{aligned}$$

Note that both the class environment Γ_C and the program context Σ are largely direct translations of their $\lambda_{\mathbf{TC}}$ counterparts. One notable difference is the fact that the environment Γ under which to interpret the $\lambda_{\mathbf{TC}}$ method implementation is now explicitly abstracted over in the $F_{\mathbf{D}}$ method implementation. Consider for instance the case of D_2, where the variables a, b, δ_1 and δ_2, which in $\lambda_{\mathbf{TC}}$ are implicitly provided by the typing environment, are now explicit in the term level.

Elaborating the $\lambda_{\mathbf{TC}}$ expression results in the following $F_{\mathbf{D}}$ expression:

$$
\begin{aligned}
&\mathbf{let}\ \ \mathit{refl} : \forall a.\mathit{Eq}\, a \Rightarrow a \rightarrow \mathit{Bool}\\
&\qquad\qquad = \Lambda a.\lambda\delta_3 : \mathit{Eq}\, a.\lambda x : a.\delta_3.(==)\, x\, x\\
&\mathbf{in}\ \ \mathbf{let}\ \ \mathit{main} : \mathit{Bool}\\
&\qquad\qquad = \mathit{refl}\, (\mathit{Int}, \mathit{Int})\, (D_2\, \mathit{Int}\, \mathit{Int}\, D_1\, D_1)\, (5, 42)\\
&\mathbf{in}\ \ \mathit{main}
\end{aligned}
$$

Unlike the corresponding $F_{\{\}}$ expression, shown in Section 6.4.1, records storing the method types and implementations do not need to be passed around explicitly. In $F_{\mathbf{D}}$, they are replaced by class constraints and dictionaries, respectively. The construction of these dictionaries through type class resolution is shown in Figure 7.7.

Example 2 $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$. Elaborating Example 2, including the extension from Section 7.2.2, results in the following $F_{\mathbf{D}}$ class environment (since no instance declarations exist, the method environment Σ remains empty):

$$
\begin{aligned}
\Gamma_C = {} & \mathit{base} : \mathit{Base}\, a : a \rightarrow \mathit{Bool}\\
&, \mathit{sub}_1 : \mathit{Sub}_1\, a : a \rightarrow \mathit{Bool}\\
&, \mathit{sub}_2 : \mathit{Sub}_2\, a : a \rightarrow \mathit{Bool}
\end{aligned}
$$

The $F_{\mathbf{D}}$ class environment no longer needs to store superclasses, as these are all flattened into additional local constraints during elaboration.

Similarly to Section 6.4.1, we focus solely on elaborating *test2*, which results in the following $F_{\mathbf{D}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \ \mathit{test}_2 \colon \forall a. \mathit{Base}\, a \Rightarrow \mathit{Sub}_1\, a \Rightarrow \mathit{Base}\, a \Rightarrow \mathit{Sub}_2\, a \Rightarrow a \rightarrow \mathit{Bool} \\
&\qquad\qquad = \Lambda a. \lambda \delta_1 : \mathit{Base}\, a. \lambda \delta_2 : \mathit{Sub}_1\, a. \lambda \delta_3 : \mathit{Base}\, a. \lambda \delta_4 : \mathit{Sub}_2\, a. \\
&\qquad\qquad\quad \lambda x : a. \delta_1.\mathit{base}\, x \\
&\textbf{in}\ \ \mathit{True}
\end{aligned}
$$

The only difference with the $F_{\{\}}$ elaboration is that we now use class constraints instead of passing around a record type (storing the method types).

7.4.2 Elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$

As both $F_{\mathbf{D}}$ and $F_{\{\}}$ are extensions of System F, the elaboration from former to latter is mostly trivial, leaving common features unchanged. The mapping of $F_{\mathbf{D}}$ dictionaries into $F_{\{\}}$ records, however, is non-trivial. Briefly, dictionary types are elaborated into record types, as shown in Figure 7.3 (top right), and dictionaries into record expressions, possibly nested within type and term abstractions and applications, as shown in Figure 7.5.

In particular, a dictionary type, TC, which corresponds to a unique entry $(m : TC\, a : \sigma')$ in the class environment Γ_C, elaborates to a record type whose field has the same name as the dictionary type's method name, m, and the type of that field is determined by the elaboration of σ'. A $TC\, \sigma$ dictionary elaborates to a record expression which is surrounded, firstly, by abstractions over type and term variables that arise from the method type's class constraints and, secondly, by type and term applications that properly instantiate those abstractions.

Metatheory. The following theorems confirm that the $F_{\mathbf{D}}$-to-$F_{\{\}}$ elaboration is indeed appropriate.

The first theorem states that a well-typed $F_{\mathbf{D}}$ expression always elaborates to a $F_{\{\}}$ expression that is also well-typed in the translated context.

> **Theorem 7** (Type Preservation).
> *If* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$,
> *then there are unique* Γ *and* σ *such that* $\Gamma \vdash_{tm} e : \sigma$,
> *where* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Secondly, and more importantly, the dynamic semantics is also preserved by the elaboration.

> **Theorem 8** (Semantic Preservation).
> *If* $\Sigma;\Gamma_C;\bullet \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Sigma \vdash e \longrightarrow^* v$
> *then there exists a* v *such that* $\Sigma;\Gamma_C;\bullet \vdash_{tm} v : \sigma \rightsquigarrow v$ *and* $e \simeq v$*.*

Thirdly, the elaboration is entirely deterministic.

> **Theorem 9** (Determinism).
> *If* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_1$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_2$*, then* $e_1 = e_2$*.*

7.4.3 Elaboration Decomposition

An elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ can always be decomposed into two elaborations through $F_{\mathbf{D}}$. This intuition is formalized in Theorems 10 and 11 respectively.

> **Theorem 10** (Elaboration Equivalence - Expressions).
> *If* $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma$
> *then* $P;\Gamma_C;\Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$
> *where* $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C;\Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$*.*

> **Theorem 11** (Elaboration Equivalence - Dictionaries).
> *If* $P;\Gamma_C;\Gamma \vDash [Q] \rightsquigarrow e$ *and* $\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma$
> *then* $P;\Gamma_C;\Gamma \vDash^M [Q] \rightsquigarrow d$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_d d : Q \rightsquigarrow e$
> *where* $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$*.*

Theorem 10 also has a type checking mode counterpart, which has been omitted for space reasons. The full proofs can be found in Section L of the coherence proof appendix.

7.5 Coherence Revisited

As mentioned previously in Section 7.3, the invariant that type class instances do not overlap is crucial in proving Theorem 1. This uniqueness property is made explicit in $F_{\mathbf{D}}$. Our proof thus proceeds by elaborating the $\lambda_{\mathbf{TC}}$ expression to two possibly different $F_{\mathbf{D}}$ expressions and subsequently elaborating these $F_{\mathbf{D}}$

expressions to $F_{\{\}}$ expressions. Consequently, the proof is split in two main steps. The first part is the most involved, where we use a technique based on logical relations to prove that any two $F_{\mathbf{D}}$ expressions originating from the same $\lambda_{\mathbf{TC}}$ expression are contextually equivalent. The second part proves that the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$ is contextual equivalence preserving. This step follows straightforwardly from the fact that the $F_{\mathbf{D}}$-to-$F_{\{\}}$ elaboration is deterministic. Together, these prove that the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ through $F_{\mathbf{D}}$ is coherent. Theorem 2 follows from this result, together with Theorem 10.

The remainder of this section explains the techniques we used to prove Theorem 1 in detail.

7.5.1 Coherent Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$

Logical Relations.

Logical relations [77, 89, 92] are key to proving contextual equivalence. In our type system, the logical relation for expressions is mostly standard, though the relation for dictionaries is novel.

Dictionaries. The logical relation over two open dictionaries is defined by means of an auxiliary relation on closed dictionaries. We define this value relation for closed dictionaries as follows. Note that from now on, we will omit elaborations when they are entirely irrelevant. The appendix uses the same convention.

Definition 4 (Value Relation for Dictionaries)**.**
The dictionary values $D\,\overline{\sigma}_j\,\overline{dv_{1\,i}}$ and $D\,\overline{\sigma}_j\,\overline{dv_{2\,i}}$ are in the value relation, defined as:

$$(\Sigma_1 : D\,\overline{\sigma}_j\,\overline{dv_{1\,i}}, \Sigma_2 : D\,\overline{\sigma}_j\,\overline{dv_{2\,i}}) \in \mathcal{V}[\![Q]\!]^{\Gamma_C} \triangleq$$

$$((\Sigma_1 : dv_{1\,i}, \Sigma_2 : dv_{2\,i}) \in \mathcal{V}[\![[\overline{\sigma}_j/\overline{a}_j]Q_i]\!]^{\Gamma_C}\ ,\ \forall i)$$

$$\wedge\ \ \Sigma_1;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv_{1\,i}} : R(Q)\ \ \wedge\ \ \Sigma_2;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv_{2\,i}} : R(Q),$$

$$\textit{where}\ (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q').m \mapsto e_1 \in \Sigma_1 \wedge Q = [\overline{\sigma}_j/\overline{a}_j]Q'$$

The value relation is indexed by the dictionary type Q. We require both dictionaries to be well-typed, and their dictionary arguments to be in the value relation as well. The relation has four additional parameters: the contexts Σ_1 and Σ_2, which annotate the dictionaries, the class environment Γ_C, used in the well-typing condition, and the type substitution R.

In order to define logical equivalence between open dictionaries, we substitute all free variables with closed terms, thus reducing them to closed dictionary values. Three kinds of variables exist (term variables x, type variables a and dictionary variables δ). This results in three separate semantic interpretations of the typing context Γ. The type substitution $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ maps all type variables $a \in \Gamma$ onto closed types. $\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ maps each term variable $x \in \Gamma$ to two expressions e_1 and e_2 that are in the expression value relation (see Definition 6), and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ maps each dictionary variable $\delta \in \Gamma$ to two logically related dictionary values. We use ϕ_1 and ϕ_2 to denote the substitution for the first and second expression, respectively.

Definition 5 (Logical Equivalence for Dictionaries)**.**
Two dictionaries d_1 and d_2 are logically equivalent, defined as:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : Q \triangleq$$

$$\forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C},\ \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C},\ \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} :$$

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$$

Two dictionaries d_1 and d_2 are logically equivalent if any substitution of their free variables (with related expressions / dictionaries) results in related dictionary values.

Expressions. The value relation for expressions is mostly standard, with two notable deviations. Firstly, the relation is defined over two different method environments Σ_1 and Σ_2. Hence, both expressions are annotated with their respective environment. Secondly, the dictionary abstraction case is novel.

Definition 6 (Value Relation for Expressions)**.**
Two values v_1 *and* v_2 *are in the value relation, defined as:*

$$(\Sigma_1 : \mathit{True}, \Sigma_2 : \mathit{True}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \mathit{False}, \Sigma_2 : \mathit{False}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![a]\!]_R^{\Gamma_C} \triangleq$$

$$(a \mapsto (\sigma, r)) \in R \ \wedge \ (v_1, v_2) \in r$$

$$\wedge \ \Sigma_1; \Gamma_C; \bullet \vdash_{tm} v_1 : \sigma \ \wedge \ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} v_2 : \sigma$$

$$(\Sigma_1 : \lambda x : \sigma_1.e_1, \Sigma_2 : \lambda x : \sigma_1.e_2) \in \mathcal{V}[\![\sigma_1 \rightarrow \sigma_2]\!]_R^{\Gamma_C} \triangleq$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_1 : R(\sigma_1 \rightarrow \sigma_2)$$

$$\wedge \ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_2 : R(\sigma_1 \rightarrow \sigma_2)$$

$$\wedge \forall (\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} :$$

$$(\Sigma_1 : (\lambda x : \sigma.e_1)\, e_3, \Sigma_2 : (\lambda x : \sigma.e_2)\, e_4) \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \lambda \delta : Q.e_1, \Sigma_2 : \lambda \delta : Q.e_2) \in \mathcal{V}[\![Q \Rightarrow \sigma]\!]_R^{\Gamma_C} \triangleq$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : Q.e_1 : R(Q \Rightarrow \sigma) \ \wedge \ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : Q.e_2 : R(Q \Rightarrow \sigma)$$

$$\wedge \forall (\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C} :$$

$$(\Sigma_1 : (\lambda \delta : Q.e_1)\, dv_1, \Sigma_2 : (\lambda \delta : Q.e_2)\, dv_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \Lambda a.e_1, \Sigma_2 : \Lambda a.e_2) \in \mathcal{V}[\![\forall a.\sigma]\!]_R^{\Gamma_C} \triangleq$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_1 : R(\forall a.\sigma) \ \wedge \ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_2 : R(\forall a.\sigma)$$

$$\wedge \forall \sigma', \forall r \in \mathit{Rel}[\sigma'] : \ \Gamma_C; \bullet \vdash_{ty} \sigma' \Rightarrow$$

$$(\Sigma_1 : (\Lambda a.e_1)\, \sigma', \Sigma_2 : (\Lambda a.e_2)\, \sigma') \in \mathcal{E}[\![\sigma]\!]_{R, a \mapsto (\sigma', r)}^{\Gamma_C}$$

Consider the interesting case of dictionary abstraction. The relation requires the terms to be well-typed, and the applications for all related input dictionaries to be in the expression relation $\mathcal{E}$. The definition of this $\mathcal{E}$ relation is as follows:

Definition 7 (Expression Relation).
Two expressions e_1 *and* e_2 *are in the expression relation, defined as:*

$$(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_1 : R(\sigma) \;\wedge \Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_2 : R(\sigma)$$

$$\wedge\, \exists v_1, v_2, \Sigma_1 \vdash e_1 \longrightarrow^* v_1, \Sigma_2 \vdash e_2 \longrightarrow^* v_2, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C}$$

In this definition, expressions are reduced to values and those values must be in the value relation. This is well-defined because $F_{\mathbf{D}}$ is strongly normalizing (Theorem 5).

Section 7.6 discusses how the logical relations can be adapted to support non-terminating expressions.

Finally, we can give the definition of logical equivalence for open expressions:

Definition 8 (Logical Equivalence for Expressions).
Two expressions e_1 *and* e_2 *are logically equivalent, defined as:*

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \triangleq$$

$$\forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C},\; \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C},\; \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C} :$$

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

We also provide a definition of logical equivalence for contexts:

Definition 9 (Logical Equivalence for Contexts).
Two contexts M_1 *and* M_2 *are logically equivalent, defined as:*

$$\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma') \triangleq$$

$$\forall e_1, e_2 : \Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \;\Rightarrow$$

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma'$$

Proof of $\lambda_{\mathbf{TC}}$-to-$F_{\mathbf{D}}$ Coherence.

With the above definitions we are ready to formally state the metatheory and establish the coherence theorems from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$.

Design Principle of $F_{\mathbf{D}}$. We emphasize that $F_{\mathbf{D}}$ captures the intention of type class instances. Theorem 12 states that any two dictionary values for the same constraint are logically related:

Theorem 12 (Value Relation for Dictionary Values)**.**
If $\Sigma_1; \Gamma_C; \bullet \vdash_d dv_1 : Q$ *and* $\Sigma_2; \Gamma_C; \bullet \vdash_d dv_2 : Q$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$ *then* $(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$*.*

Note that two environments Σ_1 and Σ_2 are logically equivalent under Γ_C, written $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$, when they contain the same dictionary constructors and the corresponding method implementations are logically equivalent. The full definition can be found in Section G.3 of the coherence proof appendix.

Coherent Resolution. We now prove that constraint resolution is semantically coherent, that is, if multiple resolutions of the same constraint exist, they are logically equivalent.

Theorem 13 (Logical Coherence of Dictionary Resolution)**.**
If $P; \Gamma_C; \Gamma \models^M [Q] \rightsquigarrow d_1$ *and* $P; \Gamma_C; \Gamma \models^M [Q] \rightsquigarrow d_2$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$
then $\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : Q$ *where* $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$*.*

Coherent Elaboration. Furthermore, in order to prove that the elaboration from $\lambda_{\mathbf{TC}}$to $F_{\mathbf{D}}$is coherent, we show that all elaborations of the same expression are logically equivalent [2].

Theorem 14 (Logical Coherence of Expression Elaboration)**.**
If $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$
then $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *where* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$*.*

Contextual Equivalence. We prove that all logically equivalent expressions are contextually equivalent. Together with Theorem 14, this shows coherence of the $\lambda_{\mathbf{TC}}$-to-$F_{\mathbf{D}}$ elaboration.

[2] Theorem 14 also has a type checking mode counterpart, which has been omitted here for space reasons.

We first provide a formal definition of contextual equivalence for $F_{\mathbf{D}}$ expressions. Kleene equivalence for $F_{\mathbf{D}}$ is defined similarly to Definition 2 and can be found in Section I.1 of the coherence proof appendix.

Definition 10 (Contextual Equivalence for $F_{\mathbf{D}}$ Expressions).
Two expressions $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma$,
are contextually equivalent, written $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$,
if forall $M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow Bool)$
and forall $M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow Bool)$
where $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow Bool)$,
we have that $\Sigma_1 : M_1[e_1] \simeq \Sigma_2 : M_2[e_2]$.

Theorem 15 (Logical Equivalence implies Contextual Equivalence).
If $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

7.5.2 Deterministic Elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$

Contextual Equivalence.

Similarly to expressions, the elaboration from a $\lambda_{\mathbf{TC}}$ context M to a $F_{\{\}}$ context M can always be decomposed into two elaborations, through a $F_{\mathbf{D}}$ context M. The syntax and typing judgments can be found in Sections A and F of the coherence proof appendix, respectively.

We now formally define contextual equivalence for $F_{\{\}}$ expressions through $F_{\mathbf{D}}$ contexts.

Definition 11 (Contextual Equivalence in $F_{\mathbf{D}}$ Context).
Two expressions $\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Gamma \vdash_{tm} e_2 : \sigma$,
where $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma$,
are contextually equivalent, written $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$,
if forall $M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_1$
and forall $M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_2$
where $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow Bool)$,
we have that $M_1[e_1] \simeq M_2[e_2]$,
where $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$
and $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$.

Proof of $F_{\mathbf{D}}$-to-$F_{\{\}}$ Coherence.

We continue by proving that contextual equivalence is preserved by the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$:

> **Theorem 16** (Elaboration preserves Contextual Equivalence).
> *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
> *and* $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma \rightsquigarrow e_1$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2$
> *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

Proof of $\lambda_{\mathbf{TC}}$-to-$F_{\{\}}$ Coherence.

Finally, in order to link back to Theorem 2 (which has no notion of $F_{\mathbf{D}}$), we prove that contextual equivalence with $F_{\mathbf{D}}$ contexts implies contextual equivalence with $\lambda_{\mathbf{TC}}$ contexts:

> **Theorem 17** (Contextual Equivalence in $F_{\mathbf{D}}$ implies Contextual Equivalence in $\lambda_{\mathbf{TC}}$).
> *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
> *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$
> *and* $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$ *then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

For clarity, we restate coherence Theorems 2 and 1:

Theorem 2 (Expression Coherence - Restated).
If $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
then $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Theorem 2 follows by combining Theorems 10, 14, 15, 16 and 17.

Theorem 1 (Coherence - Restated).
If $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$
then $\Gamma_{C1} = \Gamma_{C2}$, $P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Finally, we show that Theorem 1 follows from Theorem 2. The full proofs can be found in Section M of the coherence proof appendix.

7.6 Discussion of Possible Extensions

As the goal of this work was to find a proof technique to formally establish coherence for type class resolution, a stripped down source calculus was employed in order not to clutter the proof. This section provides a brief discussion of extending our coherence proof to support several mainstream language features.

Ambiguous Type Schemes. As mentioned previously, our work is orthogonal to ambiguous type schemes, which have already been extensively studied by Jones [40]. We believe our work and the proof by Jones can be combined, which would then relax the restriction of bidirectional type checking, and prove coherence for both ambiguous type schemes and type class resolution.

General Recursion. Recursion is an important feature, present in any real world programming language. It is important to note that, while $\lambda_{\mathbf{TC}}$ does not feature recursion on the expression level (as it does not affect the essence of the coherence proof), type class resolution itself is recursive. Dictionary values are constructed dynamically from a statically given set of dictionary constructors (one constructor per type class instance). The system can thus recursively generate an arbitrary number of dictionaries from a finite set of instances.

Our logical relations can be adapted to support general recursion, through well-known techniques, such as step indexing [3]. While this results in a significantly longer and more cluttered proof, we do not anticipate any major complications.

Multi-Parameter Type Classes. Just like regular type class instances, multi-parameter instances (as supported by GHC) are subject to the no-overlap rule. Hence, they respect our main assumption. They may indeed give rise to more ambiguity, but this is the kind of ambiguity that is studied by Jones [40], not the kind that shows up during resolution. Note that functional dependencies were originally introduced by Jones [46] as a way to resolve the ambiguity caused by multi-parameter instances.

Dependent Types. Dependently typed languages, e.g., Agda [19] and Idris [12], include language features that are inspired by type classes. Proving resolution coherence in a dependently typed setting requires significant extension of our calculi, as dependent types collapse the term and type levels into a single level and thus enable more powerful type signatures for classes and instances. Furthermore, our logical relation needs to be extended to support dependent types [6] as well. Fortunately, the essence of our proof strategy still applies. That is, the intermediate language incorporates separate binding structures for dictionaries, and enforces the uniqueness of dictionaries. We thus believe a non-trivial extension of our proof methodology can be used to prove coherence for type class resolution in the setting of dependently typed languages.

Non-overlapping Instances. Our work is built on top of the assumption that type class instances do not overlap. This is enforced during the type checking of instance declarations, and made explicit in the intermediate language. Whether a constraint is entailed directly from an instance, through user provided constraints in a type annotation, or through local evidence, is not actually relevant, as all evidence ultimately has to originate from a non-overlapping instance declaration.

Therefore, our work can be extended to include features where the assumption holds true. This includes, among others, GADT's [72], implication constraints [11], type constructors, higher kinded types and constraint kinds [68], e.g., Bottu et al. [11] informally discuss the coherence of implication constraints based on the same assumption. These features are all included in GHC.

Modules. Modules, as supported by GHC, pose an interesting challenge, as they are known to cause a form of ambiguity.[3] GHC does not statically check the uniqueness of instances across modules, thus indirectly allowing users to write overlapping instances, as long as no ambiguity arises during resolution. Adapting our global uniqueness assumption to accommodate this additional freedom remains an interesting challenge.

Laziness. The operational semantics of the $F_{\mathbf{D}}$ and $F_{\{\}}$ calculi in this work are given through standard call-by-name semantics, in order to approximate Haskell's laziness. The system can easily be adapted to either call-by-value or call-by-need, with little impact on the proofs.

It is important to note though, that while expressions are evaluated lazily, type class resolution itself is eager, and constructs the full dictionaries at compile time. This complicates supporting certain GHC features that rely on laziness, like cyclic and infinite dictionaries. They could be supported through loop detection and deferring the construction of dictionaries to runtime, but these would nonetheless pose a significant challenge.

[3] `http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/`

7.7 Related Work

Type Classes. Jones [40, 42] formally proves coherence for the framework of qualified types, which generalizes from type classes to arbitrary evidence-backed type constraints. He focuses on nondeterminism in the typing derivation, and assumes that resolution is coherent.

Morris [60] presents an alternative, denotational semantics for type classes (without superclasses) that avoids elaboration and instead interprets qualified type schemes as the set of denotations of all its monomorphic instantiations that satisfy the qualifiers. The nondeterminism of resolution does not affect these semantics.

Kahl and Scheffczyk [48] present named type class instances that are not used during resolution, but can be explicitly passed to functions. Nevertheless, they violate the uniqueness of instances, and give rise to incoherence of the form illustrated by our *discern* function in Section 7.2.3.

Unlike most other languages with type classes (such as Haskell, Mercury or PureScript) Coq [87] does not enforce the non-overlapping instances condition. Consequently, coherence does not hold for type class resolution in Coq. The reason for this alternative design choice is twofold: (a) Since Coq's type system is more complex than that found in regular programming languages, it is not always possible to decide whether two instances overlap [52, Chapter 2: Typeclasses]. (b) Type class members in Coq are often proofs and, unlike for expressions, users are often indifferent to coherence in the presence of proofs (even though from a semantic point of view, Coq differentiates between them). This concept is known as “proof irrelevance” [28], that is, as long as at least one proof exists, the concrete choice between these proofs is irrelevant. Users can deal with this lack of coherence by either assigning priorities to overlapping instances, or by manually curating the instance database and locally removing specific instances.

Winant and Devriese [98] introduce explicit dictionary application to the Haskell language, and prove coherence for this extended system. Their proof is parametric in the constraint entailment judgment and thus assumes that the constraint solver produces “canonical” evidence. They proceed by introducing a disjointness condition to explicitly applied dictionaries, in order to ensure that coherence is preserved by their extension. This work proves their aforementioned assumption, by establishing coherence for type class resolution.

Dreyer et al. [20] blend ML modules with Haskell type class resolution. Unlike Haskell, they feature multiple global (or outer) scopes; instances within one such global scope must not overlap. Moreover, global instances are shadowed by those given through type signatures. While their language has been formalized, no formal proof of coherence is given.

Implicits. Cochis [83] is a calculus with highly expressive implicit resolution, including local instances. It achieves coherence by imposing restrictions on the implicit context and enforcing a deterministic resolution process. This allows for a much simpler coherence proof.

OCaml's modular implicits [97] do not enforce uniqueness of "instances" but dynamically ensure coherence by rejecting programs where there are multiple possible resolution derivations. This approach has not been formalized yet.

Other. Reynolds [80] introduced the notion of coherence in the context of the Forsythe language's intersection types; he proved coherence directly in terms of the denotational semantics of the language.

In contrast, Bi et al. [7, 8] consider a setting where subtyping for intersection types is elaborated to coercions. Inspired by Biernacki and Polesiuk [9], they use an approach based on contextual equivalence and logical relations, which has inspired us in turn. However, they do not create an intermediate language to avoid the problem of a more expressive target language. This leads to a notion of contextual equivalence that straddles two languages and complicates their proofs.

7.8 Scientific Output

This chapter has presented a formal proof that type class resolution is coherent by means of logical relations and an intermediate language with explicit dictionaries.

The material found in this chapter is largely taken from the following publication:

> Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. Proc. ACM Program. Lang. 3, ICFP, Article 91 (August 2019), 28 pages. DOI:https://doi.org/10.1145/3341695

In this work, the contributions of the different authors are as follows:

- The specification of the $\lambda_{\mathbf{TC}}$, $F_{\mathbf{D}}$ and $F_{\{\}}$ calculi are developed by the author of this thesis, Tom Schrijvers, and later on Ningning Xie.

- The coherence proof is mainly written by the author of this thesis, Tom Schrijvers, and later on Ningning Xie.
- The elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$, as well as the related theorems (Section N in the coherence proof Appendix) are largely written by Koar Marntirosian.
- Proofreading the proof appendix was shared work between the author of this thesis, Koar Marntirosian and Ningning Xie.

Chapter 8

Extension: Quantified Constraints

> "It's still magic even if you know how it's done."
>
> A Hat Full of Sky
> Terry Pratchett

8.1 Introduction

Over the years type classes have been the subject of many language extensions that increase their expressive power and enable new applications. Examples of such extensions include: multi-parameter type classes [47]; functional dependencies [46]; or associated types [15]. Several of these implemented extensions were inspired by the analogy between type classes and predicates in Horn clauses. Yet, Horn clauses have their limitations. As a small side-product of their work on derivable type classes, Hinze and Peyton Jones [38] have proposed to raise the expressive power of type classes to essentially the universal fragment of Hereditiary Harrop logic [34] with what they call *quantified class constraints*. Their motivation was to deal with higher-kinded types which seemed to require instance declarations that were impossible to express in the type-class system of Haskell at that time.

Unfortunately, Hinze and Peyton Jones never did elaborate on quantified class constraints. Later, Lämmel and Peyton Jones [50] found a workaround for the particular problem of the derivable type classes work that did not involve quantified class constraints. Nevertheless the idea of quantified class constraints has whet the appetite of many researchers and developers. GHC ticket #2893[1], requesting for quantified class constraints, was opened in 2008 and is still open today. Commenting on this ticket in 2009, Peyton Jones states that *"their lack is clearly a wart, and one that may become more pressing"*, yet clarifies in 2014 that *"(t)he trouble is that I don't know how to do type inference in the presence of polymorphic constraints."* In 2010, 10 years after the original idea, Hinze [36] rues that the feature has not been implemented yet. As recently as 2016, Chauhan et al. [16] regret that "*Haskell does not allow the use of universally quantified constraints*" and now in 2017 Spivey [88] has to use pseudo-Haskell when modeling with quantified class constraints. While various workarounds have been proposed and are used in practice [49, 82, 93], none has stopped the clamor for proper quantified class constraints.

This work finally elaborates the original idea of quantified class constraints into a fully fledged language design.

Specifically, the contributions of this work are:

- We provide an overview of the two main advantages of quantified class constraints (Section 8.2):
  1. they provide a natural way to express more of a type class's specification, and
  2. they enable terminating type class resolution for a larger class of applications.
- We elaborate the type system sketch of Hinze and Peyton Jones [38] for quantified type class constraints into a full-fledged formalization (Section 8.3). Our formalization borrows the idea of focusing from Cochis [83], a calculus for Scala-style implicits [66, 67], and adapts it to the Haskell setting. We account for two notable differences: a global set of non-overlapping instances and support for superclasses.
- We present a type inference algorithm that conservatively extends that of Haskell 98 (Section 8.4) and comes with a dictionary-passing elaboration into System F (Section 8.5).
- We discuss the termination conditions on a system with quantified class constraints (Section 8.6).

[1] `https://ghc.haskell.org/trac/ghc/ticket/2893`

- We provide a prototype implementation, which incorporates higher-kinded datatypes and accepts all[2] examples in this work, at `https://github.com/gkaracha/quantcs-impl`.

8.2 Motivation

This section illustrates the expressive power afforded by quantified class constraints to capture several requirements of type class instances more succinctly, and to provide terminating resolution for a larger group of applications.

8.2.1 Precise and Succinct Specifications

Monad Transformers Consider the MTL type class for monad transformers [45]:

```
class Trans t where
  lift :: Monad m ⇒ m a → (t m) a
```

What is not formally expressed in the above type class declaration, but implicitly expected, is that for any type *T* that instantiates *Trans* there should also be a *Monad* instance of the form:

```
instance Monad m ⇒ Monad (T m) where ...
```

Because the type checker is not told about this requirement, it will not accept the following definition of monad transformer composition.

```
newtype (t1 ⋆ t2) m a = C {runC :: t1 (t2 m) a}
instance (Trans t1, Trans t2) ⇒ Trans (t1 ⋆ t2) where
  lift = C ∘ lift ∘ lift
```

The idea of this code is to *lift* from monad *m* to (*t2 m*) and then to *lift* from (*t2 m*) to *t1* (*t2 m*). However, the second *lift* is only valid if *t2 m* is a monad and the type checker has no way of establishing that this fact holds for all monad transformers *t2*. Workarounds for this problem do exist in current Haskell [39, 82, 93], but they clutter the code with heavy encodings.

Quantified class constraints allow us to state this requirement explicitly as part of the *Trans* class declaration:

[2] except for the *HFunctor* example (Section 8.2.1), which needs higher-rank types [73].

```
class (∀ m. Monad m ⇒ Monad (t m)) ⇒ Trans t where
  lift :: Monad m ⇒ m a → (t m) a
```

The instance for transformer composition $t_1 * t_2$ now typechecks.

Second-Order Functors Another example can be found in the work of Hinze [37]. He represents parameterized datatypes, like polymorphic lists and trees, as the fixpoint *Mu* of a *second-order functor*:

```
data Mu h a = In {out :: h (Mu h) a}
data List2 f a = Nil | Cons a (f a)
type List = Mu List2
```

A second-order functor *h* is a type constructor that sends functors to functors. This can be concisely expressed with the quantified class constraint ∀ *f*. *Functor f* ⇒ *Functor* (*h f*), for example in the *Functor* instance of *Mu*:

```
instance (∀ f. Functor f ⇒ Functor (h f)) ⇒ Functor (Mu h)
  where fmap f (In x) = In (fmap f x)
```

Although this is Hinze's preferred formulation he remarks that:

> *Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 98 [93], but the resulting code is somewhat clumsy.*

Johann and Ghani use essentially the same data-generic representation, the fixpoint of second-order functors, to represent so-called *nested datatypes* [10]. For instance, [36] represents perfect binary trees with the nested datatype

```
data Perfect a = Zero a | Succ (Perfect (a, a))
```

This can be expressed with the generic representation as *Mu HPerf*, the fixpoint of the second-order functor *HPerf*, defined as

```
data HPerf f a = HZero a | HSucc (f (a, a))
```

Johann and Ghani's notion of second-order functor differs slightly from Hinze's.[3] Ideally, their notion would be captured by the following class declaration:

```
class (∀ f. Functor f ⇒ Functor (h f)) ⇒ HFunctor h where
  hfmap :: (Functor f, Functor g)
     ⇒ (∀ x. f x → g x) → (∀ x. h f x → h g x)
```

[3] It is more in line with the category theoretical notion of endofunctors over the category of endofunctors.

Like in Hinze's case, the quantified class constraint expresses that a second-order functor takes first-order functors to first-order functors. Additionally, second-order functors provide a second-order *fmap*, called *hfmap*, which replaces *f* by *g*, to take values of type *h f x* to type *h g x*. Yet, in the absence of actual support for quantified class constraints, Johann and Ghani provide the following declaration instead:

```
class HFunctor h where
ffmap :: Functor f ⇒ (a → b) → (h f a → h f b)
hfmap :: (Functor f, Functor g)
   ⇒ (∀ x. f x → g x) → (∀ x. h f x → h g x)
```

In essence, they inline the *fmap* method provided by the quantified class constraint in the *HFunctor* class. This is unfortunate because it duplicates the *Functor* class's functionality.

8.2.2 Terminating Corecursive Resolution

Quantified class constraints were first proposed by Hinze and Peyton Jones [38] as a solution to a problem of diverging type class resolution. Consider their generalized rose tree datatype

```
data GRose f a = GBranch a (f (GRose f a))
```

and its *Show* instance

```
instance (Show a, Show (f (GRose f a))) ⇒ Show (GRose f a)
   where show (GBranch x xs) = unwords [show x, “ − “, show xs]
```

Notice the two constraints in the instance context which are due to the two *show* invocations in the method definition. Standard recursive type class resolution would diverge when faced with the constraint (*Show* (*GRose* [ ] *Bool*)). Indeed, it would recursively resolve the instance context: *Show Bool* is easily dismissed, but *Show* [*GRose* [ ] *a*] requires resolving *Show* (*GRose* [ ] *Bool*) again. Clearly this process loops.

To solve this problem, Hinze and Peyton Jones proposed to write the *GRose* instance with a quantified type class constraint as:

```
instance (Show a, ∀ x. Show x ⇒ Show (f x)) ⇒ Show (GRose f a)
   where show (GBranch x xs) = unwords [show x, “ − “, show xs]
```

This would avoid the diverging loop in the type system extension they sketch, because the two recursive resolvents, *Show Bool* and ∀ *x*. *Show x* ⇒ *Show* [*x*] are readily discharged with the available *Bool* and [*a*] instances.

When faced with the same looping issue in their *Scrap Your Boilerplate* work, [51] implemented a different solution: *cycle-aware constraint resolution.* This approach detects that a recursive resolvent is identical to one of its ancestors and then ties the (co-)recursive knot at the level of the underlying type class dictionaries.

Unfortunately, cycle-aware resolution is not a panacea. It only deals with a particular class of diverging resolutions, those that cycle. The fixpoint of the second-order functor *HPerf* presented above is beyond its capabilities.

```
instance (Show (h (Mu h) a)) ⇒ Show (Mu h a) where
  show (In x) = show x
instance (Show a, Show (f (a, a))) ⇒ Show (HPerf f a) where
  show (HZero a) = “(Z“ ++ show a ++ “)“
  show (HSucc xs) = “(S“ ++ show xs ++ “)“
```

Resolving *Show* (*Mu HPerf Int*) diverges without cycling back to the original constraint due to the nestedness of the perfect tree type: In contrast, with quantified type class constraints we can formulate the instances in a way that resolution does terminate.

```
instance (Show a,
  ∀ f x. (Show x, ∀ y. Show y ⇒ Show (f y)) ⇒ Show (h f x))
   ⇒ Show (Mu h a) where show (In x) = show x
instance (Show a, ∀ x. Show x ⇒ Show (f x)) ⇒ Show (HPerf f a) where
  show (HZero a) = “(Z“ ++ show a ++ “)“
  show (HSucc xs) = “(S“ ++ show xs ++ “)“
```

8.2.3 Summary

In summary, quantified type class constraints enable (1) expressing more of a type class's specification in a natural and succinct manner, and (2) terminating type class resolution for a larger group of applications.

In the remainder of this chapter we provide a declarative type system for a Haskell-like calculus with quantified class constraints (Section 8.3). Type inference is shown in Section 8.4 and Section 8.5 provides an elaboration into System F. Section 8.6 presents the conditions we require to ensure termination in the presence of quantified class constraints. Finally, Section 8.7 discusses related work and Section 8.9 concludes.

$$
\begin{array}{lcll}
pgm & ::= & e \mid cls; pgm \mid inst; pgm & \textit{program} \\
cls & ::= & \textbf{class } A \Rightarrow TC\ a \textbf{ where } \{\ f :: \sigma\ \} & \textit{class decl.} \\
inst & ::= & \textbf{instance } A \Rightarrow TC\ \tau \textbf{ where } \{\ f = e\ \} & \textit{instance decl.} \\
 & & & \\
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let } x = e_1 \textbf{ in } e_2 & \textit{term} \\
 & & & \\
\tau & ::= & a \mid \tau_1 \rightarrow \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid C \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme} \\
 & & & \\
A & ::= & \bullet \mid A, C & \textit{axiom set} \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a.C & \textit{constraint} \\
Q & ::= & TC\ \tau & \textit{class constraint} \\
 & & & \\
P & ::= & \langle A_S, A_I, A_L \rangle & \textit{program theory}
\end{array}
$$

Figure 8.1: Source Syntax

8.3 Declarative Type System

This section provides the declarative type system specification for $\lambda_{\mathbf{TC}}^{\Rightarrow}$, an extension of $\lambda_{\mathbf{TC}}$ with quantified class constraints.

8.3.1 Syntax

Figure 8.1 presents the, mostly standard, syntax of our source language. A program pgm consists of class declarations cls, instance declarations $inst$ and a top-level expression e. For simplicity, each class has a single parameter and a single method.

Terms e comprise a λ-calculus extended with let-bindings. By convention, we use f to denote a method name and x, y, z to denote any kind of term variable name.

Types also appear in Figure 8.1. Like all extensions of the Damas-Milner system [17] with qualified types [41], we discriminate between monotypes τ, qualified types ρ and type schemes σ. Note that, to avoid clutter, our formalization does not feature higher-kinded types, but our prototype implementation does.

Our calculus differs from Haskell'98 in that it conservatively generalizes the

$\boxed{P; \Gamma \vdash_{\texttt{tm}} e : \sigma}$ (Term Typing)

$$\frac{x \notin \mathbf{dom}(\Gamma) \qquad \Gamma \vdash_{\texttt{ty}} \tau_1 \\ P; \Gamma, x : \tau_1 \vdash_{\texttt{tm}} e : \tau_2}{P; \Gamma \vdash_{\texttt{tm}} \lambda x.e : \tau_1 \to \tau_2}\;(\to\mathrm{I}) \qquad \frac{P; \Gamma \vdash_{\texttt{tm}} e_1 : \tau_1 \to \tau_2 \\ P; \Gamma \vdash_{\texttt{tm}} e_2 : \tau_1}{P; \Gamma \vdash_{\texttt{tm}} e_1\ e_2 : \tau_2}\;(\to\mathrm{E})$$

$$\frac{x \notin \mathbf{dom}(\Gamma) \qquad P; \Gamma, x : \tau \vdash_{\texttt{tm}} e_1 : \tau \qquad P; \Gamma, x : \tau \vdash_{\texttt{tm}} e_2 : \sigma}{P; \Gamma \vdash_{\texttt{tm}} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \sigma}\;\textsc{TmLet}$$

$$\frac{(x : \sigma) \in \Gamma}{P; \Gamma \vdash_{\texttt{tm}} x : \sigma}\;\textsc{TmVar} \qquad \frac{\Gamma \vdash_{\texttt{ct}} C \\ P,_{\mathrm{L}}\ C; \Gamma \vdash_{\texttt{tm}} e : \rho}{P; \Gamma \vdash_{\texttt{tm}} e : C \Rightarrow \rho}\;(\Rightarrow\mathrm{I}) \qquad \frac{P; \Gamma \vdash_{\texttt{tm}} e : C \Rightarrow \rho \\ P; \Gamma \models C}{P; \Gamma \vdash_{\texttt{tm}} e : \rho}\;(\Rightarrow\mathrm{E})$$

$$\frac{a \notin \Gamma \qquad P; \Gamma, a \vdash_{\texttt{tm}} e : \sigma}{P; \Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma}\;(\forall\mathrm{I}) \qquad \frac{P; \Gamma \vdash_{\texttt{tm}} e : \forall a.\sigma \\ \Gamma \vdash_{\texttt{ty}} \tau}{P; \Gamma \vdash_{\texttt{tm}} e : [a \mapsto \tau]\sigma}\;(\forall\mathrm{E})$$

$\boxed{\Gamma \vdash_{\texttt{cls}} \mathit{cls} : A_S; \Gamma_c}$ (Class Declaration Typing)

$$\frac{\Gamma, a \vdash_{\texttt{ct}} C_i \qquad \Gamma, a \vdash_{\texttt{ty}} \sigma \qquad \Gamma_c = [f : \forall a. TC\ a \Rightarrow \sigma]}{\Gamma \vdash_{\texttt{cls}} \mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} : [\overline{\forall a. TC\ a \Rightarrow C_i}]; \Gamma_c}\;\textsc{Class}$$

$\boxed{P; \Gamma \vdash_{\texttt{inst}} \mathit{inst} : A_I}$ (Class Instance Typing)

$$\frac{\overline{b} = \mathbf{fv}(\tau) \qquad \Gamma, \overline{b} \vdash_{\texttt{ax}} A \\ \mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} \\ P,_{\mathrm{L}}\ A; \Gamma, \overline{b} \models [\tau/a]C_i \qquad P,_{\mathrm{L}}\ A,_{\mathrm{L}}\ TC\ \tau; \Gamma, \overline{b} \vdash_{\texttt{tm}} e : [\tau/a]\sigma}{P; \Gamma \vdash_{\texttt{inst}} \mathbf{instance}\ A \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \} : [\forall \overline{b}.A \Rightarrow TC\ \tau]}\;\textsc{Instance}$$

Figure 8.2: Declarative Type System (Selected Rules)

language of constraints. In Haskell'98 the constraints that can appear in type signatures and in class and instance contexts are basic class constraints Q of the form $TC\,\tau$. As a consequence, the constraint schemes or axioms that are derived from instances (and for superclasses) are Horn clauses of the form:

$$\forall \bar{a}. Q_1 \wedge \ldots \wedge Q_n \Rightarrow Q_0$$

These axioms are similar to rank-1 polymorphic types in the sense that the quantifiers (and the implication) only occur on the outside. We allow a more general form of constraints C where, in analogy with higher-rank types, quantifiers and implications occur in nested positions. This more expressive form of constraints can occur in signatures and class/instance contexts. Consequently, the syntactic sort C of constraints and axioms is one and the same.

Note that constraint schemes of the form $\forall \bar{a}.(Q_1 \wedge \ldots \wedge Q_n) \Rightarrow Q_0$, used in earlier formalizations of type classes (e.g., Morris [60]), are not valid syntax for our constraints C because we do not provide a notation for conjunction. Yet, we can easily see the scheme notation as syntactic sugar for a curried representation:

$$\forall \bar{a}.(Q_1 \wedge \cdots \wedge Q_n) \Rightarrow Q_0 \quad \equiv \quad \forall \bar{a}. Q_1 \Rightarrow (\ldots (Q_n \Rightarrow Q_0) \ldots)$$

We denote a list of C-constraints as A, short for *axiom set* as we use them to represent, among others, axioms given through type class instances.

Finally, Figure 8.1 presents typing environments Γ, which are entirely standard, and the program theory P. The latter is a triple of three axiom sets: the superclass axioms A_S, the instance axioms A_I and local axioms A_L. We use the notation $P,_{\mathrm{L}}\, C$ to denote that we extend the local component of the triple, and similar notation for the other components. In earlier type class formalizations these separate kinds of axioms are typically conflated into a single axiom set. However, in this chapter it is convenient to distinguish them for accurately stating the different restrictions imposed on them. Moreover, it is instructive for contrasting with regular Haskell. In Haskell, the local constraints are basic type class constraints Q only, while the instance and superclass axioms have the more expressive Horn clause form. In contrast, in our setting all three components support the same (and more general) form of *Harrop formulae.*

8.3.2 The Type System

Figure 8.2 presents the main judgments of our declarative type system for the language of Figure 8.1, namely term typing and typing of class and instance declarations.

Type & Constraint Well-Scopedness The judgments for well-scopeness of types, constraints and axiom sets are denoted $\Gamma \vdash_{\mathtt{ty}} \sigma$, $\Gamma \vdash_{\mathtt{ct}} C$ and $\Gamma \vdash_{\mathtt{ax}} A$ respectively. Their definitions are straightforward and can be found in Appendix A.4.

Term Typing Term typing takes the form $P; \Gamma \vdash_{\mathtt{tm}} e : \sigma$ and can be read as *"under program theory P and typing environment Γ, expression e has type σ"*. The rules are almost literally those of Chakravarty et al. [15]. There is only one difference, which is a simplification for the sake of convenience: following Vytiniotis et al. [94] we have opted for recursive let-bindings that are not generalized. Nevertheless, we generalize the type of top-level bindings (see Appendix A.4).

Apart from that, there are no noticeable differences with conventional Haskell in the typing rules. All the interesting differences are concentrated in the definition of the constraint entailment judgment $P; \Gamma \models C$, which is used in the constraint elimination Rule ($\Rightarrow$E). The definition of this auxiliary judgment is discussed in detail in Section 8.3.3.

Class Declaration Typing Typing for class declarations takes the form $\Gamma \vdash_{\mathtt{cls}} \mathit{cls} : A_S; \Gamma_c$ and is given by Rule Class, presented in Figure 8.2.

In addition to checking the well-formedness of the method type, we ensure that the class context $(C_1, \ldots, C_n)$ is also well-formed, extending the environment with the local variable a. In turn, this implies that $\mathit{fv}(C_i) \subseteq \{a\}$, in line with the Haskell standard.

As usual, typing a class declaration extends the typing environment with the method typing, and the program's theory with the superclass axioms. For instance, the extended monad transformer class yields the superclass axiom:

$$\forall\, t. \mathit{Trans}\ t \Rightarrow (\forall\, m. \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (t\ m))$$

Class Instance Typing Instance typing takes the form $P; \Gamma \vdash_{\mathtt{inst}} \mathit{inst} : A_I$ and is given by Rule Instance, also presented in Figure 8.2.

We check the well-formedness of the instance context A under the extended typing environment, and that each superclass constraint C_i is entailed by the instance context.

Finally, we check that the method implementation e has the type indicated by the class declaration, appropriately instantiated for the instance in question.

Program Typing The judgment for program typing ties everything together and takes the form $P; \Gamma \vdash_{\mathtt{pgm}} pgm : \sigma$. Its definition is straightforward and can be found in Appendix A.4.

8.3.3 Constraint Entailment

Following the approach of Schrijvers et al. [83] for their Cochis calculus, we present constraint entailment in two steps. First, we provide an easy-to-understand and expressive, yet also highly ambiguous, specification. Then we present a syntax-directed, semi-algorithmic variant that takes the ambiguity away, but has a more complicated formulation inspired by the *focusing* technique used in proof search [5, 54, 58].

Declarative Specification Constraint entailment takes the form $P; \Gamma \models C$, and its high-level declarative specification is given by the following rules:

$$\frac{a \notin \Gamma \qquad P; \Gamma, a \models C}{P; \Gamma \models \forall a. C}\ (\forall\text{IC}) \qquad \frac{P; \Gamma \models \forall a. C \qquad \Gamma \vdash_{\mathtt{ty}} \tau}{P; \Gamma \models [\tau/a] C}\ (\forall\text{EC})$$

$$\frac{C \in P}{P; \Gamma \models C}\ (\textsc{SpecC}) \qquad \frac{P,_{\text{L}} C_1; \Gamma \models C_2}{P; \Gamma \models C_1 \Rightarrow C_2}\ (\Rightarrow\text{IC}) \qquad \frac{\begin{array}{c} P; \Gamma \models C_1 \Rightarrow C_2 \\ P; \Gamma \models C_1 \end{array}}{P; \Gamma \models C_2}\ (\Rightarrow\text{EC})$$

If we interpret constraints C as logical formulas, the above rules are nothing more than the rules of the universal fragment of Hereditiary Harrop logic [34]. Rule (SpecC) is the standard axiom rule. Rules (⇒IC) and (⇒EC) correspond to implication introduction and elimination, respectively. Similarly, Rules (∀IC) and (∀EC) correspond to introduction and elimination of universal quantification, respectively. These are also essentially the rules Hinze and Peyton Jones [38] propose.

While compact and elegant, there is a serious downside to these rules: They are highly ambiguous and give rise to many trivially different proofs for the same constraint. For instance, assuming $\Gamma = \bullet, a$ and $P = \langle \bullet, \bullet, Eq\, a \rangle$, here are only two of the infinitely many proofs of $P; \Gamma \models Eq\, a$:

$$\frac{Eq\, a \in P}{P; \Gamma \models Eq\, a}\ (\textsc{SpecC})$$

$\boxed{P;\Gamma \models C}$ (Constraint Entailment)

$$\frac{P;\Gamma \models [C]}{P;\Gamma \models C}$$

$\boxed{P;\Gamma \models [C]}$ (Constraint Resolution)

$$\frac{P,_{\text{L}}\, C_1;\Gamma \models [C_2]}{P;\Gamma \models [C_1 \Rightarrow C_2]}\ (\Rightarrow\text{R}) \qquad \frac{P;\Gamma, b \models [C]}{P;\Gamma \models [\forall b.C]}\ (\forall\text{R})$$

$$\frac{C \in P :\ \Gamma;[C] \models Q \rightsquigarrow A \qquad \forall C_i \in A :\ \ P;\Gamma \models [C_i]}{P;\Gamma \models [Q]}\ (Q\text{R})$$

$\boxed{\Gamma;[C] \models Q \rightsquigarrow A}$ (Constraint Matching)

$$\frac{\Gamma;[C_2] \models Q \rightsquigarrow A}{\Gamma;[C_1 \Rightarrow C_2] \models Q \rightsquigarrow A, C_1}\ (\Rightarrow\text{L})$$

$$\frac{\Gamma;[[\tau/b]C] \models Q \rightsquigarrow A \qquad \Gamma \vdash_{\text{ty}} \tau}{\Gamma;[\forall b.C] \models Q \rightsquigarrow A}\ (\forall\text{L}) \qquad \frac{}{\Gamma;[Q] \models Q \rightsquigarrow \bullet}\ (Q\text{L})$$

Figure 8.3: Tractable Constraint Entailment

versus

$$\frac{\dfrac{\dfrac{Eq\,a \in P'}{P';\Gamma \models Eq\,a}\ (\textsc{SpecC})}{P;\Gamma \models Eq\,a \Rightarrow Eq\,a}\ (\Rightarrow\text{IC}) \qquad \dfrac{Eq\,a \in P}{P;\Gamma \models Eq\,a}\ (\textsc{SpecC})}{P;\Gamma \models Eq\,a}\ (\Rightarrow\text{EC})$$

where $P' = P,_{\text{L}}\, Eq\,a$. Observe that the latter proof makes an unnecessary appeal to implication introduction.

Type-Directed Specification To avoid the trivial forms of ambiguity like in the example, we adopt a solution from proof search known as *focusing* [5]. This solution was already adopted by the Cochis calculus, for the same reason. The key idea of focusing is to provide a syntax-directed definition of constraint entailment where only one inference rule applies at any given time.

Figure 8.3 presents our definition of constraint entailment with focusing. The main judgment $P; \Gamma \models C$ is defined in terms of two auxiliary judgments, $P; \Gamma \models [C]$ and $\Gamma; [C] \models Q \rightsquigarrow A$, each of which is defined by structural induction on the constraint enclosed in square brackets.

The main entailment judgment is equivalent to the first auxiliary judgment $P; \Gamma \models [C]$. This auxiliary judgment focuses on the constraint C whose entailment is checked – we call this constraint the "goal". There are three rules, for the three possible syntactic forms of C. Rules ($\Rightarrow$R) and ($\forall$R) decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple class constraint Q, Rule (QR) selects an axiom C from the theory P to discharge it. The selected axiom must *match* the goal, a notion that is captured by the second auxiliary judgment. Matching gives rise to a sequence A of new (and hopefully simpler) goals whose entailment is checked recursively.

The second auxiliary judgment $\Gamma; [C] \models Q \rightsquigarrow A$ focuses on the axiom C and checks whether it matches the simple goal Q. Again, there are three rules for the three possible forms the axiom can take. Rule (QL) expresses the base case where the axiom is identical to the goal and there are no new goals. Rule ($\Rightarrow$L) handles an implication axiom $C_1 \Rightarrow C_2$ by recursively checking whether C_2 matches the goal. At the same time it yields a new goal C_1 which needs to be entailed in order for the axiom to apply. Finally, Rule ($\forall$L) handles universal quantification by instantiating the quantified variable in a way that recursively yields a match.

It is not difficult to see that this type-directed formulation of entailment greatly reduces the number of proofs for given goal.[4] For instance, for the example above there is only one proof:

$$\dfrac{\dfrac{Eq\,a \in P \qquad \Gamma; [Eq\,a] \models Eq\,a \rightsquigarrow \bullet \; (Q\mathrm{L})}{P; \Gamma \models [Eq\,a]}\;(Q\mathrm{R})}{P; \Gamma \models Eq\,a}$$

8.3.4 Remaining Nondeterminism

While focusing makes the definition of constraint entailment type-directed, there are still two sources of nondeterminism. As a consequence, the specification is still ambiguous and not an algorithm.

[4] Without loss of expressive power. See for example Pfenning [74].

Overlapping Axioms The first source of non-determinism is that in Rule $(Q\mathrm{R})$ there may be multiple matching axioms that make the entailment go through. For applications of logic where proofs are irrelevant this is not a problem, but in Haskell where the proofs have computational content (namely the method implementations) this is a cause for concern. Haskell'98 also faces this problem. Consider two instances for the same type:

```
class Default a where {default :: a}
instance Default Bool where {default = True}
instance Default Bool where {default = False}
```

The two instances give rise to two different proofs for *Default Bool*, with distinct computational content (*True* vs. *False*). We steer away from this problem in the same was as Haskell'98, by requiring that instance declarations do not overlap. This does not rule out the possibility of distinct proofs for the same goal, but at least distinct proofs have the same computational content. Consider a class hierarchy where *C* is the superclass of both *D* and *E*.

```
class C a where {...}
class C a ⇒ D a where {...}
class C a ⇒ E a where {...}
```

This gives rise to the superclass axioms $\forall\ a.\, D\ a \Rightarrow C\ a$ and $\forall\ a.\, E\ a \Rightarrow C\ a$. Given additionally two local constraints *D ty* and *E ty*, we have two ways to establish *C ty*. The proofs are distinct, yet ultimately the computational content is the same. This is easy to see as only instances supply the computational content and there can be at most one instance for any given type *ty*.

In summary, non-overlap of instances is sufficient to ensure *coherence*.

Guessing Polymorphic Instantiation A second source of ambiguity is that Rule $(\forall\mathrm{L})$ requires guessing an appropriate type τ for substituting the type variable b. Guessing is problematic because there are an infinite number of types to choose from and more than one of those choices can make the entailment work out. Choosing an appropriate type is a problem for the type inference algorithm in the next section. Different choices leading to different proofs is a more fundamental problem that also manifests itself in Haskell'98. Consider the following instances.

```
instance C Char where {...}
instance C Bool where {...}
instance C a ⇒ D Int where {...}
```

The third instance gives rise to the axiom $\forall\ a.\, C\ a \Rightarrow D\ Int$. When resolving *D Int* with this axiom we can choose *a* to be either *Char* or *Bool* and thus select a different *C* instance.

$\boxed{unamb(C)}$ (Unambiguity)

$$\frac{\bullet \vdash_{\mathtt{unamb}} C}{unamb(C)}\ \textsc{Unamb}$$

$\boxed{\overline{a} \vdash_{\mathtt{unamb}} C}$ (Unambiguity)

$$\frac{\overline{a} \subseteq fv(Q)}{\overline{a} \vdash_{\mathtt{unamb}} Q}\ (Q\mathrm{U}) \qquad \frac{\overline{a}, a \vdash_{\mathtt{unamb}} C}{\overline{a} \vdash_{\mathtt{unamb}} \forall a.C}\ (\forall\mathrm{U}) \qquad \frac{unamb(C_1) \qquad \overline{a} \vdash_{\mathtt{unamb}} C_2}{\overline{a} \vdash_{\mathtt{unamb}} C_1 \Rightarrow C_2}\ (\Rightarrow\mathrm{U})$$

Figure 8.4: Unambiguity

Haskell'98 avoids this problem by requiring that all quantified type variables, like a in the example, appear in the head of the axiom. Because our axioms have a more general, recursively nested form, we generalize this requirement in a recursively nested fashion. The predicate $unamb(C)$ in Figure 8.4 formalizes the requirement in terms of the auxiliary judgment $\overline{a} \vdash_{\mathtt{unamb}} C$, where $\overline{a}$ are type variables that need to be determined by the head of C. Rule $(Q\mathrm{U})$ constitutes the base case where Q is the head and contains the determinable type variables $\overline{a}$. Rule $(\forall\mathrm{U})$ processes a quantifier by adding the new type variable to the list of determinable type variables $\overline{a}$. Finally, Rule $(\Rightarrow\mathrm{U})$ checks whether the head C_2 of the implication determines the type variables $\overline{a}$. It also recursively checks whether C_1 is unambiguous on its own. The latter check is necessary because left-hand sides of implications are themselves added as axioms to the theory in Rule (⇒R); hence they must be well-behaved on their own.

The predicate $unamb(C)$ must be imposed on all constraints that are added to the theory. This happens in four places: the instance axioms added in Rule Instance, the superclass axioms added in Rule Class, the local axioms added when checking against a given signature in Rule (⇒I) and the local axioms added during constraint entailment checking in Rule (⇒R). These four places can be traced back to three places in the syntax: class and instance heads, and (method) signatures.

8.4 Type Inference

We provide a type inference algorithm with elaboration into System F [30]. To simplify the presentation, this section focuses solely on type inference. The

$\boxed{\Gamma \vdash_{\mathtt{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E}$ (Term Typing)

$$\frac{\bar{b}, \bar{d} \text{ fresh} \qquad (x : \forall \bar{a}. \bar{C} \Rightarrow \tau) \in \Gamma}{\Gamma \vdash_{\mathtt{tm}} x : [\bar{b}/\bar{a}]\tau \rightsquigarrow x\ \bar{b}\ \bar{d} \mid \overline{(d : [\bar{b}/\bar{a}]C)}; \bullet}\ \textsc{TmVar}$$

$$\frac{\begin{array}{c} a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathtt{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{A}_1; E_1 \\ \Gamma, x : \tau_1 \vdash_{\mathtt{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{A}_2; E_2 \qquad \vdash_{\mathtt{ty}} \tau_1 \rightsquigarrow \upsilon_1 \end{array}}{\Gamma \vdash_{\mathtt{tm}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_2 \mid (\mathcal{A}_1, \mathcal{A}_2); (E_1, E_2, a \sim \tau_1)}\ \textsc{TmLet}$$

$$\frac{a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathtt{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E}{\Gamma \vdash_{\mathtt{tm}} \lambda x.e : a \to \tau \rightsquigarrow \lambda(x : a).t \mid \mathcal{A}; E}\ \textsc{TmAbs}$$

$$\frac{a \text{ fresh} \qquad \Gamma \vdash_{\mathtt{tm}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{A}_1; E_1 \qquad \Gamma \vdash_{\mathtt{tm}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{A}_2; E_2}{\Gamma \vdash_{\mathtt{tm}} e_1\ e_2 : a \rightsquigarrow t_1\ t_2 \mid (\mathcal{A}_1, \mathcal{A}_2); (E_1, E_2, \tau_1 \sim \tau_2 \to a)}\ \textsc{TmApp}$$

Figure 8.5: Constraint Generation for Terms with Elaboration

parts of the rules highlighted in gray concern elaboration and are discussed in Section 8.5.

To make the connection to the relations of the declarative specification (Section 8.3.2) more clear, corresponding rules share the same name.

8.4.1 Preliminaries

Before diving into the details of the algorithm, we first introduce some additional notation and constructs.

Variable-Annotated Constraints & Type Equalities Since our goal is to perform type inference and elaboration to System F simultaneously, we annotate all constraints with their corresponding System F evidence term (dictionary variable d). We keep the notational burden minimal by reusing the same letters as in Figure 8.1, yet with a calligraphic font:

$\mathcal{P}$	$::=$	$\langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{A}_L \rangle$	*variable-annotated theory*
$\mathcal{A}$	$::=$	$\bullet \mid \mathcal{A}, \mathcal{C}$	*variable-annotated axiom set*
$\mathcal{C}$	$::=$	$d : C$	*variable-annotated constraint*
$\mathcal{Q}$	$::=$	$d : Q$	*variable-annotated class constraint*

Additionally, like every HM(X)-based system, our type-inference algorithm proceeds by first generating type constraints from the program text (constraint generation) and then solving these constraints independently of the program text (constraint solving).

During constraint generation, our algorithm gives rise to both (variable-annotated) constraints $\mathcal{A}$, as well as type equalities E:

$$E \quad ::= \quad \bullet \mid E, \tau_1 \sim \tau_2 \qquad\qquad \textit{type equalities}$$

Type & Evidence Substitutions Furthermore, we introduce two kinds of substitutions: type substitutions θ and dictionary substitutions η:

$$\begin{array}{lcll} \theta & ::= & \bullet \mid \theta \cdot [\tau/a] & \textit{type substitution} \\ \eta & ::= & \bullet \mid \eta \cdot [t/d] & \textit{evidence substitution} \end{array}$$

A type substitution θ maps type variables to monotypes, while an evidence substitution η maps dictionary variables d to System F terms t (see Section 8.5.1 for the formal syntax of System F terms).

8.4.2 Constraint Generation For Terms

Figure 8.5 presents constraint generation for terms. The relation takes the form $\Gamma \vdash_{\mathtt{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E$. Given a typing environment Γ and a term e we infer (1) a monotype τ, (2) a set of wanted constraints $\mathcal{A}$, and (3) a set of wanted equalities E. Its definition is standard.

Rule TmVar handles variables. We instantiate the polymorphic type $\forall \overline{a}.\overline{C} \Rightarrow \tau$ of a term variable x with fresh unification variables $\overline{b}$, introducing $\overline{C}$ as wanted constraints, instantiated likewise. Rule TmAbs assigns a fresh unification variable to the abstracted term variable x, and adds it to the context for checking the body of the abstraction. Rule TmApp handles applications $(e_1\ e_2)$. We collect wanted class and equality constraints from each subterm, we generate a fresh type variable a for the result and record that the type of e_1 is a function type $(\tau_1 \sim \tau_2 \rightarrow a)$. Rule TmLet handles (possibly recursive) let bindings.

8.4.3 Constraint Solving

The type class and equality constraints derived from terms are solved with the following two algorithms.

Solving Equality Constraints We solve a set of equality constraints E by means of unification. The function $\mathit{unify}(\overline{a}; E) = \theta_{\perp}$ takes the set of equalities and a set of "untouchable" type variables, and returns either the most general unifier θ of the equalities or fails if none exists. The untouchable type variables $\overline{a}$ originate from type signatures; all other type variables are unification variables. The unifier is of course only allowed to substitute unification variables.

The definition of this unification function is folklore, following Damas and Milner [17] and accounting for signatures; it can be found in Appendix A.4.

Solving Type Class Constraints Figure 8.6 defines the judgment for solving type class constraints; it takes the form $\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_2; \eta$. Given a set of untouchable type variables $\overline{a}$ and a theory $\mathcal{P}$, it (exhaustively) replaces a set of constraints $\mathcal{A}_1$ with a set of *simpler*, residual constraints $\mathcal{A}_2$, via the auxiliary judgment $\overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A}; \eta$, explained below.

This form differs from the specification in Figure 8.3: we allow constraints to be partially entailed, which in turn allows us to perform *simplification* [43] of top-level signatures. This is standard practice in Haskell when inferring types. For instance, when inferring the signature for

$$f\ x = [x] == [x]$$

Haskell simplifies the derived constraint $Eq\ [a]$ to $Eq\ a$, yielding the signature $\forall\ a. Eq\ a \Rightarrow a \rightarrow Bool$.

Simplification Auxiliary judgment $\overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A}; \eta$ uses the theory $\mathcal{P}$ to simplify a single constraint $\mathcal{C}$ to a set of simpler constraints without instantiating any of the untouchable type variables $\overline{a}$. Following the focusing approach, the judgment is defined by three rules, one for each of the syntactic forms of the goal $\mathcal{C}$.

Rules ($\Rightarrow$R) and ($\forall$R) recursively simplify the head of the goal. Observe that we add the bound variable b to the untouchables $\overline{a}$ when going under a binder in Rule ($\forall$R). Once the goal is stripped down to a simple class constraint $\mathcal{Q}$, Rule ($\mathcal{Q}$R) selects an axiom $\mathcal{C}$ whose head matches the goal, and uses it to replace the goal with a set of simpler constraints $\mathcal{A}$ (a process known as *context reduction* [44]). Goal matching is performed by judgment $\overline{a}; [\mathcal{C}] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta; \eta$, discussed below.

Matching Auxiliary judgment $\overline{a}; [\mathcal{C}] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta; \eta$ focuses on the axiom $\mathcal{C}$ and checks whether it matches the simple goal $\mathcal{Q}$. The main difference between this algorithmic relation and its declarative specification in Figure 8.3 lies in the type substitution θ. Instead of guessing a type for instantiating a polymorphic

$\boxed{\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_2; \eta}$ (Constraint Solving Algorithm)

$$\frac{\nexists \mathcal{C} \in \mathcal{A}_1 : \quad \overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A}_2 ; \eta}{\overline{a}; \mathcal{P} \models \mathcal{A}_1 \rightsquigarrow \mathcal{A}_1 ; \bullet}\ \textsc{Stop}$$

$$\frac{\overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A}_2 ; \eta_1 \qquad \overline{a}; \mathcal{P} \models \mathcal{A}_1, \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3; \eta_2}{\overline{a}; \mathcal{P} \models \mathcal{A}_1, \mathcal{C} \rightsquigarrow \mathcal{A}_3; (\eta_2 \cdot \eta_1)}\ \textsc{Step}$$

$\boxed{\overline{a}; \mathcal{P} \models [\mathcal{C}] \rightsquigarrow \mathcal{A} ; \eta}$ (Constraint Simplification)

$$\frac{\begin{array}{cc} \vdash_{\mathsf{ct}} C_1 \rightsquigarrow \upsilon_1 & \overline{a}; \mathcal{P},_{\mathrm{L}} (d_1 : C_1) \models [d_2 : C_2] \rightsquigarrow \overline{(d : C)} ; \eta \\ \overline{d}', d_1, d_2 \text{ fresh} & \eta' = [\lambda(d_1 : \upsilon_1).[\overline{d'\ d_1/d}](\eta(d_2))/d_0] \end{array}}{\overline{a}; \mathcal{P} \models [d_0 : C_1 \Rightarrow C_2] \rightsquigarrow \overline{(d' : C_1 \Rightarrow C)} ; \eta'}\ (\Rightarrow\mathrm{R})$$

$$\frac{\begin{array}{cc} & \overline{d}', d_C \text{ fresh} \\ \overline{a}, b; \mathcal{P} \models [d_C : C_0] \rightsquigarrow \overline{(d : C)} ; \eta & \eta' = [\Lambda b.[\overline{d'\ b/d}](\eta(d_C))/d_0] \end{array}}{\overline{a}; \mathcal{P} \models [d_0 : \forall b. C_0] \rightsquigarrow \overline{(d' : \forall b. C)} ; \eta'}\ (\forall\mathrm{R})$$

$$\frac{\mathcal{C} \in \mathcal{P} : \quad \overline{a}; [\mathcal{C}] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; \mathcal{P} \models [\mathcal{Q}] \rightsquigarrow \mathcal{A} ; \eta}\ (\mathcal{Q}\mathrm{R})$$

$\boxed{\overline{a}; [\mathcal{C}] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta ; \eta}$ (Constraint Matching)

$$\frac{d_1, d_2 \text{ fresh} \qquad \overline{a}; [d_2 : C_2] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; [d : C_1 \Rightarrow C_2] \models \mathcal{Q} \rightsquigarrow \mathcal{A},\ d_1 : \theta(C_1); \theta ; [d\ d_1/d_2] \cdot \eta}\ (\Rightarrow\mathrm{L})$$

$$\frac{d' \text{ fresh} \qquad \overline{a}; [d' : C] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta ; \eta}{\overline{a}; [d : \forall b. C] \models \mathcal{Q} \rightsquigarrow \mathcal{A}; \theta ; [d\ (\theta(b))/d'] \cdot \eta}\ (\forall\mathrm{L})$$

$$\frac{\theta = \mathit{unify}(\overline{a}; \tau_1 \sim \tau_2)}{\overline{a}; [d' : TC\ \tau_1] \models d : TC\ \tau_2 \rightsquigarrow \bullet; \theta ; [d'/d]}\ (\mathcal{Q}\mathrm{L})$$

Figure 8.6: Constraint Entailment with Dictionary Construction

$\boxed{\Gamma \vdash_{\mathtt{cls}} cls : \mathcal{A}_S ; \Gamma_c \rightsquigarrow fdata ; \overline{fval}}$ (Class Declaration Typing)

$$\frac{\begin{array}{c} cls = (\mathbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \}) \\ \Gamma, a \vdash_{\mathtt{ty}} \sigma \qquad \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon \qquad \Gamma, a \vdash_{\mathtt{ct}} C_i \\ \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad d, \overline{d}^n \text{ fresh} \qquad fdata = \mathbf{data}\ T_{TC}\ a = K_{TC}\ \overline{\upsilon}^n\ \upsilon \\ fval_1 = \mathbf{let}\ f : (\forall a. T_{TC}\ a \to \upsilon) = \Lambda a. \lambda (d : T_{TC}\ a). proj_{TC}^{n+1}(d) \\ fval_2^i = \mathbf{let}\ d_i : (\forall a. T_{TC}\ a \to \upsilon_i) = \Lambda a. \lambda (d : T_{TC}\ a). proj_{TC}^{i}(d) \end{array}}{\Gamma \vdash_{\mathtt{cls}} cls : [\overline{d_i : \forall a. TC\ a \Rightarrow C_i}^n]; [f : \forall a. TC\ a \Rightarrow \sigma] \rightsquigarrow fdata ; fval_1, \overline{fval_2}^n} \ \textsc{Class}$$

$\boxed{\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} inst : \mathcal{A}_I \rightsquigarrow fval}$ (Class Instance Typing)

$$\frac{\begin{array}{c} inst = (\mathbf{instance}\ (C_1, \ldots, C_n) \Rightarrow TC\ \tau\ \mathbf{where}\ \{\ f = e\ \}) \\ \mathbf{class}\ (C'_1, \ldots, C'_m) \Rightarrow TC\ a\ \mathbf{where}\ \{\ f :: \sigma\ \} \\ \bar{b} = fv(\tau) \qquad \overline{d}, \overline{d}', d_I \text{ fresh} \qquad \mathcal{P}_I = \mathcal{P},_{\mathrm{L}} \overline{d : C} \qquad \Gamma_I = \Gamma, \bar{b} \\ \Gamma_I \vdash_{\mathtt{ct}} C_i \qquad \bar{b}; \mathcal{P}_I,_{\mathrm{L}} (d_I : \forall \bar{b}. \overline{C}^n \Rightarrow TC\ \tau); \Gamma_I \vdash_{\mathtt{tm}} e : [\tau / a]\sigma \rightsquigarrow t \\ \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i \qquad \bar{b}; \mathcal{P}_I \models \overline{d' : [\tau/a] C' \rightsquigarrow \bullet ; \eta} \\ fval = \mathbf{let}\ d_I : (\forall \bar{b}. \overline{\upsilon} \to T_{TC}\ \tau) = \Lambda \bar{b}. \lambda \overline{(d : \upsilon)}. K_{TC}\ \tau\ \overline{\eta(d')}\ t \end{array}}{\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} inst : [d_I : \forall \bar{b}. \overline{C} \Rightarrow TC\ \tau] \rightsquigarrow fval} \ \textsc{Instance}$$

Figure 8.7: Declaration Elaboration

axiom in Rule (∀L) (top-down), we defer the choice until the head of the axiom is met, in Rule (QL) (bottom-up). Observe that Rule (∀L) does not record b as untouchable, effectively turning it into a unification variable. Thus, by unifying the head of the axiom with the goal we can determine without guessing an instantiation for all top-level quantifiers, captured by the type substitution θ.

As an example, consider the derivation of one-step simplification of $\forall b. Eq\ b \Rightarrow Eq\ [b]$, when $(\forall a. Eq\ a \Rightarrow Eq\ [a]) \in \mathcal{P}$:[5]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{unify(b; a \sim b) = \theta = [b/a]}{b; [Eq\ [a]] \models Eq\ [b] \rightsquigarrow \bullet ; \theta}\ (Q\mathrm{L})}{b; [Eq\ a \Rightarrow Eq\ [a]] \models Eq\ [b] \rightsquigarrow Eq\ b; \theta}\ (\Rightarrow\mathrm{L})}{b; [\forall a. Eq\ a \Rightarrow Eq\ [a]] \models Eq\ [b] \rightsquigarrow Eq\ b; \theta}\ (\forall\mathrm{L})}{b; \mathcal{P}, Eq\ b \models [Eq\ [b]] \rightsquigarrow Eq\ b}\ (Q\mathrm{R})}{b; \mathcal{P} \models [Eq\ b \Rightarrow Eq\ [b]] \rightsquigarrow (Eq\ b \Rightarrow Eq\ b)}\ (\Rightarrow\mathrm{R})}{\bullet; \mathcal{P} \models [\forall b. Eq\ b \Rightarrow Eq\ [b]] \rightsquigarrow (\forall b. Eq\ b \Rightarrow Eq\ b)}\ (\forall\mathrm{R})$$

[5] We omit the evidence substitutions for brevity.

Search As Section 8.3.4 has remarked, there may be multiple matching axioms, e.g., due to overlapping superclass axioms. The straightforward algorithmic approach to the involved nondeterminism is search, possibly implemented by backtracking. The GHC Haskell implementation can employ a heuristic to keep this search shallow. It does so by using the superclass constraints very selectively: whenever a new local constraint is added to the theory, it pro-actively derives all its superclasses and adds them as additional local axioms. When looking for a match, it does not consider the superclass axioms and prefers the local axioms over the instance axioms. If a matching local axiom exists, it immediately discharges the entire goal without further recursive resolution. This is the case because in regular Haskell local axioms are always simple class constraints Q.

In our setting, we can also implement a (modified version) of GHC's heuristic, but this does not obviate the need for deep search. The reason is that our local axioms are not necessarily simple axioms, and matching against them may leave residual goals that require further recursive resolution. When that recursive resolution gets stuck, we have to backtrack over the choice of axiom. Consider the following example.

$$\textbf{class}\ (E\ a \Rightarrow C\ a) \Rightarrow D\ a$$
$$\textbf{class}\ (G\ a \Rightarrow C\ a) \Rightarrow F\ a$$

Given local axioms $D\ a$, $F\ a$ and $G\ a$, consider what happens when we resolve the goal $C\ a$. The superclasses $E\ a \Rightarrow C\ a$ and $G\ a \Rightarrow C\ a$ of respectively $D\ a$ and $F\ a$ both match this goal. If we pick the first one, we get stuck when recursively resolving $E\ a$. However, if we backtrack and consider the second one instead, we can recursively resolve $G\ a$ against the given local constraint.

In summary, because we do not see a general way to avoid search, our prototype implementation uses backtracking for choosing between the different axioms.[6]

Implementation Our prototype implementation is available at `https://github.com/gkaracha/quantcs-impl`. It incorporates higher-kinded datatypes and performs type inference, elaboration into System F (as explained in the next section), and type checking of the generated code.

The examples we have tested with the prototype provide confidence that our system is sound and that the elaboration is type preserving. The formal proof of the metatheory is future work.

[6] It is worth mentioning that the rules of Figure 8.6 conservatively extend standard Haskell resolution, both in terms of expressivity and performance.

$$\boxed{\overline{a}; \mathcal{P}; \Gamma \vdash_{\texttt{tm}} e : \sigma \rightsquigarrow t} \qquad \text{(Explicitly Annotated Term Typing)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\texttt{tm}} e : \tau_1 \rightsquigarrow t \mid \mathcal{A}_e; E_e \qquad \overline{d} \text{ fresh} \qquad \theta = \mathit{unify}(\overline{a}, \overline{b}; E_e, \tau_1 \sim \tau_2) \\ \vdash_{\texttt{ct}} C_i \rightsquigarrow \upsilon_i \qquad \overline{a}, \overline{b}; \mathcal{P}, _{\mathrm{L}} \overline{d : C} \models \theta(\mathcal{A}_e) \rightsquigarrow \bullet; \eta \end{array}}{\overline{a}; \mathcal{P}; \Gamma \vdash_{\texttt{tm}} e : (\forall \overline{b}. \overline{C} \Rightarrow \tau_2) \rightsquigarrow \Lambda \overline{b}. \lambda \overline{(d : \upsilon)}. \eta(\theta(t))} \; (\preceq)$$

Figure 8.8: Subsumption Rule

8.4.4 Checking Declarations

Figure 8.7 defines type checking of class and instance declarations.

Class Declaration Typing Typing for class declarations is given by Rule Class. For the purposes of type inference, Rule Class is identical to the corresponding rule of Figure 8.2, so we defer its analysis to Section 8.5.5 which discusses elaboration.

Instance Declaration Typing Typing for instance declarations takes the form $\mathcal{P}; \Gamma \vdash_{\texttt{inst}} \mathit{inst} : \mathcal{A}_I \rightsquigarrow \mathit{fval}$ and is given by Rule Instance. For the most part it is identical to the corresponding rule of Figure 8.2.

The most notable difference is the handling of the method implementation e: method implementations have their type imposed by the method signature in the class declaration. Hence, we need to *check* rather than *infer* their type.

This operation is expressed succinctly by relation $\overline{a}; \mathcal{P}; \Gamma \vdash_{\texttt{tm}} e : \sigma \rightsquigarrow t$, presented in Figure 8.8. Essentially, it ensures that the inferred type for e subsumes the expected type σ. A type σ_1 is said to subsume type σ_2 if any expression that can be assigned type σ_1 can also be assigned type σ_2.

Rule $(\preceq)$ performs type inference and type subsumption checking simultaneously: First, it infers a monotype τ_1 for expression e, as well as wanted constraints $\mathcal{A}_e$ and type equalities E_e. Type equalities E_e should have a unifier and the inferred type τ_1 should also be unifiable with the expected type τ_2. Finally, the given constraints $\overline{C}$ should completely entail the wanted constraints $\mathcal{A}_e$.

$$
\begin{array}{lcll}
\mathit{fpgm} & ::= & t \mid \mathit{fval}; \mathit{fpgm} \mid \mathit{fdata}; \mathit{fpgm} & \textit{program} \\
\mathit{fval} & ::= & \mathbf{let}\ x : \upsilon = t & \textit{value binding} \\
\mathit{fdata} & ::= & \mathbf{data}\ T\ a = K\ \overline{\upsilon} & \textit{datatype} \\
 & & & \\
t & ::= & x \mid K \mid \lambda(x : \upsilon).t \mid t_1\ t_2 \mid \Lambda a.t \mid t\ \upsilon & \textit{term} \\
 & \mid & \mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_2 \mid \mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2 & \\
 & & & \\
\upsilon & ::= & a \mid \upsilon_1 \to \upsilon_2 \mid \forall a.\upsilon \mid T\ \upsilon & \textit{type}
\end{array}
$$

Figure 8.9: System F Syntax

8.4.5 Program Typing

Type inference and elaboration for programs is straightforward and can be found in Appendix A.4.

8.5 Translation to System F

This section discusses the elaboration aspect of the algorithm presented in Section 8.4.

8.5.1 Target Language: System F

Syntax The syntax of System F [30] – extended with data types and recursive let-bindings – is presented in Figure 8.9 and is entirely standard. Like in the source language, we elide all mention of kinds. Without loss of generality, we simplify matters by allowing only data types with a single type parameter and a single data constructor and case expressions with a single branch; this is sufficient for our dictionary-passing translation of type classes.

Semantics & Typing Since the operational semantics and typing for System F with data types are entirely standard and do not contribute to the novelty of this chapter, we omit them from our main presentation. They can be found in Appendix A.9.

8.5.2 Elaboration of Types & Constraints

Our system follows the traditional approach of translating source type class constraints into explicitly-passed System F terms, the so-called *dictionaries* [32, 95]. This transition is reflected in the translation of types, performed by judgment $\vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon$:

$$\frac{}{\vdash_{\mathsf{ty}} a \rightsquigarrow a}\ \textsc{TyVar} \qquad \frac{\vdash_{\mathsf{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathsf{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\vdash_{\mathsf{ty}} \tau_1 \rightarrow \tau_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ \textsc{TyArr}$$

$$\frac{\vdash_{\mathsf{ct}} C \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathsf{ty}} \rho \rightsquigarrow \upsilon_2}{\vdash_{\mathsf{ty}} C \Rightarrow \rho \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ \textsc{TyQual} \qquad \frac{\vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon}{\vdash_{\mathsf{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon}\ \textsc{TyAll}$$

Rules TyVar, TyArr and TyAll are straightforward. Rule TyQual elaborates a qualified type into a System F arrow type: the constraint C is translated into the dictionary type υ_1, via relation $\vdash_{\mathsf{ct}} C \rightsquigarrow \upsilon$ which performs elaboration of constraints:

$$\frac{\vdash_{\mathsf{ty}} \tau \rightsquigarrow \upsilon}{\vdash_{\mathsf{ct}} TC\ \tau \rightsquigarrow T_{TC}\ \upsilon}\ (\mathrm{C}Q) \qquad \frac{\vdash_{\mathsf{ct}} C \rightsquigarrow \upsilon}{\vdash_{\mathsf{ct}} \forall a.C \rightsquigarrow \forall a.\upsilon}\ (\mathrm{C}\forall)$$

$$\frac{\vdash_{\mathsf{ct}} C_1 \rightsquigarrow \upsilon_1 \qquad \vdash_{\mathsf{ct}} C_2 \rightsquigarrow \upsilon_2}{\vdash_{\mathsf{ct}} C_1 \Rightarrow C_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ (\mathrm{C}{\Rightarrow})$$

Rule $(\mathrm{C}Q)$ elaborates a class constraint $TC\ \tau$ into a type constructor application $T_{TC}\ \upsilon$, which corresponds to the type of dictionaries that witness $TC\ \tau$. Rule $(\mathrm{C}\forall)$ is straightforward. Rule $(\mathrm{C}{\Rightarrow})$ elaborates implication constraints of the form $C_1 \Rightarrow C_2$ into System F arrow types $\upsilon_1 \rightarrow \upsilon_2$, that is, types of *dictionary transformers*. As a concrete example, the constraint corresponding to the *Show* instance for type *HPerf* (Section 8.2.2):

$$\forall\ f\ a.\ Show\ a \Rightarrow (\forall\ x.\ Show\ x \Rightarrow Show\ (f\ x)) \Rightarrow Show\ (HPerf\ f\ a)$$

is elaborated into the type

$$\forall\ f\ a.\ TShow\ a \rightarrow (\forall\ x.\ TShow\ x \rightarrow TShow\ (f\ x)) \rightarrow TShow\ (HPerf\ f\ a)$$

8.5.3 Elaboration of Terms

Term elaboration is straightforward. Rule TmVar handles term variables. The instantiation of the type scheme $\forall \overline{a}.\overline{C} \Rightarrow \tau$ to $[\overline{b}/\overline{a}]\tau$ becomes explicit in the System F representation, by the application of x to type variables $\overline{b}$, as well as the fresh dictionary variables $\overline{d}$, corresponding one-to-one to the implicit constraints $\overline{C}$. Rule TmAbs elaborates λ-abstractions. Since in System F all bindings are explicitly typed, in the elaborated term we annotate the binding of x with its type a. Similarly, Rule TmLet elaborates let bindings, again explicitly annotating x with its type υ_1 in the elaborated term. Rule TmApp is straightforward.

8.5.4 Dictionary Construction

The entailment algorithm of Figure 8.6 constructs explicit witness proofs (in the form of dictionary substitutions) while entailing a constraint.

Simplification The evidence substitution η in the simplification relation shows how to construct a witness for the wanted constraint $\mathcal{C}$ from the simpler constraints $\mathcal{A}'$ and program theory $\mathcal{P}$.

The goal of Rule ($\Rightarrow$R) is to build an evidence substitution η', which constructs a proof for $(d_0 : C_1 \Rightarrow C_2)$ from the proofs $\overline{d}'$ for the simpler constraints $\overline{C_1 \Rightarrow C}$. It is instructive to consider the generated evidence substitution in parts, also taking the types into account:

1. η illustrates how to generate a proof for $(d_2 : C_2)$, from the local assumption $(d_1 : C_1)$ and local residual constraints $(\overline{d : C})$.

2. $[\overline{d'\ d_1/d}]$ generates proofs for the (local) residual constraints $(\overline{d : C})$, by applying the residual constraints $(\overline{d' : C_1 \Rightarrow C})$ to the local assumption $(d_1 : C_1)$.

3. $([\overline{d'\ d_1/d}] \cdot \eta)(d_2)$ is a proof for C_2, under assumptions $(d_1 : C_1)$ and $(\overline{d' : C_1 \Rightarrow C})$.

4. Finally, we construct the proof for $(d_0 : C_1 \Rightarrow C_2)$ by explicitly abstracting over d_1: $\lambda(d_1 : \upsilon_1).[\overline{d'\ d_1/d}](\eta(d_2))$

Rule ($\forall$R) proceeds similarly. Finally, Rule (QR) generates the evidence substitution via constraint matching, which we discuss next.

Matching Similarly, the evidence substitution η in the matching relation shows how to construct a witness for the wanted constraint $\mathcal{Q}$ from the simpler constraints $\mathcal{A}$ and program theory $\mathcal{P}$.

Rule ($\Rightarrow$L) generates two fresh dictionary variables, d_1 for the residual constraint $\theta(C_1)$, and d_2 for the local assumption C_2. Finally, dictionary d_2 is replaced by the application of the dictionary transformer d to the residual dictionary d_1. Rule ($\forall$L) behaves similarly. The instantiation of the axiom d becomes explicit, by applying it to the chosen type $\theta(b)$. Finally, Rule (QL) is straightforward: since the wanted and the given constraints are identical (given that they unify), the wanted dictionary d is replaced by the given d'.

8.5.5 Declaration Elaboration

Figure 8.7 presents the elaboration of both class and instance declarations into System F.

Elaboration of Class Declarations A declaration for a class TC is encoded in System F as a dictionary type T_{TC}, with a single data constructor K_{TC} and $n+1$ arguments: n arguments for the superclass dictionaries (of type $\overline{v}^n$) and one more for the method implementation (of type v). For example, the *Trans* declaration of Section 8.2.1 gives rise to the following dictionary type:

$$\begin{aligned}\textbf{data}\ T_{Trans}\ t = K_{Trans}\ &(\forall m.T_{Monad}\ m \to T_{Monad}\ (t\ m))\\ &(\forall m\ a.T_{Monad}\ m \to m\ a \to (t\ m)\ a)\end{aligned}$$

Accordingly, we generate $n+1$ projection functions that extract each of the arguments (d_i extracts the i-th superclass dictionary and f the method implementation). We use $proj^i_{TC}(d)$ to denote pattern matching against d and extracting the i-th argument:

$$proj^i_{TC}(d) \equiv \textbf{case}\ d\ \textbf{of}\ K_{TC}\ \overline{x}^k \to x_i \qquad ,\overline{x}^k\ \text{fresh}$$

where k denotes the arity of data constructor K_{TC}. E.g., the superclass projection function for class *Trans* takes the form:

$$\begin{aligned}&d_{sc} : \forall t.T_{Trans}\ t \to (\forall m.T_{Monad}\ m \to T_{Monad}\ (t\ m))\\ &d_{sc} = \Lambda t.\lambda(d : T_{Trans}\ t).\ \textbf{case}\ d\ \textbf{of}\ \{\ K_{Trans}\ d'\ \_ \to d'\ \}\end{aligned}$$

Elaboration of Class Instances A class instance is elaborated into a System F dictionary transformer d_I:

$$\textbf{let}\ d_I : (\forall\overline{b}.\overline{v} \to T_{TC}\ \tau) = \Lambda\overline{b}.\lambda\overline{(d : v)}.K_{TC}\ \tau\ \overline{\eta(d')}\ t$$

Given dictionaries $\overline{d}$ – corresponding to the given context constraints – we need to provide all arguments of the data constructor K_{TC}: (a) the instantiation of the class type parameter, (b) the superclass dictionaries, and (c) the method implementation. The first argument is trivial. We obtain the superclass dictionaries by applying the evidence substitution η on the dictionary variables $\overline{d}'$ that abstract over the required superclass constraints. The method implementation t is elaborated via premise

$$\bar{b}; \mathcal{P}_I; \Gamma_I \vdash_{\mathtt{tm}} e : [\tau/a]\sigma \rightsquigarrow t$$

which elaborates type subsumption in a similar manner.

8.6 Termination of Resolution

Termination of resolution is the cornerstone of the overall termination of type inference. This section discusses how to enforce termination by means of syntactic conditions on the axioms. These conditions are adapted from those of Cochis [83] and generalize the earlier conditions for Haskell by Sulzmann et al. [91].

Overall Strategy We show termination by characterising the resolution process as a (resolution) tree with goals in the nodes and axioms on the (multi-)edges. The initial goal sits at the root of the tree. A multi-edge from a parent node to its children presents an axiom that matches the parent node's goal and its children are the residual goals. Resolution terminates iff the tree is finite. Hence, if it does not terminate, there is an infinite path from the root in the tree, that denotes an infinite sequence of axiom applications.

To show that there cannot be such an infinite path, we use a norm $\|\cdot\|$ that maps the head [7] of every goal C to a natural number, its size. The size of a class constraint $TC\ \tau$ is the size of its type parameter τ, which is given by the following equations:

$$\begin{array}{lcl} \|a\| & = & 1 \\ \|\tau_1 \rightarrow \tau_2\| & = & 1 + \|\tau_1\| + \|\tau_2\| \end{array}$$

If we can show that this size strictly decreases from any parent goal to its children, then we know that, because the order on the natural numbers is well-founded, on any path from the root there is eventually a goal that has no children.

Termination Condition It is trivial to show that the size strictly decreases, if we require that every axiom makes it so. This requirement is formalised as the

[7] The head of a constraint is defined as: $head(Q) = Q$; $head(\forall a.C) = head(C)$; and $head(C_1 \Rightarrow C_2) = head(C_2)$.

termination condition of axioms $\mathit{term}(C)$:

$$\frac{}{\mathit{term}(Q)}\ (Q\mathrm{T}) \qquad \frac{\mathit{term}(C)}{\mathit{term}(\forall a.C)}\ (\forall\mathrm{T})$$

$$\frac{\begin{array}{cccc}\mathit{term}(C_1) & \mathit{term}(C_2) & Q_1 = \mathit{head}(C_1) & Q_2 = \mathit{head}(C_2) \\ \|Q_1\| < \|Q_2\| & \multicolumn{3}{c}{\forall a \in \mathit{fv}(C_1) \cup \mathit{fv}(C_2) : \quad \mathit{occ}_a(Q_1) \leqslant \mathit{occ}_a(Q_2)}\end{array}}{\mathit{term}(C_1 \Rightarrow C_2)}\ (\Rightarrow\mathrm{T})$$

Rule ($\Rightarrow$T) for $C_1 \Rightarrow C_2$ enforces the main condition, that the size of the residual constraint's head Q_1 is strictly smaller than the head Q_2 of C_2. In addition, the rule ensures that this property is stable under type substitution. Consider for instance the axiom $\forall a.C\ (a \to a) \Rightarrow C\ (a \to \mathit{Int} \to \mathit{Int})$. The head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate a to $(\mathit{Int} \to \mathit{Int} \to \mathit{Int})$, then the head's size becomes 10 while the context constraint's size becomes 11. Declaratively, we can formulate stability as:

$$\forall \theta.\mathit{dom}(\theta) \subseteq \mathit{fv}(C_1) \cup \mathit{fv}(C_2) \Rightarrow \|\theta(Q_1)\| < \|\theta(Q_2)\|$$

The rule uses instead an equivalent algorithmic formulation which states that the number of occurrences of any free type variable a may not be larger in Q_1 than in Q_2. Here the number of occurrences of a type variable a in a class constraint $TC\ \tau$ (denoted as $\mathit{occ}_a(TC\ \tau)$) is the same as the number of free occurrences of a in the parameter τ, where function $\mathit{occ}_a(\tau)$ is defined as:

$$\begin{array}{lcl}\mathit{occ}_a(b) & = & \left\{\begin{array}{ll}1 & \text{, if } a = b \\ 0 & \text{, if } a \neq b\end{array}\right. \\ \mathit{occ}_a(\tau_1 \to \tau_2) & = & \mathit{occ}_a(\tau_1) + \mathit{occ}_a(\tau_2)\end{array}$$

Finally, as the constraints have a recursive structure whereby their components are themselves used as axioms, the rules also enforce the termination condition recursively on the components.

Superclass Condition If we could impose the termination condition above on all axioms in the theory P, we would be set. Unfortunately, this condition is too strong for the superclass axioms. Consider the superclass axiom $\forall\ a.\ \mathit{Ord}\ a \Rightarrow \mathit{Eq}\ a$ of the standard Haskell'98 *Ord* type class. Here both *Ord*} *a* and *Eq*} *a* have size 1; in other words, the size does not strictly decrease and so the axiom does not satisfy the termination condition.

To accommodate this and other examples, we impose an alternative condition for superclass axioms. This superclass condition relaxes the strict size decrease to a non-strict size decrease and makes up for it by requiring that the superclass relation forms a *directed acyclic graph* (DAG). The superclass relation is defined as follows on type classes.

Definition 12 (Superclass Relation)**.** *Given a class declaration*

$$\textbf{class}\ (C_1, \ldots, C_n) \Rightarrow TC\ a\ \textbf{where}\ \{\ f :: \sigma\ \}$$

each type class TC_i *is a superclass of* TC*, where* $head(C_i) = TC_i\ \tau_i$.

Observe that the DAG induces a well-founded partial order on type classes. Hence, on any path in the resolution tree, any uninterrupted sequence of superclass axiom applications has to be finite. For the length of such a sequence, the size of the goal does not increase (but might not decrease either). Yet, after a finite number of steps the sequence has to come to an end. If the path still goes on at that point, it must be due to the application of an instance or local axiom, which strictly decreases the goal size. Hence, overall we have preserved the variant that the goal size decreases after a bounded number[8] of steps.

Termination & Soundness Finally, although we have not proven it formally yet, we are confident that soundness of type inference and preservation of typing under elaboration hold independently of termination (and thus are not affected by whether the termination conditions are met). Such a property is crucial for integrating our algorithm within GHC in the future, where flags such as *UndecidableInstances* are heavily used.

8.7 Related Work

This section discusses related work, focusing mostly on comparing our approach with existing encodings/workarounds in Haskell. The history of quantified class constraints and their demand in previous research was already discussed in Section 8.1.

The Coq Proof Assistant Coq provides very flexible support for type classes [86] and allows for arbitrary formulas in class and instance contexts – actually the contexts are just parameters. For instance, we can model the *Trans* class as:

```
Class Trans (T : (Type → Type) → Type → Type)
  ‘{∀ M, ‘{Monad M} → Monad (T M)} :=
  {lift : ∀ A M, ‘{Monad M} → M A → (T M) A}.
```

[8] bounded by the height of the superclass DAG

The downside of Coq's flexibility is that resolution can be ambiguous and non-terminating. The accepted workaround is for the programmer to perform resolution manually when necessary. This is acceptable in the context of Coq's interactive approach to proving, but would mean a great departure from Haskell's non-interactive type inference.

Trifonov's Workaround and Monatron Trifonov [93] gives an encoding of quantified class constraints in terms of regular class constraints. The encoding introduces a new type class that encapsulates the quantified constraint, e.g. *Monad_t t* for ∀ *m*. *Monad m* ⇒ *Monad* (*t m*), and that provides the implied methods under a new name. This expresses the *Trans* problem as follows:

```
class Monad_t t where
  treturn :: Monad m ⇒ a → t m a
  tbind :: Monad m ⇒ t m a → (a → t m b) → t m b
class Monad_t t ⇒ Trans t where
  lift :: Monad m ⇒ m a → t m a
```

While this approach captures the intention of the quantified constraint, it does not enable the type checker to see that *Monad* (*t m*) holds for any transformer *t* and monad *m*. While the monad methods are available for *t m*, they do not have the usual name.

For this reason, Trifonov presents a further (non-Haskell'98) refinement of the encoding, which was adopted by the Monatron [39] library[9] among others. A non-essential difference is that Monatron merges the above *Monad t* and *Trans* into a single class:

```
class MonadT t where
  lift :: Monad m ⇒ m a → t m a
  treturn :: Monad m ⇒ a → t m a
  tbind :: Monad m ⇒ t m a → (a → t m b) → t m b
```

The key novelty is that it also makes the methods *treturn* and *tbind* available under their usual name with a single *Monad* instance for all monad transformers.

```
instance (Monad m, MonadT t) ⇒ Monad (t m) where
  return = treturn
  (≫=) = tbind
```

With these definitions the monad transformer composition does type check. Unfortunately, the head of the *Monad* (*t m*) instance is highly generic and easily overlaps with other instances.

[9] For the implementation see `https://hackage.haskell.org/package/Monatron`

The MonadZipper Because they found Monatron's overlapping instances untenable, Schrijvers and Oliveira [82] presented a different workaround for this problem in the context of their monad zipper datatype, which is an extended form of transformer composition. Their solution adds a method *mw* to the *Trans* type class:

```
class Trans t where
  lift :: Monad m ⇒ m a → t m a
  mw :: Monad m ⇒ MonadWitness t m
```

For any monad *m* this method returns a GADT [71] witness for the fact that *t m* is a monad. This is possible because with GADTs, type class instances can be stored in the data constructors.

```
data MonadWitness (t :: (⋆ → ⋆) → (⋆ → ⋆)) m where
  MW :: Monad (t m) ⇒ MonadWitness t m
```

By pattern matching on the witness of the appropriate type the programmer can bring the required *Monad* (*t2 m*) constraint into scope to satisfy the type checker.

```
instance (Trans t1, Trans t2) ⇒ Trans (t1 ⋆ t2) where
  lift :: ∀ m a. Monad m ⇒ m a → (t1 ⋆ t2) m a
  lift = case (mw :: MonadWitness t2 m) of
    MW → C ∘ lift ∘ lift
  mw = ...
```

The downside of this approach is that it offloads part of the type checker's work on the programmer. As a consequence, the code becomes cluttered with witness manipulation.

The constraint Library Kmett's constraint library [49] provides generic infrastructure for reifying quantified constraints in terms of GADTs, generalizing the MonadZipper solution above. Additionally, it complements the encoding with ample utilities for the manipulation of such constraints. Unfortunately, it suffers from the same problem: passing, construction and deconstruction of dictionaries needs to be manually performed by the programmer.

Corecursive Resolution Fu et al. [27] address the divergence problem that arises for generic nested datatypes. They turn the diverging resolution with user-supplied instances into a terminating resolution in terms of automatically derived instances. These auxiliary instances are derived specifically to deal with the query at hand; they shift the pattern of divergence to the term-level in the form of co-recursively defined dictionaries. The authors do point out that the class of divergent cases they support is limited and that deriving quantified instances would be beneficial.

Cochis The calculus of coherent implicits, COCHIS [83], and its focusing-based resolution in particular, have been a major inspiration of this work. Just like this work, COCHIS supports recursive resolution of quantified constraints. Yet, there are a number of significant differences. Firstly, COCHIS does not feature a separate syntactic sort for type classes, but implicitly resolves regular terms in the Scala tradition. As a consequence, it does not distinguish between instance and superclass axioms, e.g., for the sake of enforcing termination and coherence. Perhaps more significantly, COCHIS features local "instances" as opposed to our globally scoped instances. Local instances may overlap with one another and coherence is obtained by prioritizing those instances that are introduced in the innermost scope. This way COCHIS's resolution is entirely deterministic, while ours is non- deterministic (yet coherent) due to overlapping local and superclass axioms.

Quantified class constraints have been a recurring feature request for Haskell in the literature. This chapter gets things rolling by formally studying quantified class constraints in a simplified Haskell-like calculus. However, there is still plenty of work to be done to integrate quantified class constraints in a full blown Haskell compiler such as GHC. Therefore, we see ample opportunities for extending our system in the near future.

8.8 Quantified Constraints in GHC

Quantified class constraints have been introduced in GHC 8.6.1, using the *QuantifiedContraints* language pragma. Despite being a relatively new language extension, the pragma is currently—at the time of writing—being used in 170 Haskell packages, which make up about 1% of the Hackage library.

While the GHC implementation is based on this work, constraint entailment is handled differently. While this work employs backtracking for selecting and matching against axioms, the GHC community decided against this approach, as they were worried about a possible performance penalty. In fact, multiple alternatives exist, each with their own issues: (1) When multiple overlapping axioms exist, the compiler could simply select the first with a matching head. However, confusingly, the ordering of the instance context would now have an impact on whether or not the program is accepted. (2) Heuristics could be employed to determine which axioms to prefer over others. However, this would add significant complexity to the compiler, and could potentially be confusing during debugging. (3) The compiler could detect overlapping axioms, and conservatively reject the program. In the end, GHC uses a very simple heuristic which always prefers simple axioms over quantified ones, and prefers local constraints over global instances. When no simple axioms matches, and the compiler detects overlapping matching heads, it will conservatively reject the program. While this approach works in most common cases, it is no full substitute for backtracking, as discussed in Section 8.4.3.

8.9 Scientific Output

This chapter has motivated the need for, and presented a fully fledged design of quantified class constraints.

The material found in this chapter is largely taken from the following publication:

> Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017). Association for Computing Machinery, New York, NY, USA, 148–161. DOI:https://doi.org/10.1145/3122955.3122967

This work was originally performed in the context of the master thesis of the author of this text, under the mentoring of Georgios Karachalias and supervision of Tom Schrijvers. The contributions of the different authors are as follows:

- Both the calculus used in this work and the type inference algorithm are a joint effort between the author of this thesis, Georgios Karachalias and Tom Schrijvers.
- The compiler prototype implementation was orginally developed by the author of this thesis, whereas the latest version is a revision by Georgios Karachalias.

Chapter 9

Meta Theory: Coherence for Quantified Constraints

9.1 Introduction

Chapter 8 increased the expressivity of Haskell's type classes by introducing the quantified class constraints language extension. Note, however, that doing so requires a significant update to the type class resolution mechanism. This chapter answers the following research question:

> How to prove coherence of type class resolution in the presence of quantified class constraints?

Doing so showcases the adaptability of the coherence proof described in Chapter 7 by extending the proof with quantified class constraints.

Section 9.2 describes an alternative formulation of $\lambda_{\mathbf{TC}}^{\Rightarrow}$, as well as a new intermediate language $F_{\mathbf{D}}^{\Rightarrow}$. Updates to the core theorems of the Chapter 7 proof are presented in Section 9.3. Finally, Section 9.4 discusses the required changes to the logical relations and coherence proof itself. The full proof can be found in Appendix C.

9.2 Calculus Updates

This section discusses the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ source and $F_{\mathbf{D}}^{\Rightarrow}$ intermediate language, used in the updated proof of coherence with support for quantified class constraints. We will emphasise both the updates in the calculi with respect to the $\lambda_{\mathbf{TC}}$ and $F_{\mathbf{D}}$ languages from Chapter 7, as well as the changes with respect to the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ language from Chapter 8.

We start off with a discription of the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ calculus and its differences with $\lambda_{\mathbf{TC}}$ from Section 9.2.1. Note, however, that while Chapter 8 describes a full type inference algorithm for $\lambda_{\mathbf{TC}}^{\Rightarrow}$, this section introduces an alternative formulation of $\lambda_{\mathbf{TC}}^{\Rightarrow}$ with a bidirectional type system. We make a second deviation from the calculus as described in Chapter 8 by disallowing recursive expressions. We thus follow the example set by Chapter 7, as both type inference and recursion are orthogonal to type class resolution. These two simplifications thus allow us to focus on the coherence property, without getting lost in technical details regarding type inference. Section 9.2.2 illustrates the new bidirectional type system with an example derivation.

As the intermediate language $F_{\mathbf{D}}$ features explicit dictionaries, it too needs to be adapted to support quantified constraints. This includes support for the extended constraints themselves, and a more expressive form of dictionaries, with their own operational semantics. Section 9.2.3 describes these changes to the $F_{\mathbf{D}}^{\Rightarrow}$ calculus. Section 9.2.4 illustrates the new intermediate language with and example derivation, translating from $\lambda_{\mathbf{TC}}^{\Rightarrow}$. Note that these advanced dictionaries are still translated into standard System F (as described in Section 8.5). The target language $F_{\{\}}$ can thus be left unchanged.

9.2.1 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Updates

This section presents $\lambda_{\mathbf{TC}}^{\Rightarrow}$, our source calculus used in the extended version of our proof of coherence. We discuss both the difference between $\lambda_{\mathbf{TC}}^{\Rightarrow}$ and $\lambda_{\mathbf{TC}}$, as well as the changes compared to $\lambda_{\mathbf{TC}}^{\Rightarrow}$ as presented in Chapter 8.

Concretely, compared to $\lambda_{\mathbf{TC}}$, the new calculus is extended with (1) the more expressive form of constraints C from Chapter 8, (2) support for constraints C in class and instance declarations, and (3) abstraction over constraints C in qualified types ρ.

Compared to the Chapter 8 $\lambda_{\mathbf{TC}}^{\Rightarrow}$, we now include a bidirectional type system for $\lambda_{\mathbf{TC}}^{\Rightarrow}$. This system requires a limited number of type annotations, as shown in the grammar extension in Figure 9.1.

$$e \quad ::= \quad \textbf{let}\ \ x : \sigma = e_1\ \textbf{in}\ \ e_2 \mid e :: \tau \mid \dots \qquad\qquad \textit{Expression}$$

Figure 9.1: Updated Grammar for $\lambda^{\Rightarrow}_{\mathbf{TC}}$ with Type Annotations, Extension of Figure 8.1

We provide a high-level overview of the different updated typing relations. The type and constraint well-formedness relations are unsurprising and can be found in Appendix A.5.2. The expression and program typing relations have received a number of largely minor updates, which can be found in Appendix A.5.2. The most interesting judgement in this context is the constraint resolution relation, as presented in Figure 9.2. Following Chapter 8, we employ a focusing approaching for solving type class constraints. We thus split resolution in constraint entailment and constraint matching. Constraint entailment $P; \Gamma_C; \Gamma \vDash [C] \rightsquigarrow e$ succeeds if the constraint C can be proven from the axiom set P, producing the proof e in the process. Note that this is slightly different from constraint simplification in Chapter 8, as we solve any residual constraints right away. Rules sEntailT-arrow and sEntailT-forall correspond to rules $\Rightarrow R$ and $\forall R$ from Chapter 8, with the exception of the more straightforward translation. Rules sEntailT-inst and sEntailT-local resolve a class constraint Q by finding a matching instance declaration and local axiom, respectively. Note that these rules make the constraint resolution process non-syntax-directed.

Constraint matching $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_1$ succeeds if the axiom C can be used to prove class constraint Q. The relation is quite involved, as it needs to deconstruct the C axiom, while keeping track of all the aspects of this axiom that still need to be handled. Concretely, it tracks the type variables $\overline{a}$ that have yet to be instantiated, and the residual constraints $\overline{C}$ that still need to be proven, along with their local names $\overline{\delta}$. Finally, it takes a proof e_0 for C, and constructs the proof e_1 for constraint Q, along with an instantiation $\overline{\tau}$ for $\overline{a}$. Rule sMatchT-arrow attempts to prove Q from $C_1 \Rightarrow C_2$, by first assuming evidence δ_1 for C_1, and applying this to the given proof for $C_1 \Rightarrow C_2$ to obtain evidence for C_2. It then recursively attempts to prove Q from this new proof for C_2. If this works out, we get back a substitution $\overline{\tau}$ for the variables $\overline{a}$. We substitute, and construct a new proof e_1 for $[\overline{\tau}/\overline{a}]C_1$. Finally, we substitute this result for our temporary proof δ_1 in e_2, to produce our result for Q. Rule sMatchT-forall attempts to prove Q from $\forall a.C$, by marking the variable a for instantiation. The actual instantiation of these variables happens in Rule sMatchT-classconstr, where a unifying substitution is constructed to prove $TC\,\tau_1$ from $TC\,\tau_0$. This substitution is then propagated back, to be applied to the residual constraints.

$\boxed{P; \Gamma_C; \Gamma \vDash [C] \rightsquigarrow e}$ *(Constraint Entailment)*

sEntailT-arrow

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash [C_2] \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1 \end{array}}{P; \Gamma_C; \Gamma \vDash [C_1 \Rightarrow C_2] \rightsquigarrow \lambda \delta_1 : \sigma_1 . e}$$

sEntailT-forall

$$\frac{P; \Gamma_C; \Gamma, a \vDash [C] \rightsquigarrow e}{P; \Gamma_C; \Gamma \vDash [\forall a . C] \rightsquigarrow \Lambda a . e}$$

sEntailT-inst

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q').m \mapsto \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y : e, P_2 \\ P_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_0 \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C'_i \rightsquigarrow \sigma'_i}^{\,i} \\ \overline{\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_C C_y \rightsquigarrow \sigma''_y}^{\,y} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ e'_0 = \Lambda \overline{a}_j . \lambda \, \overline{\delta'_i : \sigma'_i}^{\,i} . \{ m = \Lambda \overline{b}_k . \lambda \, \overline{\delta_y : \sigma''_y}^{\,y} . e_0 \} \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash e'_0 : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q'] \vDash Q \rightsquigarrow \bullet \vdash e_1 \end{array}}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e_1}$$

sEntailT-local

$$\frac{\begin{array}{c} (\delta : C) \in \Gamma \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta : C] \vDash Q \rightsquigarrow \bullet \vdash e \end{array}}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e}$$

$\boxed{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_1}$ *(Constraint Matching)*

sMatchT-arrow

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C}, \delta_1 : C_1 \vdash e_0 \, \delta_1 : C_2] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_2 \\ P; \Gamma_C; \Gamma \vDash [[\overline{\tau} / \overline{a}] C_1] \rightsquigarrow e_1 \end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C_1 \Rightarrow C_2] \vDash Q \rightsquigarrow \overline{\tau} \vdash [e_1 / \delta_1] e_2}$$

sMatchT-forall

$$\frac{P; \Gamma_C; \Gamma; [\overline{a}, a; \overline{\delta} : \overline{C} \vdash e_0 \, a : C] \vDash Q \rightsquigarrow \overline{\tau}, \tau \vdash e_1}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : \forall a . C] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_1}$$

sMatchT-classconstr

$$\frac{\begin{array}{c} \tau_1 = [\overline{\tau} / \overline{a}] \tau_0 \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_i \rightsquigarrow \sigma_i}^{\,i} \end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : TC \, \tau_0] \vDash TC \, \tau_1 \rightsquigarrow \overline{\tau} \vdash [\overline{\sigma} / \overline{a}] e_0}$$

Figure 9.2: Constraint resolution for $\lambda^{\Rightarrow}_{\mathbf{TC}}$

9.2.2 Example Derivation

As an example of the new bidirectional type system for $\lambda^{\Rightarrow}_{\mathbf{TC}}$, consider typing the following program:

```
class A a where
  foo :: a → a
instance A Bool where
  foo = impl1
instance (A a, A b) ⇒ A (a, b) where
  foo = impl2
instance (A b, ∀ a. A a ⇒ A (a, b)) ⇒ A [b] where
  foo = impl3
foo :: [Bool] → [Bool] [True, True]
```

Note that this program will not be accepted by GHC, even with the QuantifiedConstraints extension, as the third instance does not respect the termination condition: $A\ (a, b)$ is not smaller than $A\ [b]$. This condition is introduced in Chapter 8 and is actively enforced in GHC (unless the *UndecidableInstances* language extension is enabled) to guarantee termination of the type inference algorithm. As this chapter is not concerned with type inference, we will not verify this condition in the bidirectional system.

As typing class and instance declarations has not been altered in an interesting way, compared to $\lambda_{\mathbf{TC}}$, we will focus on the typing derivation for the expression $(foo :: [Bool] \to [Bool])\ [True, True]$. While our calculus does not feature tuples or lists, for the sake of this exercise, we will assume corresponding typing rules.

Note that typing a method *foo* in our bidirectional type system requires an explicit type annotation $[Bool] \to [Bool]$ in order for the expression to be accepted. This allows us to go to checking mode, where we can apply rule sTm-CheckT-meth:

$$\dfrac{\dfrac{\dfrac{(foo : Aa : a \to a) \in \Gamma_C \qquad P;\Gamma_C;\Gamma \vDash [A[Bool]] \rightsquigarrow e \qquad \ldots}{P;\Gamma_C;\Gamma \vdash_{tm} foo \Leftarrow [Bool] \to [Bool] \rightsquigarrow e}\ \textsc{sTm-checkT-meth}}{P;\Gamma_C;\Gamma \vdash_{tm} foo :: [Bool] \to [Bool] \Rightarrow [Bool] \to [Bool] \rightsquigarrow e}\ \textsc{sTm-infT-Ann} \qquad \dfrac{\ldots}{P;\Gamma_C;\Gamma \vdash_{tm} [\, True\, ,\ True\,] \Rightarrow [Bool] \rightsquigarrow [\, True\, ,\ True\,]}\ \textsc{sTm-infT-List}}{P;\Gamma_C;\Gamma \vdash_{tm} (foo :: [Bool] \to [Bool])\, [\, True\, ,\ True\,] \Rightarrow [Bool] \rightsquigarrow e\, [\, True\, ,\ True\,]}\ \textsc{sTm-infT-ArrE}$$

The most interesting aspect here is the constraint entailment derivation. Handling the three class and instance declarations above results in the following environments:

$$\begin{aligned}
\Gamma_C &= foo : \bullet \Rightarrow A a : a \rightarrow a \\
P &= (D_1 : A Bool).foo \mapsto \bullet : impl1 \\
&\quad , \; (D_2 : \forall a.\forall b.A a \Rightarrow A b \Rightarrow A(a, b)).foo \mapsto \bullet, a, b, \delta_1 : A a, \delta_2 : A b : impl2 \\
&\quad , \; (D_3 : \forall b.A b \Rightarrow (\forall a.A a \Rightarrow A(a, b)) \Rightarrow A[b]).foo \mapsto \\
&\qquad \bullet, b, \delta_3 : A b, \delta_4 : \forall a.A a \Rightarrow A(a, b) : impl3
\end{aligned}$$

Figure 9.3 shows an example derivation for proving $A[Bool]$, where we define P_1 and P_2 to contain the first and first two instance declarations respectively. The entailment process works by first selecting the instance declaration for lists (the third declaration) and matching against it. We then deconstruct this axiom, adding b to the list of instantiatable variables, and adding both constraints to the residual constraints. When we have deconstructed the axiom into a class constraint $A[b]$, we unify b with $Bool$, and propagate this unification back. The residual constraints thus become $A Bool$ and $\forall a.A a \Rightarrow A(a, Bool)$. These are solved recursively, and their proofs are substituted in our proof for $A[Bool]$. This results in the final $F_{\{\}}$ expression $(\Lambda b.\lambda \delta_3 : \sigma_3.\lambda \delta_4 : \sigma_4.\{m = e_0\})\; Bool\; e_2\; e_3$.

9.2.3 $F_{\mathbf{D}}^{\Rightarrow}$ Updates

Besides translating directly from the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ calculus into $F_{\{\}}$, like we do in Chapter 8, we follow the example set in Chapter 7 and introduce an intermediate language. This section introduces $F_{\mathbf{D}}^{\Rightarrow}$ as an extension of $F_{\mathbf{D}}$: System F with explicit dictionaries (like in Chapter 7), with support for more expressive dictionaries, which are able to represent quantified constraints.

The syntax for the $F_{\mathbf{D}}^{\Rightarrow}$ grammar is shown in Figure 9.4. The extension on top of $F_{\mathbf{D}}$ is twofold: (1) mirroring $\lambda_{\mathbf{TC}}^{\Rightarrow}$, the intermediate language supports quantified constraints C, and (2) in order to represent these constraints, dictionaries d have been made significantly more expressive. Note the non-standard definition of dictionary values. Unlike in regular dictionaries, a dictionary constructor is only a value when fully applied. By disallowing partially applied dictionary constructors in values, we gain the canonical forms property where we can determine the form of the dictionary based purely on its type. For example, a dictionary value with a type $C_1 \Rightarrow C_2$ has to be of the form $\lambda \delta : C_1.d$. Without this restriction, the dictionary value could also be a partially applied constructor. This property is exploited, for instance when proving the compatibility lemma

$$\dfrac{\begin{array}{c} P_1;\Gamma_C;\bullet,b,\delta_3 : Ab,\delta_4 : \forall a.Aa \Rightarrow A(a,b) \vdash_{tm} impl3 \Rightarrow \tau_0 \rightsquigarrow e_0 \\ \Gamma_C;\bullet,b \vdash_C Ab \rightsquigarrow \sigma_3 \qquad \Gamma_C;\bullet,b \vdash_C \forall a.Aa \Rightarrow A(a,b) \rightsquigarrow \sigma_4 \\ e_1 = (\Lambda b.\lambda\delta_3 : \sigma_3.\lambda\delta_4 : \sigma_4.\{m = e_0\})\, Bool\, \delta_5\, \delta_6 \end{array}}{P;\Gamma_C;\Gamma;[\bullet,b;\bullet,\delta_5 : Ab,\delta_6 : \forall a.Aa \Rightarrow A(a,b) \vdash (\Lambda b.\lambda\delta_3 : \sigma_3.\lambda\delta_4 : \sigma_4.\{m = e_0\})\, b\, \delta_5\, \delta_6 : A[b]] \vDash A[Bool] \rightsquigarrow \bullet, Bool \vdash e_1} \textsc{sEntailT-classConstr}$$

$$\dfrac{\dfrac{\cdots}{P;\Gamma_C;\Gamma \vDash [\forall a.Aa \Rightarrow A(a,Bool)] \rightsquigarrow e_3}\textsc{sEntailT-inst}}{P;\Gamma_C;\Gamma;[\bullet,b;\bullet,\delta_5 : Ab \vdash (\Lambda b.\lambda\delta_3 : \sigma_3.\lambda\delta_4 : \sigma_4.\{m = e_0\})\, b\, \delta_5 : (\forall a.Aa \Rightarrow A(a,b)) \Rightarrow A[b]] \vDash A[Bool] \rightsquigarrow \bullet, Bool \vdash [e_3/\delta_6]e_1} \textsc{sMatchT-arrow}$$

$$\dfrac{\dfrac{\cdots}{P;\Gamma_C;\Gamma \vDash [ABool] \rightsquigarrow e_2}\textsc{sEntailT-inst}}{P;\Gamma_C;\Gamma;[\bullet,b;\bullet \vdash (\Lambda b.\lambda\delta_3 : \sigma_3.\lambda\delta_4 : \sigma_4.\{m = e_0\})\, b : Ab \Rightarrow (\forall a.Aa \Rightarrow A(a,b)) \Rightarrow A[b]] \vDash A[Bool] \rightsquigarrow \bullet, Bool \vdash [e_2/\delta_5][e_3/\delta_6]e_1} \textsc{sMatchT-arrow}$$

$$\dfrac{}{P;\Gamma_C;\Gamma;[\bullet;\bullet \vdash \Lambda b.\lambda\delta_3 : \sigma_3.\lambda\delta_4 : \sigma_4.\{m = e_0\} : \forall b.Ab \Rightarrow (\forall a.Aa \Rightarrow A(a,b)) \Rightarrow A[b]] \vDash A[Bool] \rightsquigarrow \bullet \vdash [e_2/\delta_5][e_3/\delta_6]e_1} \textsc{sMatchT-forall}$$

$$\dfrac{}{P;\Gamma_C;\Gamma \vDash [A[Bool]] \rightsquigarrow [e_2/\delta_5][e_3/\delta_6]e_1} \textsc{sEntailT-inst}$$

Figure 9.3: Example Constraint Entailment Derivation

$$\begin{array}{lcll}
e & ::= & \lambda\delta : C.e \mid \ldots & \textit{expression} \\
\\
\sigma & ::= & C \Rightarrow \sigma \mid \ldots & \textit{type} \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a.C & \textit{constraint} \\
\\
d & ::= & \delta \mid D \mid \lambda\delta : C.d \mid d_1\, d_2 \mid \Lambda a.d \mid d\,\sigma & \textit{dictionary} \\
dv & ::= & D\,\overline{\sigma}\,\overline{d} \mid \lambda\delta : C.d \mid \Lambda a.d & \textit{dictionary value}
\end{array}$$

Figure 9.4: Grammar for $F_{\mathbf{D}}^{\Rightarrow}$, extension of Figure 7.2

for method calls (Lemma 127). While this restriction is not technically enforced through the grammar of dictionary values, it is made explicit in our logical relations.

The updates to type and constraint well-formedness are unsurprising and can be found in Appendix A.7.2. The same holds for the expression typing rules, which can be found in Appendix A.7.2. A prominent departure from $F_{\mathbf{D}}$ is the typing relation for dictionaries: whereas before dictionaries enjoyed a fixed structure, they now require a System F-like type system (shown in Figure 9.5). Rule D-CON is the most notable: typing a dictionary constructor - corresponding to an instance declaration - requires looking up both the class (for the type of the method) and instance declaration (for the type of the constructor, and the accompanying implementation), and translating the method implementation. The constructor is then translated into a $F_{\{\}}$ record, where any remaining type and dictionary variables are bound.

Finally, as dictionaries have gotten more expressive, the operational semantics of $F_{\mathbf{D}}^{\Rightarrow}$ need to be extended to dictionaries as well. These additional evaluation rules closely mirror those of System F, and can be found in Figure 9.6.

9.2.4 Example Translation

Illustrating the translation from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\mathbf{D}}^{\Rightarrow}$, we return to the example from Section 9.2.2. Unlike our translation to $F_{\{\}}$, we now translate the method call to a dictionary constructor, rather than passing in the implementation directly. This constructor acts as a pointer to a global set of instance declarations, containing the implementation to use. The proof for $A[Bool]$ thus becomes $D_3\, Bool\, D_1\, (D_2\, Bool\, D_1)$, where D_i refers to the i^{th} instance declaration.

Using the dictionary typing relation, this dictionary gets translated into the exact same $F_{\{\}}$ expression from Section 9.2.2.

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : C \rightsquigarrow e}$ *(Dictionary Typing)*

D-VAR

$$\frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C \rightsquigarrow \delta}$$

D-CON

$$\frac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma'_i}^{\,i} \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q / a]\sigma_m \rightsquigarrow e\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q \rightsquigarrow \Lambda \overline{a}_j . \lambda\, \overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\}}$$

D-DABS

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \lambda \delta : C_1 . d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda \delta : \sigma_1 . e}$$

D-DAPP

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e_1\, e_2}$$

D-TYABS

$$\frac{\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e}{\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a . d : \forall a . C \rightsquigarrow \Lambda a . e}$$

D-TYAPP

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a . C \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d\, \sigma : [\sigma / a]C \rightsquigarrow e\, \sigma}$$

Figure 9.5: Dictionary typing relation for $F_{\mathbf{D}}^{\Rightarrow}$

$\boxed{\Sigma \vdash e \longrightarrow e'}$ *($F_{\mathbf{D}}$ Evaluation)*

$$\textsc{iEval-method}\ \frac{d \longrightarrow d'}{\Sigma \vdash d.m \longrightarrow d'.m} \qquad \textsc{iEval-methodVal}\ \frac{(D : C).m \mapsto e \in \Sigma}{\Sigma \vdash (D\,\overline{\sigma}_m\,\overline{d}_n).m \longrightarrow e\,\overline{\sigma}_m\,\overline{d}_n}$$

$\boxed{d \longrightarrow d'}$ *($F_{\mathbf{D}}$ Dictionary Evaluation)*

$$\textsc{iDictEval-app}\ \frac{d_1 \longrightarrow d_1'}{d_1\,d_2 \longrightarrow d_1'\,d_2} \qquad \textsc{iDictEval-appAbs}\ \frac{}{(\lambda\delta : C.d_1)\,d_2 \longrightarrow [d_2/\delta]d_1} \qquad \textsc{iDictEval-tyApp}\ \frac{d \longrightarrow d'}{d\,\sigma \longrightarrow d'\,\sigma}$$

$$\textsc{iDictEval-tyAppAbs}\ \frac{}{(\Lambda a.d)\,\sigma \longrightarrow [\sigma/a]d}$$

Figure 9.6: Operational Semantics for $F_{\mathbf{D}}^{\Rightarrow}$, Extension of Figure 7.3

9.3 Meta-Theory

As both the source and intermediate calculi ($\lambda_{\mathbf{TC}}^{\Rightarrow}$ and $F_{\mathbf{D}}^{\Rightarrow}$ respectively) have grown more expressive, their meta-theoretical properties need to be revisited as well. Concretely, the properties and proofs from Chapter 7 need to be adapted with support for dictionary operational semantics. This section discusses the main updated theorems related to $\lambda_{\mathbf{TC}}^{\Rightarrow}$, $F_{\mathbf{D}}^{\Rightarrow}$ and the elaboration between them. Section 9.4 handles the proof of coherence of type class resolution.

9.3.1 $F_{\mathbf{D}}^{\Rightarrow}$ Type Safety

When proving progress (Theorem 3) and preservation (Theorem 4) for the $F_{\mathbf{D}}$ calculus, we could make assumptions about the structure of dictionaries. However, as $F_{\mathbf{D}}^{\Rightarrow}$ features a more expressive form of dictionaries, this is no longer possible. For example, in the method case $d.m$ (rule iTm-method), we could assume d to be of the form $D\,\overline{\sigma}\,\overline{d}$. With our more expressive dictionaries, we can no longer make this assumption as d first needs to be evaluated. We thus introduce separate theorems for the type safety of dictionaries.

Theorem 18 (Progress for Dictionaries)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_d d : C$*, then either* d *is a dictionary value, or there exists* d' *such that* $d \longrightarrow d'$*.*

Theorem 19 (Preservation for Dictionaries)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$*, and* $d \longrightarrow d'$*, then* $\Sigma; \Gamma_C; \Gamma \vdash_d d' : C$*.*

The formal proof of Theorems 18 and 19 can be found in Appendix C.5.3.

9.3.2 Strong Normalisation for $F_{\mathsf{D}}^{\Rightarrow}$

Following Chapter 7, $F_{\mathsf{D}}^{\Rightarrow}$ does not support recursive expressions. Strong normalisation (Theorem 5) thus continues to hold. The proof of strong normalisation for $F_{\mathsf{D}}^{\Rightarrow}$ requires considerable changes though: (1) A separate logical relation is added to denote normalising dictionaries (Figure 9.7). (2) The strong normalisation relation for expressions has been updated, along with the context interpretation relations, to account for the more advanced dictionaries. (3) A separate—non-trivial—proof of strong normalisation for dictionaries is added (Appendix C.5.4). (4) The existing proof of strong normalisation for expressions is updated to account for the evaluation of dictionaries.

9.3.3 Elaboration from $\lambda_{\mathsf{TC}}^{\Rightarrow}$ to $F_{\mathsf{D}}^{\Rightarrow}$

The type preservation theorems are updated with the more expressive constraints and corresponding dictionaries. Type preservation for expressions, class and instance declarations are altered in a non-surprising manner. The more expressive constraints are incorporated in preservation of type and constraint well-formedness (Theorem 25) as well. Note though that preservation for constraint proving is now split in two separate theorems, for constraint entailment and matching.

9.3.4 Elaboration from $F_{\mathsf{D}}^{\Rightarrow}$ to $F_{\{\}}$

While elaborating $F_{\mathsf{D}}^{\Rightarrow}$ dictionaries to $F_{\{\}}$ expressions has gotten more involved, the translation remains deterministic. The property is formally proven in Appendix C.8.3.

$\boxed{d \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}}$ (Strong Normalization Relation for Dictionaries)

$$d \in \mathcal{SN}[\![\, TC\,\sigma]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : TC\,\sigma \wedge \exists D, \overline{\sigma}_j, \overline{d}_i : d \longrightarrow^{*} D\,\overline{\sigma}_j\,\overline{d}_i$$

$$\text{where } \Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2$$

$$\wedge\ (m : TC\,a : \sigma_m) \in \Gamma_C \wedge\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_{ty} \sigma_j}^{\,j}$$

$$\wedge\ \overline{d_i \in \mathcal{SN}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Sigma,\Gamma_C}}^{\,i} \wedge\ \sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q$$

$$\wedge\, e \in \mathcal{SN}[\![\forall \overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma_m]\!]_{\bullet}^{\Sigma_1,\Gamma_C}$$

$$d \in \mathcal{SN}[\![C_1 \Rightarrow C_2]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : C_1 \Rightarrow C_2 \wedge \exists dv : d \longrightarrow^{*} dv$$

$$\wedge\, \forall d' : d' \in \mathcal{SN}[\![C_1]\!]^{\Sigma,\Gamma_C} \Rightarrow d\,d' \in \mathcal{SN}[\![C_2]\!]^{\Sigma,\Gamma_C}$$

$$d \in \mathcal{SN}[\![\forall a.C]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : \forall a.C \wedge \exists dv : d \longrightarrow^{*} dv$$

$$\wedge\, \forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow d\,\sigma \in \mathcal{SN}[\![[\sigma/a]C]\!]^{\Sigma,\Gamma_C}$$

Figure 9.7: Strong Normalisation Relation for Dictionaries

Similarly, dictionary elaboration soundness from $F_{\overset{\Rightarrow}{\mathbf{D}}}$ to $F_{\{\}}$ becomes more convoluted as the possible forms of dictionaries increase. Note that term and dictionary elaboration soundness need to be proven by mutual induction, as a dictionary constructor (corresponding to an instance declaration) is translated into (the translation of) its accompanying method implementation. The full proofs can be found in Appendix C.8.2.

Finally, the inclusion of evaluation for dictionaries complicates the semantic preservation proof. A separate theorem is added for semantic preservation for dictionaries to amend Theorem 70. The full proofs are shown in Appendix C.8.4.

9.4 Coherence

As $F_{\mathbf{D}}^{\Rightarrow}$ dictionaries now feature their own operational semantics, proving coherence gets more involved. The definitions of our logical relations are adapted to this more expressive form of dictionaries (Section 9.4.1), and the coherence theorems are updated accordingly (Section 9.4.2).

9.4.1 Logical Relations

This section presents the required updates to the logical relations from Chapter 7, in order to account for more expressive dictionaries with evaluation. The changes are shown in Figures 9.8 and 9.9.

Firstly, as dictionaries have grown more expressive, with their own operational semantics, we can no longer make assumptions about their form. We thus introduce a separate relation for closed dictionaries: $(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C]\!]^{\Gamma_C}$. Note that even though the evaluation of dictionaries closely mirrors that of System F, its logical relations are defined quite differently. This is the case for two main reasons: (1) The expression value relation is defined over every type, and the form of the expression can be determined based on its type. For example, a closed expression value with a function type $\sigma_1 \rightarrow \sigma_2$ has to be unifyable with $\lambda x : \sigma_1 . e$. The same does not hold for dictionaries: a dictionary value with a given type $C_1 \Rightarrow C_2$ could be either a lambda form $\lambda \delta : C_1 . d$ or a partially instantiated dictionary constructor $D \, \overline{\sigma}_m \, \overline{d}_n$. For this reason, we restrict the dictionary value relation to class constraints only. This restriction makes sense as a dictionary needs to be fully instantiated into a class constraint type, before it can be applied to a method. Furthermore, while the closed dictionary relation differentiates on the type of the dictionary, it does not make any assumptions on the form of the dictionary itself. (2) When handling a function or forall type in the expression value relation, we can apply it to an argument and evaluate the result until we reach another element of the value relation. However, this approach does not work for dictionaries, as we can't distinguish a lambda dictionary from a partially applied constructor, solely by the type. A partially applied constructor does not evaluate, and we don't want to accept this into the value relation, as it would break our ability to derive the form of a dictionary by its type. For this reason, we build up the dictionary step by step in the closed dictionary relation, only evaluate at the very end when we achieve a class constraint, and only allow fully instantiated constructors in the value relation.

Secondly, both the expression value relation and the context interpretation judgments have been updated. Expressions now allow abstraction over quantified constraints, which are instantiated with closed dictionaries. The same holds true for the interpretation judgment $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$.

Thirdly, while not strictly necessary, we simplified the open dictionary relation. While the expression relations need to keep track of a type variable mapping R, this is not actually the case for dictionaries. As constraints can never take the form of a type variable, these can be substituted right away, and we no longer need to keep track of the mapping.

9.4.2 Coherence Theorem Updates

This section discusses the required updates to both the coherence theorem proofs and required helper theorems. We restate our coherence theorem for expressions for clarity:

Theorem 2 (Expression Coherence - Restated)**.**
If $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
then $P;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Chapter 7 proves this theorem for $\lambda_{\mathbf{TC}}$ using a five step approach: (Step 1) We prove that an elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ can always be decomposed into an equivalent set of two translations through $F_{\mathbf{D}}$ (Theorem 10). (Step 2) We show that any two $F_{\mathbf{D}}$ translations from the same $\lambda_{\mathbf{TC}}$ expression are logically equivalent (Theorem 14). (Step 3) We show that any two logically equivalent $F_{\mathbf{D}}$ expressions are also contextually equivalent (Theorem 15). (Step 4) We prove that translating $F_{\mathbf{D}}$ expressions to $F_{\{\}}$ is contextual equivalence preserving with $F_{\mathbf{D}}$ contexts (Theorem 16). (Step 5) We show that any two contextually equivalent $F_{\{\}}$ expressions under $F_{\mathbf{D}}$ contexts are also contextually equivalent under $\lambda_{\mathbf{TC}}$ contexts (Theorem 17).

Step 1 is proven through mutual induction on several different theorems for expressions, environments, types, etc. While every theorem requires mild alterations, the bulk of the work can be found in the constraint entailment theorem, which is now split into constraint entailment and matching:

Theorem 11 (Equivalence - Constraint Entailment)**.**
If $P;\Gamma_C;\Gamma \vDash [C] \rightsquigarrow e$ *and* $\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma$
then $P;\Gamma_C;\Gamma \vDash^M [C] \rightsquigarrow d$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_d d : C \rightsquigarrow e$
where $\Gamma_C;\Gamma \vdash_C^M C \rightsquigarrow C$ *and* $\vdash_{ctx}^M P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$.

$\boxed{(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}}$ (Closed Dictionary Value Relation)

$$(\Sigma_1 : D\,\overline{\sigma}_m\,\overline{d}_{n\,1}, \Sigma_2 : D\,\overline{\sigma}_m\,\overline{d}_{n\,2}) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_m\,\overline{d}_{n\,1} : Q \rightsquigarrow e_1 \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_m\,\overline{d}_{n\,2} : Q \rightsquigarrow e_2$$

$$\forall d_{1\,i}, d_{2\,i} : \overline{(\Sigma_1 : d_{1\,i}, \Sigma_2 : d_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_m/\overline{a}_m]C_i]\!]^{\Gamma_C}}^{\,i<n}$$

$$\textit{where } (D : \forall \overline{a}_m.\overline{C}_n \Rightarrow Q').m \mapsto e_1 \in \Sigma_1 \wedge Q = [\overline{\sigma}_m/\overline{a}_m]Q'$$

$\boxed{(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C]\!]^{\Gamma_C}}$ (Closed Dictionary Relation)

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C_1 \Rightarrow C_2]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : C_1 \Rightarrow C_2 \rightsquigarrow e_2$$

$$\wedge \forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![C_1]\!]^{\Gamma_C} \Rightarrow (\Sigma_1 : d_1\,d_3, \Sigma_2 : d_2\,d_4) \in \mathcal{E}[\![C_2]\!]^{\Gamma_C}$$

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![\forall a.C']\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : \forall a.C' \rightsquigarrow e_1 \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : \forall a.C' \rightsquigarrow e_2$$

$$\wedge \forall \sigma : \Gamma_C;\bullet \vdash_{ty} \sigma \rightsquigarrow \sigma \Rightarrow (\Sigma_1 : d_1\,\sigma, \Sigma_2 : d_2\,\sigma) \in \mathcal{E}[\![[\sigma/a]C']\!]^{\Gamma_C}$$

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![Q]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : Q \rightsquigarrow e_1 \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : Q \rightsquigarrow e_2$$

$$\wedge \exists dv_1, dv_2, d_1 \longrightarrow^* dv_1, d_2 \longrightarrow^* dv_2, (\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$$

$\boxed{\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C}$ (Logical Equivalence for Open Dictionaries)

$$\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C$$

$$\triangleq \forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}, \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C},$$

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

Figure 9.8: Updated Logical Relations for Dictionaries

$\boxed{(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C}}$ (Closed Expression Value Relation)

$$
\begin{aligned}
&(\Sigma_1 : \lambda\delta : C.e_1, \Sigma_2 : \lambda\delta : C.e_2) \in \mathcal{V}[\![C \Rightarrow \sigma]\!]_R^{\Gamma_C}\\
&\triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : C.e_1 : R(C \Rightarrow \sigma) \rightsquigarrow e_1\\
&\wedge\, \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : C.e_2 : R(C \Rightarrow \sigma) \rightsquigarrow e_2\\
&\wedge\, \forall (\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} :\\
&(\Sigma_1 : (\lambda\delta : C.e_1)\, d_1, \Sigma_2 : (\lambda\delta : C.e_2)\, d_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}
\end{aligned}
$$

Figure 9.9: Updated Logical Relations for Expressions, Extension of Definitions 6 and 7

Theorem 20 (Equivalence - Constraint Matching)**.**
If $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C_0] \vDash Q_1 \rightsquigarrow \overline{\tau} \vdash e_1$
and $\Sigma; \Gamma_C; \Gamma, \overline{a}, \overline{\delta} : \overline{C} \vdash_d d_0 : C_0 \rightsquigarrow e_0$
where $\Gamma_C; \Gamma, \overline{a} \vdash_C^M C_0 \rightsquigarrow C_0$ *and* $\overline{\Gamma_C; \Gamma, \overline{a} \vdash_C^M C_i \rightsquigarrow C_i}^{\,i}$
and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx}^M P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : C_0] \vDash^M Q_1 \rightsquigarrow \overline{\tau} \vdash d_1$
and $\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d d_1 : Q_1 \rightsquigarrow e_1$ *where* $\Gamma_C; \Gamma \vdash_Q^M Q_1 \rightsquigarrow Q_1$.

Pay special attention to the second theorem: it handles the case where a wanted constraint Q_1 matches a given axiom C_0 in the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ language, and thus transforms a given proof e_0 into a new proof e_1 for Q_1 in the $F_{\{\}}$ language. The theorem states that when the given $F_{\{\}}$ proof e_0 is the translation of a $F_{\mathbf{D}}^{\Rightarrow}$ dictionary d_0, that the aforementioned process can be decomposed by first constructing a $F_{\mathbf{D}}^{\Rightarrow}$ dictionary d_1 from d_0 and then translating it into the original proof e_1.

Step 2 is one of the most involved parts of our coherence proof as it consists of many moving parts. The theorem requires updates in four distinct aspects of the proof. First, the coherence theorem for expressions itself (Theorem 14) needs to be adapted to $\lambda_{\mathbf{TC}}^{\Rightarrow}$. While this entails small updates throughout the theorem, the rule sTm-check-var and rule sTm-check-meth cases are most interesting, as they interact directly with $F_{\mathbf{D}}^{\Rightarrow}$ dictionaries. Proving these cases requires coherence for dictionaries, reflexivity Theorem 50, and updated compatibility lemmas.

Second, showcasing the adaptability of our coherence proof, the coherence theorem for dictionaries (Theorem 55) is the only major theorem in the proof that needed to be rewritten from scratch. In order to make the proof more manageable, the theorem is split in four parts: We first perform induction on both constraint entailment derivations, deconstructing the wanted constraint to a class constraint. At that point, four options arise as both derivations can continue by matching against an instance declaration or local axiom. While this alters the starting dictionary, every case continues in the same manner by induction on the first constraint matching derivation, using the second part of this theorem. For the constraint matching theorem we can not assume that the given axiom we are matching against or the given evidence are the same across the derivations. We thus create two separate theorems, performing induction on the first and second derivation respectively. We deconstruct the first axiom into a class constraint in Theorem 56 before deconstructing the second axiom in Theorem 57. The final case, where both axioms are reduced to class constraints is most interesting: we introduce a separate Theorem 12 stating that any two closed dictionaries, typed with the same class constraint under logically related environments, are always logically equivalent. The clue which makes this work is the fact that the environments are logically related and that related environments can not contain overlapping instances unless their method implementations are related as well. Note that this theorem replaces the old Theorem 12, which only applied to dictionary values.

Third, reflexivity Theorems 50 and 49 are adapted to match the updated logical relations for expressions and dictionaries respectively. Notably, accounting for the evaluation of dictionaries, Lemma 130 is added, stating that the logical relation for dictionaries is preserved by evaluation. Note that we define the size of a class constraint to be 1, independently of its contained type, in order to make the induction well-founded.

Fourth, compatibility lemmas are added to account for the new dictionary forms. More notably, method compatibility Lemma 127 has been adapted with the new logical relations. The proof of this lemma is aided by our more restrictive dictionary value relation: as this logical relation is only defined for fully applied dictionary constructors (with a class constraint type), we can now make stronger assumptions on the form of the dictionaries in this relation.

Step 3 , **Step 4** and **Step 5** continue to follow from the definition of the logical relation and contextual equivalence respectively, and thus do not need to be altered.

9.5 Conclusion

This chapter has presented an extension of the proof of coherence for type class resolution, with quantified constraints. In doing so, we have shown the adaptability of the existing proof.

The language extension adds significant expressivity to the type class resolution mechanism, a core part of the coherence proof. When updating the proof, we thus made the decision to extend both the source calculus $\lambda_{\mathbf{TC}}$ and the intermediate calculus $F_{\mathbf{D}}$ with this feature. While this simplifies the translation steps and consequently the proof, it comes at the cost of increasing the scope of this extension to about half of the theorems in the proof.

Illustrating the size of this non-trivial extension: of the 51 theorems in the proof, 13 remained unchanged, 14 received small, straightforward updates, 13 have been significantly adapted and 11 theorems are new. The change is smaller for the lemmas in this work: of the 86 lemmas, 51 remain unchanged, 17 received minor updates, 2 have been significantly rewritten, and 16 are new.

Counting the updates line per line shows that this extension alters 25% of lines in the definition of the grammar and typing rules, and 43% of lines in the original proof. Of these changes, 52% are additions and 48% are modifications. This way of quantifying the changes shows a slightly different picture from before, as small, uninteresting updates can still span a large number of lines.

This chapter contains new original work and is as of yet unpublished. Concretely, the alternative formulation of quantified constraints, as well as the proof extensions and the conclusions thereof are novel. We do build on Chapter 7 for the original coherence proof and on Chapter 8 for the original formulation of the quantified constraints language extension. Both of these chapters are published work.

Chapter 10

Conclusion

The aim of this thesis is to improve the current state of the art in polymorphism, by evaluating and proving both correctness and stability properties, as well as by introducing new features. Part I focuses on the concept of stability and applies it to the process of type instantiation. Part II focuses on proving coherence for type class resolution. We introduce quantified class constraints as a case study, and evaluate the adaptability of the coherence proof with this extension.

Though we largely focus on the Haskell programming language in this text, the concepts and properties we discuss are more general. We thus believe that most of the results from this work are more widely applicable. We discuss the different results and contributions of this work below.

10.1 Parametric Polymorphism

Type Instantiation

Chapter 4 lays the groundwork for our work on stability in Chapter 5. It provides background on different forms of type instantiation, and discusses a number of design decisions related to instantiation. We introduce a family of type systems, which can be reified into four different methods of instantiation, and featuring a mix of explicit and implicit behaviour. These systems are inspired by Peyton Jones et al. [73], Eisenberg et al. [24], and Serrano et al. [85].

The contributions of this chapter are twofold:

1. We provide a comparison of the instantiation strategies in Haskell, Agda, and Idris (Section 5.5).
2. We design a family of type systems, based on the bidirectional type-checking algorithm implemented in GHC [24, 73, 85]. It is parameterised over the flavour of type instantiation. (Section 5.2)

Stability

Chapter 5 introduces the concept of *stability* in a language that supports both implicit and explicit arguments. We believe that designers of all languages supporting this mix of features need to grapple with how to best incorporate these features; those designers may wish to follow our lead in formalising the problem to seek the most stable design. While stability is uninteresting in languages featuring pure explicit or pure implicit instantiation, it turns out to be an important metric in the presence of mixed behaviour.

Using the family of type systems defined in Chapter 4, we evaluated the different flavours of instantiation, against a set of formal stability properties. The results are surprisingly unambiguous: (a) *lazy instantiation* achieves the highest stability for the compile time semantics, and (b) *shallow instantiation* results in the most stable runtime semantics.

The contributions of this chapter are twofold:

1. We introduce the concept of stability, as a method of evaluating user-facing design decisions. While it is not direct replacement for large user-studies, we present stability as a more formal complementary approach.
2. We enumerate stability properties relevant for examining instantiation in Haskell, along with examples of how these properties affect programmer experience. (Section 5.1)
3. We provide an analysis of how different choices of instantiation flavour either respect or do not respect the similarities we identify. We conclude that lazy, shallow instantiation is the most stable. (Section 5.3; proofs in coherence proof Appendix E) We hope this work proves useful in reopening a productive discussion regarding the implementation of type instantiation in GHC.

Future Work

Chapter 4 introduces both eagerness and depth as possible design decisions with an impact on the stability of type instantiation. These axes were chosen since they are an ongoing topic within the GHC community due to the recent adoption of shallow instantiation in GHC 9.0. [1] Furthermore, they are interesting because GHC has gone back and forth on its choice of instantiation, as the GHC community has difficulty agreeing on it. However, other relevant design decisions do exist. Consider for example the instantiation of multi equation declarations. In our current definition of MPLC, the type of every branch is instantiated in order to simplify the search for a single subsuming type for the declaration. Investigating alternative algorithms could have a significant impact on the stability of the language. For example, Property 10—one of the few properties that favours eager instantiation—could potentially give a different outcome, strengthening our claim that lazy instantiation is more user friendly. Or consider Property 4: due to the current, somewhat hard to predict behaviour of multi equation type inference, Property 4 does not hold in any of our four calculi. These properties certainly warrant further investigation of the handling of multi equation declarations.

Chapter 5 evaluates the stability of MPLC, a calculus which features both implicit and explicit type instantiation. Following the example set by GHC, we assume types are instantiated in an ordered fashion. Alternatively, languages like Agda allow for named instantiation. Section 5.5 provides a brief discussion on this form of type instantiation. Note that named instantiation can reduce or eliminate the need for a distinction between inferred and specified type variables, as this separation is based entirely on the need to positionally instantiate generalised variables. Omitting this distinction means that several of our properties would trivially hold. Introducing this as an alternative method of type instantiation in GHC warrants a more thorough investigation, as it could have a significant impact on the stability of the language.

The recent inclusion of the quick look algorithm in GHC 9.2, makes a difficult trade-off exchanging stability for additional expressiveness for the type inference engine. Formally investigating the stability impact of this feature makes for a highly interesting research track.

[1] Motivated by the quick-look algorithm, as discussed in `https://github.com/ghc-proposals/ghc-proposals/pull/287`

10.2 Ad-Hoc Polymorphism

Type Classes

Chapter 6 provides background on ad-hoc polymorphism and type classes in general. It introduces the $\lambda_{\mathbf{TC}}$ Haskell-like calculus with support for type classes and superclasses, and the $F_{\{\}}$ calculus to be used throughout Part II.

The main contributions of this chapter are thus:

1. We present the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ basic calculus with full-blown type class resolution (incl. superclasses), which isolates nondeterministic resolution.
2. We show an elaboration from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to the target language $F_{\{\}}$, System F with records, which are used to encode dictionaries.

Coherence

Chapter 7 has formally proven that type class resolution is coherent by means of logical relations and an intermediate language with explicit dictionaries. While we use Haskell examples throughout the chapter, the calculi and properties are more general. The proof is thus relevant for any language with type class resolution.

The contributions of this chapter are as follows:

1. We present the $F_{\mathbf{D}}$ calculus, System F with explicit dictionary-passing. This language enforces the uniqueness of dictionaries, which captures the intention of type class instances. We study its meta-theory, and define a logical relation to prove contextual equivalence.
2. We show elaborations from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\mathbf{D}}$ and from $F_{\mathbf{D}}$ to $F_{\{\}}$. We prove that a direct translation from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\{\}}$ can always be decomposed into an equivalent translation through $F_{\mathbf{D}}$.
3. We provide a formal proof of coherence of the elaboration between $\lambda_{\mathbf{TC}}^{\Rightarrow}$ and $F_{\mathbf{D}}$, using logical relations.
4. We provide a formal proof that coherence is preserved through the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$. As a consequence, by combining this with the previous result, we prove that the elaboration between $\lambda_{\mathbf{TC}}^{\Rightarrow}$ and $F_{\{\}}$ is coherent. The latter coherence result implies coherence of elaboration-based type class resolution in the presence of superclasses and flexible contexts.

Quantified Constraints

Chapter 8 has presented a fully fledged design of quantified class constraints. We have shown that this feature significantly increases the modelling power of type classes, while at the same enables a terminating type class resolution for a larger class of applications.

The contributions of this chapter are as follows:

1. We present an informal discussion of the different advantages of quantified class constraints, illustrated with examples.

2. We provide a fully fledged formalisation of quantified class constraints: the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ calculus. Our formalisation borrows the idea of focusing from Cochis [83], and adapts it to the Haskell setting. We account for two notable differences: a global set of non-overlapping instances and support for superclasses.

3. We design a type inference algorithm that conservatively extends that of Haskell 98, complete with a dictionary-passing elaboration into System F.

4. We present an informal discussion of the termination conditions on a system with quantified class constraints.

5. We make a prototype implementation available, which incorporates higher-kinded datatypes and accepts all[2] examples in this chapter, which can be found at `https://github.com/gkaracha/quantcs-impl`.

Future Work

While Chapter 8 increases the expressive power of type classes, we can still go a step further. As asked in a long-standing open GHC feature request #5927, the system can be extended further with quantification over predicates, raising the power of type classes to (a fragment of) second-order logic.

As discussed in Section 8.8, GHC has opted to avoid backtracking entirely in favour of a conservative, mild heuristic based approach of selecting overlapping axioms. While the GHC community was worried about the performance penalty of a backtracking algorithm for selecting axioms, this has not yet been investigated. Implementing this alternative algorithm, and doing a proper analysis on its performance impact, could convince the community to adopt this more expressive approach.

[2] except for the *HFunctor* example, which needs higher-rank types [73].

While the *QuantifiedConstraints* language pragma has been available for several years now, it still has a number of limitations and problems. At the time of writing, GHC has 42 open issues related to quantified constraints. Investigating and solving these issues could spread awareness about this extension and could increase its usage.

Coherence for Quantified Constraints

Chapter 9 investigates the coherence of $\lambda_{\mathbf{TC}}^{\Rightarrow}$, an extension of $\lambda_{\mathbf{TC}}$ with quantified class constraints. The chapter discusses the adaptability of the coherence proof as presented in Chapter 7, and denotes the non-trivial changes to both the calculi and the theorems.

The contributions of this chapter are:

1. We design an alternative formalisation of $\lambda_{\mathbf{TC}}^{\Rightarrow}$, featuring quantified class constraints in combination with a bidirectional type system.

2. We provide an updated formulation of $F_{\mathbf{D}}^{\Rightarrow}$, a calculus based on System F with explicit dictionary passing and support for quantified class constraints.

3. We present typing preservation theorems for the elaboration from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\mathbf{D}}^{\Rightarrow}$, including formal proofs (Section 9.3.3).

4. We show an updated formal proof of coherence of the type class resolution mechanism of $\lambda_{\mathbf{TC}}^{\Rightarrow}$ (Section 9.4).

Future Work

This work discusses a number of the meta-theoretical properties of quantified class constraints, including an informal discussion of termination of the type inference algorithm (Section 8.6), a formal proof of typing preservation theorems for the elaboration between $\lambda_{\mathbf{TC}}^{\Rightarrow}$ and $F_{\mathbf{D}}^{\Rightarrow}$ (Section 9.3.3), and a formal proof of coherence of the type class resolution mechanism of $\lambda_{\mathbf{TC}}^{\Rightarrow}$ (Section 9.4). However, several interesting properties remain unproven. A formal proof for termination of the type inference algorithm, as well as soundness of inference with respect to the $\lambda_{\mathbf{TC}}^{\Rightarrow}$ bidirectional typing system, make for interesting future work.

We provide two separate versions of the proof of coherence for type class resolution: (1) for $\lambda_{\mathbf{TC}}$, a Haskell-like calculus with type classes and support for superclasses (Chapter 7), and (2) for $\lambda_{\mathbf{TC}}^{\Rightarrow}$, with support for quantified class constraints (Chapter 9). However, several other widely used GHC features and extensions still remain to be included in this work, e.g. dependent types, higher kinded types and GADT's. A more extensive discussion of possible extensions can be found in Section 7.6.

This thesis presents a formal proof for coherence of type class resolution. While work has started on a mechanised version of this proof in the Coq proof assistant, to this day, this mechanisation has not yet been completed.

Appendix A

Additional Relations

A.1 MPLC Additional Definitions

$\boxed{\mathit{binders}^{\delta}(\sigma) = \overline{a}; \rho}$ *(Binders)*

$$\frac{\textsc{Bndr-ShallowInst}}{\mathit{binders}^{\mathcal{S}}(\rho) = \bullet; \rho} \qquad \textsc{Bndr-ShallowForall}\;\frac{\mathit{binders}^{\mathcal{S}}(\sigma) = \overline{b}; \rho}{\mathit{binders}^{\mathcal{S}}(\forall\,\overline{a}.\sigma) = \overline{a}, \overline{b}; \rho}$$

$$\textsc{Bndr-ShallowInfForall}\;\frac{\mathit{binders}^{\mathcal{S}}(\sigma) = \overline{b}; \rho}{\mathit{binders}^{\mathcal{S}}(\forall\,\overline{\{a\}}.\sigma) = \overline{\{a\}}, \overline{b}; \rho} \qquad \frac{\textsc{Bndr-DeepMono}}{\mathit{binders}^{\mathcal{D}}(\tau) = \bullet; \tau}$$

$$\textsc{Bndr-DeepFunction}\;\frac{\mathit{binders}^{\mathcal{D}}(\sigma_2) = \overline{a}; \rho_2}{\mathit{binders}^{\mathcal{D}}(\sigma_1 \to \sigma_2) = \overline{a}; \sigma_1 \to \rho_2} \qquad \textsc{Bndr-DeepForall}\;\frac{\mathit{binders}^{\mathcal{D}}(\sigma) = \overline{b}; \rho}{\mathit{binders}^{\mathcal{D}}(\forall\,\overline{a}.\sigma) = \overline{a}, \overline{b}; \rho}$$

$$\textsc{Bndr-DeepInfForall}\;\frac{\mathit{binders}^{\mathcal{D}}(\sigma) = \overline{b}; \rho}{\mathit{binders}^{\mathcal{D}}(\forall\,\overline{\{a\}}.\sigma) = \overline{\{a\}}, \overline{b}; \rho}$$

$\boxed{\textit{wrap}\,(\overline{\pi};\, e_1 \;\; e_2)}$ *(Pattern Wrapping)*

$$\textsc{PatWrap-Empty}\;\frac{}{\textit{wrap}\,(\bullet;\, e \;\; e)} \qquad \textsc{PatWrap-Var}\;\frac{\textit{wrap}\,(\overline{\pi};\, e_1 \;\; e_2)}{\textit{wrap}\,(x, \overline{\pi};\, e_1 \;\; \lambda x. e_2)} \qquad \textsc{PatWrap-TyVar}\;\frac{\textit{wrap}\,(\overline{\pi};\, e_1 \;\; e_2)}{\textit{wrap}\,(@a, \overline{\pi};\, e_1 \;\; \Lambda a. e_2)}$$

In addition to including the figures above, this appendix describes our treatment of **let** -declarations and patterns:

Let Binders A **let** -expression **let** *decl* **in** e (rule ETm-InfLet and rule ETm-CheckLet) defines a single variable, with or without a type signature. The declaration typing judgement (Figure 5.3) produces a new context Γ', extended with the binding from this declaration.

Rules Decl-NoAnnSingle and Decl-NoAnnMulti distinguish between a single equation without a type signature and multiple equations. In the former case, we synthesise the types of the patterns using the $\vdash^P$ judgement and then the type of the right-hand side. We assemble the complete type with *type*, and then generalise. The multiple-equation case is broadly similar, synthesising types for the patterns (note that each equation must yield the *same* types $\overline{\psi}$) and then synthesising types for the right-hand side. These types are then instantiated (only necessary under lazy instantiation—eager instantiation would have already done this step). This additional instantiation step is the only difference between the single-equation case and the multiple-equation case. The reason is that rule Decl-NoAnnMulti needs to construct a single type that subsumes the types of every branch. Following GHC, we simplify this process by first instantiating the types.

Rule Decl-Ann checks a declaration with a type signature. It works by first checking the patterns $\overline{\pi}_i$ on the left of the equals sign against the provided type σ. The right-hand sides e_i are then checked against the remaining type σ'_i.

Patterns The pattern synthesis relation $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ and checking relation $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$ are presented in Figure 5.4. As the full type is not yet available, synthesis produces argument descriptors $\overline{\psi}$ and a typing context extension Δ. When checking patterns, the type to check against σ is available, and the relation produces a residual type σ', along with the typing context extension Δ.

Typing a variable pattern works similarly to expressions. Under inference (rule Pat-InfVar) we construct a monotype and place it in the context. When checking a variable (rule Pat-CheckVar), its type σ_1 is extracted from the known function type and placed in the context. Type abstraction $@a$ in both synthesis and checking mode (rule Pat-InfTyVar and rule Pat-CheckTyVar respectively) produces a type argument descriptor $@a$ and extends the typing environment.

Typing data constructor patterns (rule Pat-InfCon and rule Pat-CheckCon), works by looking up the type $\forall\,\overline{a}_0.\overline{\sigma}_0 \to T\,\overline{a}_0$ of the constructor K in the typing context, and checking the applied patterns $\overline{\pi}$ against the instantiated type, and an extended context[1]. The remaining type should be the result type for the constructor, meaning that the constructor always needs to be fully applied. Note that full type schemes $\overline{\sigma}_1$ are allowed in patterns, where they are used to instantiate the variables $\overline{a}_0$ (possibly extended with guessed monotypes $\overline{\tau}_0$, if there are not enough $\overline{\sigma}_1$). Consider, for example, $f\;(\mathit{Just}\,@\mathbf{Int}\;x) = x + 1$, where the $@\mathbf{Int}$ refines the type of *Just*, which in turn assigns x the type **Int**. Note that pattern checking allows skolemising bound type variables (rule Pat-CheckInfForall), but only when the patterns are not empty in order not to lose syntax-directedness of the rules. The same holds for rule Pat-CheckForall, which only applies when no other rules match.

A.2 MPLC Core Language Definitions

The dynamic semantics of the languages in Section 5.2 are defined through a translation to System F. While the target language is largely standard, a few interesting remarks can be made. The language supports nested pattern matching through case lambdas $\textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \to e_i}^{\,i}$, where patterns π_F include both term and type variables, as well as nested constructor patterns. Note that while we reuse our type σ grammar for the core language, System F does not distinguish between inferred and specified binders.

We also define two meta-language features to simplify the elaboration, and the proofs: Firstly, in order to support eta-expansion (for translating deep instantiation to System F), we define expression wrappers $\dot{t}$, essentially a limited form of expressions with a hole $[]$ in them. An expression e can be filled in for the hole to get a new expression $\dot{t}[e]$. One especially noteworthy wrapper construct is $\lambda e_1.e_2$, explicitly abstracting over and handling the expression to be filled in. Note that, as expression wrappers are only designed to alter the type of expressions through eta-expansion, there is no need to support the full System F syntax.

[1] Extending the context for later patterns is not used in this system, but it would be required for extensions like view patterns.

Secondly, in order to define contextual equivalence, we introduce contexts M. These are again expressions with a hole $[]$ in them, but unlike expression wrappers, contexts do cover the entire System F syntax. Typing contexts is performed by the $M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2$ relation: "Given an expression e that has type σ_1 under typing environment Γ_1, then the resulting expression $M[e]$ has type σ_2 under typing environment Γ_2". We will elaborate further on contextual equivalence in Appendix B.2.

$$
\begin{array}{llll}
e & ::= & x \mid K \mid e_1\, e_2 \mid \lambda x : \sigma . e \mid e\, \sigma \mid \Lambda a . e & \textit{Expression} \\
 & \mid & \textit{undefined} \mid e_q & \\
 & \mid & \mathbf{case}\, \overline{\overline{\pi_F}_i : \overline{\psi_F} \to e_i}^{\,i} \mid \textit{True} \mid \textit{False} & \\
v & ::= & \lambda x : \sigma . e \mid \Lambda a . v \mid K\, \overline{e} & \textit{Value} \\
 & \mid & \mathbf{case}\, \overline{\overline{\pi_F}_i : \overline{\psi_F} \to e_i}^{\,i} & \\
\dot{t} & ::= & [] \mid \lambda x : \sigma . \dot{t} \mid \dot{t}\, \sigma \mid \Lambda a . \dot{t} \mid \lambda e_1 . e_2 & \textit{Expr. Wrapper} \\
M & ::= & [] \mid \lambda x : \sigma . M \mid M\, e \mid e\, M & \textit{Context} \\
 & \mid & \Lambda a . M \mid M\, \sigma & \\
\textit{arg}_F & ::= & e \mid \sigma & \textit{Argument} \\
\pi_F & ::= & x : \sigma \mid @a \mid K\, \overline{\pi_F} & \textit{Pattern} \\
\psi_F & ::= & \sigma \mid @a & \textit{Arg. descriptor}
\end{array}
$$

$\boxed{\Gamma \vdash_{tm} e : \sigma}$ *(System F Term Typing)*

$$
\textsc{tTm-Var}\ \dfrac{(x : \sigma) \in \Gamma \qquad \vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} x : \sigma}
\qquad
\textsc{tTm-Con}\ \dfrac{K : \overline{a} \,;\, \overline{\sigma} \,;\, T \in \Gamma}{\Gamma \vdash_{tm} K : \forall\, \overline{a} . \overline{\sigma} \to T\, \overline{a}}
\qquad
\textsc{tTm-App}\ \dfrac{\Gamma \vdash_{tm} e_1 : \sigma \to \sigma' \quad \Gamma \vdash_{tm} e_2 : \sigma}{\Gamma \vdash_{tm} e_1\, e_2 : \sigma'}
$$

$$
\textsc{tTm-Abs}\ \dfrac{\Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2}{\Gamma \vdash_{tm} \lambda x : \sigma_1 . e : \sigma_1 \to \sigma_2}
\qquad
\textsc{tTm-Tapp}\ \dfrac{\Gamma \vdash_{tm} e : \forall a . \sigma_1 \quad \Gamma \vdash_{ty} \sigma_2}{\Gamma \vdash_{tm} e\, \sigma : [\sigma_2 / a] \sigma_1}
\qquad
\textsc{tTm-Tabs}\ \dfrac{\Gamma, a \vdash_{tm} e : \sigma}{\Gamma \vdash_{tm} \Lambda a . e : \forall a . \sigma}
$$

$$
\textsc{tTm-Undef}\ \dfrac{}{\Gamma \vdash_{tm} \textit{undefined} : \forall\, a . a}
\qquad
\textsc{tTm-True}\ \dfrac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} \textit{True} : \textit{Bool}}
\qquad
\textsc{tTm-False}\ \dfrac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} \textit{False} : \textit{Bool}}
$$

tTm-Seq

$$\frac{}{\Gamma \vdash_{tm} e_q : \forall\, a.\forall\, b.a \to b \to b}$$

tTm-Case

$$\frac{\begin{array}{c}\overline{\Gamma \vdash^{P} \overline{\pi_F}_i : \overline{\psi_F}; \Delta}^{\,i} \\ \overline{\Gamma, \Delta \vdash_{tm} e_i : \sigma_1}^{\,i} \\ \mathit{type}\,(\overline{\psi_F}; \sigma_1\ \sigma_2)\end{array}}{\Gamma \vdash_{tm} \textbf{case}\, \overline{\overline{\pi_F}_i : \overline{\psi_F} \to e_i}^{\,i} : \sigma_2}$$

$\boxed{\Gamma \vdash^{P} \overline{\pi_F} : \overline{\psi_F}; \Delta}$ *(System F Pattern Typing)*

FPat-Empty

$$\frac{}{\Gamma \vdash^{P} \bullet : \bullet; \bullet}$$

FPat-Var

$$\frac{\Gamma, x : \sigma \vdash^{P} \overline{\pi_F} : \overline{\psi_F}; \Delta}{\Gamma \vdash^{P} x : \sigma, \overline{\pi_F} : \sigma, \overline{\psi_F}; x : \sigma, \Delta}$$

FPat-TyVar

$$\frac{\Gamma, a \vdash^{P} \overline{\pi_F} : \overline{\psi_F}; \Delta}{\Gamma \vdash^{P} @a, \overline{\pi_F} : @a, \overline{\psi_F}; a, \Delta}$$

FPat-Con

$$\frac{\begin{array}{c}K\ :\ \overline{a}_0\ ;\ \overline{\sigma}_0\ ;\ T\ \in\ \Gamma \\ \Gamma \vdash^{P} \overline{\pi_F} : [\overline{\sigma}_1/\overline{a}_0]\overline{\sigma}_0; \Delta_1 \\ \Gamma, \Delta_1 \vdash^{P} \overline{\pi_F}' : \overline{\psi_F}; \Delta_2\end{array}}{\Gamma \vdash^{P} (K\, \overline{\pi_F}), \overline{\pi_F}'\ :\ T\, \overline{\sigma}_1, \overline{\psi_F}; \Delta_1, \Delta_2}$$

$\boxed{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2}$ *(System F Context Typing)*

FCtx-Hole

$$\frac{}{[] : \Gamma; \sigma \mapsto \Gamma; \sigma}$$

FCtx-Abs

$$\frac{M : \Gamma_1; \sigma_2 \mapsto \Gamma_2, x : \sigma_1; \sigma_3}{\lambda x : \sigma_1.M : \Gamma_1; \sigma_2 \mapsto \Gamma_2; \sigma_1 \to \sigma_3}$$

FCtx-AppR

$$\frac{\begin{array}{c}M_1 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2 \to \sigma_3 \\ \Gamma_2 \vdash_{tm} e_2 : \sigma_2\end{array}}{M_1\, e_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3}$$

FCtx-AppL

$$\frac{\begin{array}{c}\Gamma_2 \vdash_{tm} e_1 : \sigma_2 \to \sigma_3 \\ M_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_2\end{array}}{e_1\, M_2 : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3}$$

FCtx-TyAbs

$$\frac{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2, a; \sigma_2}{\Lambda a.M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \forall\, a.\sigma_2}$$

FCtx-TyApp

$$\frac{M : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \forall\, a.\sigma_2}{M\, \sigma : \Gamma_1; \sigma_1 \mapsto \Gamma_2; [\sigma/a]\sigma_2}$$

FCtx-Case

$$\frac{\begin{array}{c}\overline{\Gamma_1 \vdash^{P} \overline{\pi_F}_i : \overline{\psi_F}; \Delta}^{\,i} \\ \overline{M_i : \Gamma_1, \Delta; \sigma_1 \mapsto \Gamma_2; \sigma_2}^{\,i} \\ \mathit{type}\,(\overline{\psi_F}; \sigma_2\ \sigma_3)\end{array}}{\textbf{case}\, \overline{\overline{\pi_F}_i : \overline{\psi_F} \to M_i}^{\,i} : \Gamma_1; \sigma_1 \mapsto \Gamma_2; \sigma_3}$$

Evaluation for our System F target language is largely standard and defined below. Note that, following GHC, our target language evaluates inside type abstractions (rule FEval-TyAbs). Because of this, a type abstraction $\Lambda a.e$ is a value if and only if e is a value. A more extensive discussion can be found in [13, Appendix A.3].

$\boxed{\mathit{match}\,(\overline{\pi_F}_1 \rightarrow e_1; e_2 : \sigma_2) \hookrightarrow \overline{\pi_F}_2 \rightarrow e_1'}$ *(Core Pattern Matching)*

$$\textsc{FMatch-Var}\quad \frac{}{\mathit{match}\,(x : \sigma, \overline{\pi_F} \rightarrow e_1; e_2 : \sigma) \hookrightarrow \overline{\pi_F} \rightarrow [e_2/x]e_1}$$

$$\textsc{FMatch-Con}\quad \frac{\begin{array}{c}\sigma_2 = \overline{\psi_F}_1 \rightarrow \sigma_2' \\ e_2 \hookrightarrow^{\Downarrow} K\,\overline{e} \\ (\mathbf{case}\,\overline{\pi_F}_1 : \overline{\psi_F}_1 \rightarrow e_1)\,\overline{e} \hookrightarrow^{\Downarrow} v\end{array}}{\mathit{match}\,((K\,\overline{\pi_F}_1), \overline{\pi_F}_2 \rightarrow e_1; e_2 : \sigma_2) \hookrightarrow \overline{\pi_F}_2 \rightarrow v}$$

$\boxed{e_1 \hookrightarrow e_2}$ *(Core Evaluation)*

$$\textsc{FEval-App}\quad \frac{e_1 \hookrightarrow e_1'}{e_1\,e_2 \hookrightarrow e_1'\,e_2} \qquad \textsc{FEval-AppAbs}\quad \frac{}{(\lambda x : \sigma.e_1)\,e_2 \hookrightarrow [e_2/x]e_1} \qquad \textsc{FEval-Seq}\quad \frac{e_1 \hookrightarrow e_1'}{e_q\,e_1\,e_2 \hookrightarrow e_q\,e_1'\,e_2}$$

$$\textsc{FEval-SeqVal}\quad \frac{}{e_q\,v_1\,e_2 \hookrightarrow e_2} \qquad \textsc{FEval-CaseEmpty}\quad \frac{}{\mathbf{case}\,\overline{\bullet : \bullet \rightarrow e_i}^{\,i} \hookrightarrow e_0}$$

$$\textsc{FEval-CaseMatch}\quad \frac{\forall j \in \mathit{pwhere}(\mathit{match}\,(\overline{\pi_F}_j \rightarrow e_j; e_2 : \sigma) \hookrightarrow \overline{\pi_F}_j' \rightarrow e_j')}{(\mathbf{case}\,\overline{\overline{\pi_F}_i : \sigma, \overline{\psi_F} \rightarrow e_i}^{\,i<p})\,e_2 \hookrightarrow \mathbf{case}\,\overline{\overline{\pi_F}_j' : \overline{\psi_F} \rightarrow e_j'}^{\,j<w}} \qquad \textsc{FEval-TyAbs}\quad \frac{e \hookrightarrow e'}{\Lambda a.e \hookrightarrow \Lambda a.e'}$$

$$\textsc{FEval-TyApp}\quad \frac{e_1 \hookrightarrow e_1'}{e_1\,\sigma \hookrightarrow e_1'\,\sigma} \qquad \textsc{FEval-TyAppAbs}\quad \frac{}{(\Lambda a.v_1)\,\sigma \hookrightarrow [\sigma/a]v_1} \qquad \textsc{FEval-Undef}\quad \frac{}{\mathit{undefined} \hookrightarrow \mathit{undefined}}$$

$$\textsc{FEval-TyAbsCase}\quad \frac{}{(\mathbf{case}\,\overline{@a_i, \overline{\pi_F}_i : @a, \overline{\psi_F} \rightarrow e_i}^{\,i})\,\sigma \hookrightarrow \mathbf{case}\,\overline{[\sigma/a]\overline{\pi_F}_i : [\sigma/a]\overline{\psi_F} \rightarrow [\sigma/a]e_i}^{\,i}}$$

$\boxed{e \hookrightarrow^{\Downarrow} v}$ *(Big Step Evaluation)*

FEvalBigStep-Step
$$\frac{e \hookrightarrow e' \qquad e' \hookrightarrow^{\Downarrow} v}{e \hookrightarrow^{\Downarrow} v}$$

FEvalBigStep-Done
$$\frac{}{v \hookrightarrow^{\Downarrow} v}$$

A.2.1 Translation from the Mixed Polymorphic λ-calculus

$\boxed{\Gamma \vdash^{H} h \Rightarrow \sigma \rightsquigarrow e}$ *(Head Type Synthesis)*

H-Var
$$\frac{x \ : \ \sigma \ \in \ \Gamma}{\Gamma \vdash^{H} x \Rightarrow \sigma \rightsquigarrow x}$$

H-Con
$$\frac{K \ : \ \overline{a} \ ; \ \overline{\sigma} \ ; \ T \ \in \ \Gamma}{\Gamma \vdash^{H} K \Rightarrow \forall\, \overline{a}.\overline{\sigma} \to T\,\overline{a} \rightsquigarrow K}$$

H-Ann
$$\frac{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow e}{\Gamma \vdash^{H} e : \sigma \Rightarrow \sigma \rightsquigarrow e}$$

H-Undef
$$\frac{}{\Gamma \vdash^{H} \mathit{undefined} \Rightarrow \forall\, a.a \rightsquigarrow \mathit{undefined}}$$

H-Seq
$$\frac{}{\Gamma \vdash^{H} e_q \Rightarrow \forall\, a.\forall\, b.a \to b \to b \rightsquigarrow e_q}$$

H-Inf
$$\frac{\Gamma \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e}{\Gamma \vdash^{H} e \Rightarrow \eta^{\epsilon} \rightsquigarrow e}$$

$\boxed{\Gamma \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e}$ *(Term Type Synthesis)*

Tm-InfAbs
$$\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \eta_2^{\epsilon} \rightsquigarrow e_1}{\Gamma \vdash \lambda x.e \Rightarrow \tau_1 \to \eta_2^{\epsilon} \rightsquigarrow \lambda x : \sigma_1.e_1}$$

Tm-InfTyAbs
$$\frac{\begin{array}{c}\Gamma, a \vdash e \Rightarrow \eta_1^{\epsilon} \rightsquigarrow e \\ \Gamma \vdash \forall\, a.\eta_1^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \eta_2^{\epsilon} \rightsquigarrow \dot{t}\end{array}}{\Gamma \vdash \Lambda a.e \Rightarrow \eta_2^{\epsilon} \rightsquigarrow \dot{t}[\Lambda a.e]}$$

Tm-InfApp
$$\frac{\begin{array}{c}\Gamma \vdash^{H} h \Rightarrow \sigma \rightsquigarrow e \\ \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F} \\ \Gamma \vdash \sigma' \xrightarrow{\mathit{inst}\ \delta} \eta^{\epsilon} \rightsquigarrow \dot{t}\end{array}}{\Gamma \vdash h\,\overline{arg} \Rightarrow \eta^{\epsilon} \rightsquigarrow \dot{t}[e\,\overline{arg_F}]}$$

Tm-InfLet
$$\frac{\begin{array}{c}\Gamma \vdash \mathit{decl} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_1 \\ \Gamma' \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e_2\end{array}}{\Gamma \vdash \mathbf{let}\ \mathit{decl}\ \mathbf{in}\ e \Rightarrow \eta^{\epsilon} \rightsquigarrow (\lambda x : \sigma.e_2)\, e_1}$$

Tm-InfTrue
$$\frac{}{\Gamma \vdash \mathbf{true} \Rightarrow \mathbf{Bool} \rightsquigarrow \mathit{True}}$$

Tm-InfFalse
$$\frac{}{\Gamma \vdash \mathbf{false} \Rightarrow \mathbf{Bool} \rightsquigarrow \mathit{False}}$$

$\boxed{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow e}$ *(Term Type Scheme Checking)*

Tm-CheckAbs

$$\frac{\Gamma \vdash \sigma \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_1 \to \sigma_2 ; \Gamma_1 \rightsquigarrow \dot{t} \qquad \Gamma_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2 \rightsquigarrow e_1}{\Gamma \vdash \lambda x.e \Leftarrow \sigma \rightsquigarrow \dot{t}[\lambda x : \sigma_1 . e_1]}$$

Tm-CheckTyAbs

$$\frac{\sigma = \forall \overline{\{a\}}.\forall a.\sigma' \qquad \Gamma, \overline{a}, a \vdash e \Leftarrow \sigma' \rightsquigarrow e}{\Gamma \vdash \Lambda a.e \Leftarrow \sigma \rightsquigarrow \Lambda \overline{a}.\Lambda a.e}$$

Tm-CheckLet

$$\frac{\Gamma \vdash decl \Rightarrow \Gamma' \rightsquigarrow x : \sigma_1 = e_1 \qquad \Gamma' \vdash e \Leftarrow \sigma \rightsquigarrow e_2}{\Gamma \vdash \mathbf{let}\ decl\ \mathbf{in}\ e \Leftarrow \sigma \rightsquigarrow (\lambda x : \sigma_1 . e_2)\, e_1}$$

Tm-CheckInf

$$\frac{\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho ; \Gamma_1 \rightsquigarrow \dot{t}_1 \qquad \Gamma_1 \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e \qquad \Gamma_1 \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2 \qquad e \neq \lambda, \Lambda, let}{\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow \dot{t}_1[\dot{t}_2[e]]}$$

$\boxed{\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}}$ *(Argument Type Checking)*

Arg-Empty

$$\frac{}{\Gamma \vdash^{A} \bullet \Leftarrow \sigma \Rightarrow \sigma \rightsquigarrow \bullet}$$

Arg-App

$$\frac{\Gamma \vdash e \Leftarrow \sigma_1 \rightsquigarrow e \qquad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2 \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma' \rightsquigarrow e, \overline{arg_F}}$$

Arg-Inst

$$\frac{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \sigma_2' \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F} \qquad \sigma_2' = [\tau_1 / a]\, \sigma_2}{\Gamma \vdash^{A} e, \overline{arg} \Leftarrow \forall a.\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}$$

Arg-TyApp

$$\frac{\Gamma \vdash^{A} \overline{arg} \Leftarrow [\sigma_1 / a]\, \sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}{\Gamma \vdash^{A} @\sigma_1, \overline{arg} \Leftarrow \forall a.\sigma_2 \Rightarrow \sigma_3 \rightsquigarrow \sigma_1, \overline{arg_F}}$$

Arg-InfInst

$$\frac{\sigma = \forall \{a\}.\sigma_2 \qquad \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_2' \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F} \qquad \sigma_2' = [\tau_1 / a]\, \sigma_2}{\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma_3 \rightsquigarrow \overline{arg_F}}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}}$ *(Type Instantiation)*

InstT-SInst

$$\frac{}{\Gamma \vdash \rho \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow []}$$

InstT-SForall

$$\frac{\Gamma \vdash [\tau / a]\, \sigma \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall a.\sigma \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow \lambda e.(\dot{t}[e\, \sigma])}$$

InstT-SInfForall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \overset{\mathit{inst}\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\{a\}.\sigma \overset{\mathit{inst}\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow \lambda e.(\dot{t}[e\,\sigma])}$$

InstT-Mono

$$\frac{}{\Gamma \vdash \tau \xrightarrow{\mathit{inst}\ \mathcal{D}} \tau \rightsquigarrow []}$$

InstT-Function

$$\frac{\Gamma \vdash \sigma_2 \xrightarrow{\mathit{inst}\ \mathcal{D}} \rho_2 \rightsquigarrow \dot{t}}{\Gamma \vdash \sigma_1 \to \sigma_2 \xrightarrow{\mathit{inst}\ \mathcal{D}} \sigma_1 \to \rho_2 \rightsquigarrow \lambda e.\lambda x : \sigma_1.(\dot{t}[e\,x])}$$

InstT-Forall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\mathit{inst}\ \mathcal{D}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\mathit{inst}\ \mathcal{D}} \rho \rightsquigarrow \lambda e.(\dot{t}[e\,\sigma])}$$

InstT-InfForall

$$\frac{\Gamma \vdash [\tau/a]\,\sigma \xrightarrow{\mathit{inst}\ \mathcal{D}} \rho \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\{a\}.\sigma \xrightarrow{\mathit{inst}\ \mathcal{D}} \rho \rightsquigarrow \lambda e.(\dot{t}[e\,\sigma])}$$

$\boxed{\Gamma \vdash \sigma \xrightarrow{\mathit{skol}\ \delta} \rho; \Gamma' \rightsquigarrow \dot{t}}$ *(Type Skolemisation)*

SkolT-SInst

$$\frac{}{\Gamma \vdash \rho \overset{\mathit{skol}\ \mathcal{S}}{\dashrightarrow} \rho; \Gamma \rightsquigarrow []}$$

SkolT-SForall

$$\frac{\Gamma, a \vdash \sigma \overset{\mathit{skol}\ \mathcal{S}}{\dashrightarrow} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \overset{\mathit{skol}\ \mathcal{S}}{\dashrightarrow} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-SInfForall

$$\frac{\Gamma, a \vdash \sigma \overset{\mathit{skol}\ \mathcal{S}}{\dashrightarrow} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\{a\}.\sigma \overset{\mathit{skol}\ \mathcal{S}}{\dashrightarrow} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-Mono

$$\frac{}{\Gamma \vdash \tau \xrightarrow{\mathit{skol}\ \mathcal{D}} \tau; \Gamma \rightsquigarrow []}$$

SkolT-Function

$$\frac{\Gamma \vdash \sigma_2 \xrightarrow{\mathit{skol}\ \mathcal{D}} \rho_2; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \sigma_1 \to \sigma_2 \xrightarrow{\mathit{skol}\ \mathcal{D}} \sigma_1 \to \rho_2; \Gamma_1 \rightsquigarrow \lambda e.\lambda x : \sigma_1.(\dot{t}[e\,x])}$$

SkolT-Forall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\mathit{skol}\ \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\, a.\sigma \xrightarrow{\mathit{skol}\ \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

SkolT-InfForall

$$\frac{\Gamma, a \vdash \sigma \xrightarrow{\mathit{skol}\ \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \dot{t}}{\Gamma \vdash \forall\{a\}.\sigma \xrightarrow{\mathit{skol}\ \mathcal{D}} \rho; \Gamma_1 \rightsquigarrow \Lambda a.\dot{t}}$$

$\boxed{\Gamma \vdash \mathit{decl} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e}$ *(Declaration Checking)*

Decl-NoAnnSingle

$$\frac{\begin{array}{c}\Gamma \vdash^{P} \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \\ \Gamma, \Delta \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e \\ \mathit{type}\,(\overline{\psi}; \eta^{\epsilon}\ \sigma) \quad \overline{a} = \mathbf{fv}\,(\sigma) \setminus \mathbf{dom}\,(\Gamma)\end{array}}{\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \forall\,\overline{\{a\}}.\sigma \rightsquigarrow x : \forall\,\overline{\{a\}}.\sigma = \mathbf{case}\,\overline{\pi_F} : \overline{\psi_F} \to e}$$

Decl-NoAnnMulti

$$\frac{\begin{array}{c} j > 1 \quad \overline{\Gamma \vdash^P \overline{\pi}_j \Rightarrow \overline{\psi}; \Delta_j \rightsquigarrow \overline{\pi_F}_j : \overline{\psi_F}}^{j} \\ \overline{\Gamma, \Delta_j \vdash e_j \Rightarrow \eta_j^{\epsilon} \rightsquigarrow e_j}^{j} \\ \overline{\Gamma, \Delta_j \vdash \eta_j^{\epsilon} \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_j}^{j} \quad type\,(\overline{\psi}; \rho\ \sigma) \\ \overline{a} = \mathbf{fv}\,(\sigma) \setminus \mathbf{dom}\,(\Gamma) \quad \sigma' = \forall\, \overline{\{a\}}.\sigma \end{array}}{\Gamma \vdash \overline{x\, \overline{\pi}_j = e_j}^{j} \Rightarrow \Gamma, x : \sigma' \rightsquigarrow x : \sigma' = \mathbf{case}\, \overline{\overline{\pi_F}_j : \overline{\psi_F} \rightarrow \dot{t}_j[e_j]}^{j}}$$

Decl-Ann

$$\frac{\overline{\Gamma \vdash^P \overline{\pi}_j \Leftarrow \sigma \Rightarrow \sigma'_j; \Delta_j \rightsquigarrow \overline{\pi_F}_j : \overline{\psi_F}}^{j} \quad \overline{\Gamma, \Delta_j \vdash e_j \Leftarrow \sigma'_j \rightsquigarrow e_j}^{j}}{\Gamma \vdash x : \sigma; \overline{x\, \overline{\pi}_j = e_j}^{j} \Rightarrow \Gamma, x : \sigma \rightsquigarrow x : \sigma = \mathbf{case}\, \overline{\overline{\pi_F}_j : \overline{\psi_F} \rightarrow e_j}^{j}}$$

$$\boxed{\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}$$

(Pattern Synthesis)

Pat-InfEmpty

$$\frac{}{\Gamma \vdash^P \bullet \Rightarrow \bullet; \bullet \rightsquigarrow \bullet : \bullet}$$

Pat-InfVar

$$\frac{\Gamma, x : \tau_1 \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P x, \overline{\pi} \Rightarrow \tau_1, \overline{\psi}; x : \tau_1, \Delta \rightsquigarrow x : \sigma_1, \overline{\pi_F} : \sigma_1, \overline{\psi_F}}$$

Pat-InfCon

$$\frac{\begin{array}{c} K \ : \ \overline{a}_0 \ ; \ \overline{\sigma}_0 \ ; \ T \ \in \ \Gamma \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0 / \overline{a}_0]\,(\overline{\sigma}_0 \rightarrow T\, \overline{a}_0) \Rightarrow T\, \overline{\tau}; \Delta_1 \rightsquigarrow \overline{\pi_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}; \Delta_2 \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}_2 \end{array}}{\Gamma \vdash^P (K\, @\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Rightarrow T\, \overline{\tau}, \overline{\psi}; \Delta_1, \Delta_2 \rightsquigarrow (K\, \overline{\pi_F}_1), \overline{\pi_F}_2 : T\, \overline{\sigma}, \overline{\psi_F}_2}$$

Pat-InfTyVar

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P @a, \overline{\pi} \Rightarrow @a, \overline{\psi}; a, \Delta \rightsquigarrow @a, \overline{\pi_F} : @a, \overline{\psi_F}}$$

$$\boxed{\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}$$

(Pattern Checking)

Pat-CheckEmpty

$$\frac{}{\Gamma \vdash^P \bullet \Leftarrow \sigma \Rightarrow \sigma; \bullet \rightsquigarrow \bullet : \bullet}$$

Pat-CheckVar

$$\frac{\Gamma, x : \sigma_1 \vdash^P \overline{\pi} \Leftarrow \sigma_2 \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P x, \overline{\pi} \Leftarrow \sigma_1 \rightarrow \sigma_2 \Rightarrow \sigma'; x : \sigma_1, \Delta \rightsquigarrow x : \sigma_1, \overline{\pi_F} : \sigma_1, \overline{\psi_F}}$$

Pat-CheckCon

$$\frac{\begin{array}{c} K \;:\; \overline{a}_0 \;;\; \overline{\sigma}_0 \;;\; T \;\in\; \Gamma \quad \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_1 \rightsquigarrow \dot{t} \\ \Gamma \vdash^P \overline{\pi} \Leftarrow [\overline{\sigma}_1, \overline{\tau}_0/\overline{a}_0]\,(\overline{\sigma}_0 \to T\,\overline{a}_0) \Rightarrow \rho_1; \Delta_1 \rightsquigarrow \overline{\pi_F}_1 : \overline{\psi_F}_1 \\ \Gamma, \Delta_1 \vdash^P \overline{\pi}' \Leftarrow \sigma_2 \Rightarrow \sigma_2'; \Delta_2 \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}_2 \end{array}}{\Gamma \vdash^P (K\,@\overline{\sigma}_1\ \overline{\pi}), \overline{\pi}' \Leftarrow \sigma_1 \to \sigma_2 \Rightarrow \sigma_2'; \Delta_1, \Delta_2 \rightsquigarrow (K\,\overline{\pi_F}_1), \overline{\pi_F}_2 : \sigma_1, \overline{\psi_F}_2}$$

Pat-CheckForall

$$\frac{\begin{array}{c} \Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \\ \overline{\pi} \neq \cdot \ \textit{and}\ \overline{\pi} \neq @\sigma, \overline{\pi}' \end{array}}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, a.\sigma \Rightarrow \sigma'; a, \Delta \rightsquigarrow \overline{\pi_F} : @a, \overline{\psi_F}}$$

Pat-CheckTyVar

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow [a/b]\,\sigma_1 \Rightarrow \sigma_2; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}}{\Gamma \vdash^P @a, \overline{\pi} \Leftarrow \forall\, b.\sigma_1 \Rightarrow \sigma_2; a, \Delta \rightsquigarrow \overline{\pi_F} : @a, \overline{\psi_F}}$$

Pat-CheckInfForall

$$\frac{\Gamma, a \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F} \quad \overline{\pi} \neq \cdot}{\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\, \{a\}.\sigma \Rightarrow \sigma'; a, \Delta \rightsquigarrow \overline{\pi_F} : @\overline{a}, \overline{\psi_F}}$$

A.3 $\lambda_{\textsf{TC}}$ Additional Definitions

A.3.1 Syntax

$$
\begin{array}{lcll}
pgm & ::= & e \mid cls; pgm \mid inst; pgm & \textit{spgm} \\
cls & ::= & \textbf{class}\ \overline{TC_i\ a} \Rightarrow TC\, a\, \textbf{where}\ \{m : \sigma\} & \textit{class decl.} \\
inst & ::= & \textbf{instance}\ \overline{Q} \Rightarrow TC\, \tau\, \textbf{where}\ \{m = e\} & \textit{instance decl.} \\
 & & & \\
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid m \mid \lambda x.e \mid e_1\, e_2 \mid \textbf{let}\ x : \sigma = e_1\, \textbf{in}\ e_2 \mid e :: \tau & \textit{term} \\
 & & & \\
\tau & ::= & \mathit{Bool} \mid a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid Q \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme} \\
 & & & \\
Q & ::= & TC\, \tau & \textit{class constraint} \\
C & ::= & \forall \overline{a}.\overline{Q} \Rightarrow Q & \textit{constraint} \\
 & & & \\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \overline{TC_i\ a} \Rightarrow TC\, a : \sigma & \textit{class environment} \\
P & ::= & \bullet \mid P, (D : C).m \mapsto \Gamma : e & \textit{program context} \\
M & ::= & [\bullet] \mid \lambda x.M \mid M\, e \mid e\, M \mid M :: \tau & \textit{evaluation context} \\
 & & \mid \textbf{let}\ x : \sigma = M\, \textbf{in}\ e \mid \textbf{let}\ x : \sigma = e\, \textbf{in}\ M &
\end{array}
$$

A.3.2 $\lambda_{\textsf{TC}}$ Judgments and Elaboration

$\lambda_{\textsf{TC}}$ Type & Constraint Well-Formedness

$\boxed{\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma}$ *($\lambda_{\textbf{TC}}$ Class Constraint Well-Formedness)*

$$
\textsc{osQT-TC}\ \frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma' \\ \Gamma_C = \Gamma_{C1}, m : \overline{Q}_i \Rightarrow TC\, a : \sigma, \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{\Gamma_C; \Gamma \vdash_Q TC\, \tau \rightsquigarrow [\sigma'/a]\{m : \sigma\}}
$$

$\boxed{\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma}$ *($\lambda_{\textbf{TC}}$ Constraint Well-Formedness)*

osCT-abs

$$\frac{\begin{array}{c}\overline{\Gamma_C ; \Gamma, \overline{a}_j \vdash_Q Q_i \rightsquigarrow \sigma_i}^{\,i} \\ \Gamma_C ; \Gamma, \overline{a}_j \vdash_Q Q \rightsquigarrow \sigma \\ \overline{a}_j \notin \Gamma\end{array}}{\Gamma_C ; \Gamma \vdash_C \forall \overline{a}_j . \overline{Q}_i \Rightarrow Q \rightsquigarrow \forall \overline{a}_j . \overline{\sigma}_i \to \sigma}$$

$\boxed{\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}$ *($\lambda_{\textbf{TC}}$ Type Well-Formedness)*

osTyT-bool

$$\frac{}{\Gamma_C ; \Gamma \vdash_{ty} \mathit{Bool} \rightsquigarrow \mathit{Bool}}$$

osTyT-var

$$\frac{a \in \Gamma}{\Gamma_C ; \Gamma \vdash_{ty} a \rightsquigarrow a}$$

osTyT-arrow

$$\frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma_1 \\ \Gamma_C ; \Gamma \vdash_{ty} \tau_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2}$$

osTyT-qual

$$\frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash_Q Q \rightsquigarrow \sigma_1 \\ \Gamma_C ; \Gamma \vdash_{ty} \rho \rightsquigarrow \sigma_2\end{array}}{\Gamma_C ; \Gamma \vdash_{ty} Q \Rightarrow \rho \rightsquigarrow \sigma_1 \to \sigma_2}$$

osTyT-scheme

$$\frac{\begin{array}{c}a \notin \Gamma \\ \Gamma_C ; \Gamma, a \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C ; \Gamma \vdash_{ty} \forall a . \sigma \rightsquigarrow \forall a . \sigma}$$

λ_{TC} Term Typing

$\boxed{P ; \Gamma_C ; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}$ *($\lambda_{\textbf{TC}}$ Term Inference)*

osTm-infT-true

$$\frac{\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma}{P ; \Gamma_C ; \Gamma \vdash_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}}$$

osTm-infT-false

$$\frac{\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma}{P ; \Gamma_C ; \Gamma \vdash_{tm} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}}$$

osTm-infT-let

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_k \\ \overline{\Gamma_C; \Gamma \vdash_Q Q_k \rightsquigarrow \sigma_k}^{k} \\ \Gamma_C; \Gamma \vdash_{ty} \forall \overline{a}_j . \overline{Q}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j . \overline{\sigma}_k \rightarrow \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \\ P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{Q}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{Q}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ e = \mathbf{let} \ \ x : \forall \overline{a}_j . \overline{\sigma}_k \rightarrow \sigma = \Lambda \overline{a}_j . \lambda \, \overline{\delta_k : \sigma_k}^{k} . e_1 \ \mathbf{in} \ \ e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} \mathbf{let} \ \ x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in} \ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e}$$

osTm-infT-ArrE

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma \vdash_{tm} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} e_1 \, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1 \, e_2}$$

osTm-infT-Ann

$$\frac{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm} e :: \tau \Rightarrow \tau \rightsquigarrow e}$$

$$\boxed{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

($\lambda_{\mathbf{TC}}$ Term Checking)

osTm-checkT-var

$$\frac{\begin{array}{c} (x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau) \in \Gamma \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau) \\ \overline{P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j / \overline{a}_j] Q_i \rightsquigarrow e_i}^{i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} x \Leftarrow [\overline{\tau}_j / \overline{a}_j] \tau \rightsquigarrow x \, \overline{\sigma}_j \, \overline{e}_i}$$

osTm-checkT-meth

$$\frac{\begin{array}{c} (m : \overline{Q}'_k \Rightarrow TC \, a : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a . \overline{Q}_i \Rightarrow \tau') \\ P; \Gamma_C; \Gamma \vDash TC \, \tau \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \\ \overline{P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j / \overline{a}_j][\tau / a] Q_i \rightsquigarrow e_i}^{i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j / \overline{a}_j][\tau / a] \tau' \rightsquigarrow e.m \, \overline{\sigma}_j \, \overline{e}_i}$$

osTm-checkT-ArrI

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ P; \Gamma_C; \Gamma, x : \tau_1 \vdash_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} \lambda x.e \Leftarrow \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \sigma.e}$$

osTm-checkT-Inf

$$\frac{P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

$\boxed{\Gamma_C \vdash_{cls} cls : \Gamma_C{}'}$ *(Class Decl Typing)*

osClsT-cls

$$\frac{\begin{array}{c} m \notin \mathbf{dom}(\Gamma_C) \\ \mathbf{closure}(\Gamma_C; \overline{Q}_k) = \overline{Q}_p \\ \Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_j.\overline{Q}_p \Rightarrow \tau \rightsquigarrow \sigma \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_p \Rightarrow \tau) \\ \overline{\Gamma_C; \bullet, a \vdash_Q TC_i\, a \rightsquigarrow \sigma_i}^{\,i} \\ \nexists TC' : (m : \overline{Q}'_m \Rightarrow TC'\ b : \sigma') \in \Gamma_C \\ \nexists m' : (m' : \overline{Q}'_m \Rightarrow TC\ a : \sigma') \in \Gamma_C \\ \Gamma_C{}' = m : \overline{TC_i\ a} \Rightarrow TC\, a : \forall \overline{a}_j.\overline{Q}_p \Rightarrow \tau \end{array}}{\Gamma_C \vdash_{cls} \mathbf{class}\ \ \overline{TC_i\ a} \Rightarrow TC\, a\, \mathbf{where}\ \ \{m : \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau\} : \Gamma_C{}'}$$

$\boxed{P; \Gamma_C \vdash_{inst} inst : P'}$ *(Instance Decl Typing)*

osInstT-inst

$$\frac{\begin{array}{c} (m : \overline{Q}'_i \Rightarrow TC\, a : \forall \overline{a}_j.\overline{Q}'_y \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau \rightsquigarrow \sigma \\ \mathbf{closure}(\Gamma_C; \overline{Q}_p) = \overline{Q}_q \\ \mathbf{unambig}(\forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\, \tau) \\ \overline{\Gamma_C; \bullet, \overline{b}_k \vdash_Q Q_q \rightsquigarrow \sigma_q}^{\,q} \\ \overline{P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q \vDash [\tau/a]Q'_i \rightsquigarrow e_i}^{\,i} \\ P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}'_y : [\tau/a]\overline{Q}'_y \vdash_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ D\ \mathbf{fresh} \\ \overline{\delta}_q\ \mathbf{fresh} \qquad \overline{\delta}'_y\ \mathbf{fresh} \\ (D' : \forall \overline{b}'_w.\overline{Q}'_n \Rightarrow TC\, \tau_2).m' \mapsto \Gamma' : e' \notin P\ \ \mathit{where}\ [\overline{\tau}'_w/\overline{b}'_w]\tau_2 = [\overline{\tau}'_k/\overline{b}_k]\tau \\ P' = (D : \forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\, \tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}'_y : [\tau/a]\overline{Q}'_y : e \end{array}}{P; \Gamma_C \vdash_{inst} \mathbf{instance}\ \ \overline{Q}_p \Rightarrow TC\, \tau\, \mathbf{where}\ \ \{m = e\} : P'}$$

$$\boxed{P; \Gamma_C \vdash_{pgm} pgm : \tau; P'; {\Gamma_C}' \rightsquigarrow e}$$

($\lambda_{\mathbf{TC}}$ Program Typing)

osPgmT-cls

$$\frac{\Gamma_C \vdash_{cls} cls : {\Gamma_C}' \qquad P; \Gamma_C, {\Gamma_C}' \vdash_{pgm} pgm : \tau; P'; {\Gamma_C}'' \rightsquigarrow e}{P; \Gamma_C \vdash_{pgm} cls; pgm : \tau; P'; {\Gamma_C}', {\Gamma_C}'' \rightsquigarrow e}$$

osPgmT-inst

$$\frac{P; \Gamma_C \vdash_{inst} inst : P' \qquad P, P'; \Gamma_C \vdash_{pgm} pgm : \tau; P''; {\Gamma_C}' \rightsquigarrow e}{P; \Gamma_C \vdash_{pgm} inst; pgm : \tau; P', P''; {\Gamma_C}' \rightsquigarrow e}$$

osPgmT-expr

$$\frac{P; \Gamma_C; \bullet \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C \vdash_{pgm} e : \tau; \bullet; \bullet \rightsquigarrow e}$$

$$\boxed{\mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_j}$$

(Closure over Superclass Relation)

osClosure-empty

$$\frac{}{\mathbf{closure}(\Gamma_C; \bullet) = \bullet}$$

osClosure-TC

$$\frac{(m : \overline{Q}_m \Rightarrow TC\, a : \sigma) \in \Gamma_C \qquad \mathbf{closure}(\Gamma_C; \overline{Q}_i, \overline{Q}_m) = \overline{Q}_j}{\mathbf{closure}(\Gamma_C; \overline{Q}_i, TC\, a) = \overline{Q}_j, TC\, a}$$

$$\boxed{\mathbf{unambig}(\sigma)}$$

(Unambiguity for Type Schemes)

osUnambig-scheme

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau)}$$

$$\boxed{\mathbf{unambig}(C)}$$

(Unambiguity for Constraints)

osUnambig-constraint

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\, \tau)}$$

Constraint Proving

$\boxed{P;\Gamma_C;\Gamma \vDash Q \rightsquigarrow e}$ *(Constraint Entailment)*

osEntailT-inst

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{Q}'_i \Rightarrow Q').m \mapsto \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}'_i, \overline{b}_k, \overline{\delta}_y : \overline{Q}_y : e, P_2 \\ Q = [\overline{\tau}_j/\overline{a}_j]Q' \\ P_1;\Gamma_C;\bullet,\overline{a}_j,\overline{\delta}_i:\overline{Q}'_i,\overline{b}_k,\overline{\delta}_y:\overline{Q}_y \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e \\ \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma \\ \overline{\Gamma_C;\Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \overline{\Gamma_C;\bullet,\overline{a}_j \vdash_Q Q'_i \rightsquigarrow \sigma'_i}^{i} \\ \overline{\Gamma_C;\bullet,\overline{a}_j,\overline{b}_k \vdash_Q Q_y \rightsquigarrow \sigma''_y}^{y} \\ \overline{P;\Gamma_C;\Gamma \vDash [\overline{\tau}_j/\overline{a}_j]Q'_i \rightsquigarrow e_i}^{i} \end{array}}{P;\Gamma_C;\Gamma \vDash Q \rightsquigarrow (\Lambda\overline{a}_j.\lambda\,\overline{\delta'_i:\sigma'_i}^{i}.\{m = \Lambda\overline{b}_k.\lambda\,\overline{\delta_y:\sigma''_y}^{y}.e\})\,\overline{\sigma}_j\,\overline{e}_i}$$

osEntailT-local

$$\frac{\begin{array}{c} (\delta : Q) \in \Gamma \\ \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma \end{array}}{P;\Gamma_C;\Gamma \vDash Q \rightsquigarrow \delta}$$

$\lambda_{\mathbf{TC}}$ Environment Well-Formedness

$\boxed{\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma}$ *($\lambda_{\mathbf{TC}}$ Environment Well-Formedness)*

osCtxT-empty

$$\frac{}{\vdash_{ctx} \bullet;\bullet;\bullet \rightsquigarrow \bullet}$$

osCtxT-clsEnv

$$\frac{\begin{array}{c} \Gamma_C;\bullet,a \vdash_{ty} \forall\overline{a}_j.\overline{TC_i\,a}^{i} \Rightarrow \tau \rightsquigarrow \sigma \\ \overline{a}_j, a = \mathbf{fv}(\tau) \\ \overline{\Gamma_C;\bullet,a \vdash_Q TC_i\,a \rightsquigarrow \sigma_i}^{i} \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\,b \notin \mathbf{dom}(\Gamma_C) \\ \vdash_{ctx} \bullet;\Gamma_C;\bullet \rightsquigarrow \bullet \end{array}}{\vdash_{ctx} \bullet;\Gamma_C, m : \overline{TC_i\ a} \Rightarrow TC\,a : \forall\overline{a}_j.\overline{TC_i\,a}^{i} \Rightarrow \tau;\bullet \rightsquigarrow \bullet}$$

osCtxT-tyEnvTm

$$\frac{\begin{array}{c} \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \Gamma \end{array}}{\vdash_{ctx} \bullet;\Gamma_C;\Gamma, x : \sigma \rightsquigarrow \Gamma, x : \sigma}$$

osCtxT-tyEnvTy

$$\frac{\begin{array}{c} a \notin \Gamma \\ \vdash_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \Gamma \end{array}}{\vdash_{ctx} \bullet;\Gamma_C;\Gamma, a \rightsquigarrow \Gamma, a}$$

$$\textsc{osCtxT-tyEnvD}\quad \frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash_Q TC\,\tau \rightsquigarrow \sigma \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma\end{array}}{\vdash_{ctx} \bullet ; \Gamma_C ; \Gamma , \delta : TC\,\tau \rightsquigarrow \Gamma , \delta : \sigma}$$

$$\textsc{osCtxT-pgmInst}\quad \frac{\begin{array}{c}\mathbf{unambig}(\forall \bar{b}_j . \overline{Q}_i \Rightarrow TC\,\tau) \\ \Gamma_C ; \bullet \vdash_C \forall \bar{b}_j . \overline{Q}_i \Rightarrow TC\,\tau \rightsquigarrow \forall \bar{b}_j . \overline{\sigma}_i \rightarrow [\sigma / a] \{ m : \forall \bar{a}_k . \overline{\sigma}'_y \rightarrow \sigma' \} \\ (m : \overline{Q}'_m \Rightarrow TC\,a : \forall \bar{a}_k . \overline{Q}'_y \Rightarrow \tau') \in \Gamma_C \\ \Gamma_C ; \bullet , a \vdash_{ty} \forall \bar{a}_k . \overline{Q}'_y \Rightarrow \tau' \rightsquigarrow \forall \bar{a}_k . \overline{\sigma}'_y \rightarrow \sigma' \\ \Gamma_C ; \bullet , \bar{b}_j \vdash_{ty} \tau \rightsquigarrow \sigma \\ P ; \Gamma_C ; \bullet , \bar{b}_j , \bar{\delta}_i : \overline{Q}_i , \bar{a}_k , \bar{\delta}_y : [\tau / a] \overline{Q}'_y \vdash_{tm} e \Leftarrow [\tau / a] \tau' \rightsquigarrow e \\ D \notin \mathbf{dom}(P) \\ (D' : \forall \bar{b}'_k . \overline{Q}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P \\ \mathbf{where} [\overline{\tau}_j / \bar{b}_j] \tau = [\overline{\tau}'_k / \bar{b}'_k] \tau'' \\ \vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma\end{array}}{\vdash_{ctx} P , (D : \forall \bar{b}_j . \overline{Q}_i \Rightarrow TC\,\tau).m \mapsto \bullet , \bar{b}_j , \bar{\delta}_i : \overline{Q}_i , \bar{a}_k , \bar{\delta}_y : [\tau / a] \overline{Q}'_y : e ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma}$$

A.3.3 λ_{TC} Judgments and Elaboration through F_{D}

λ_{TC} Type & Constraint Well-Formedness

$\boxed{\Gamma_C;\Gamma \vdash_Q^M Q \rightsquigarrow Q}$ *($\lambda_{\mathbf{TC}}$ Class Constraint Well-Formedness)*

$$\textsc{osQ-TC}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \\ \Gamma_C = \Gamma_{C1}, m : \overline{Q}_i \Rightarrow TC\, a : \sigma, \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma'\end{array}}{\Gamma_C;\Gamma \vdash_Q TC\, \tau \rightsquigarrow TC\, \sigma}$$

$\boxed{\Gamma_C;\Gamma \vdash_C^M C \rightsquigarrow C}$ *($\lambda_{\mathbf{TC}}$ Constraint Well-Formedness)*

$$\textsc{osC-abs}\quad \frac{\begin{array}{c}\overline{\Gamma_C;\Gamma,\overline{a}_j \vdash_Q Q_i \rightsquigarrow Q_i}^{\,i} \\ \Gamma_C;\Gamma,\overline{a}_j \vdash_Q Q \rightsquigarrow Q \\ \overline{a}_j \notin \Gamma\end{array}}{\Gamma_C;\Gamma \vdash_C \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q \rightsquigarrow \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q}$$

$\boxed{\Gamma_C;\Gamma \vdash_{ty}^M \sigma \rightsquigarrow \sigma}$ *($\lambda_{\mathbf{TC}}$ Type Well-Formedness)*

$$\textsc{osTy-bool}\quad \frac{}{\Gamma_C;\Gamma \vdash_{ty} Bool \rightsquigarrow Bool} \qquad \textsc{osTy-var}\quad \frac{a \in \Gamma}{\Gamma_C;\Gamma \vdash_{ty} a \rightsquigarrow a}$$

$$\textsc{osTy-arrow}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma_1 \\ \Gamma_C;\Gamma \vdash_{ty} \tau_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C;\Gamma \vdash_{ty} \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2} \qquad \textsc{osTy-qual}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_Q Q \rightsquigarrow Q \\ \Gamma_C;\Gamma \vdash_{ty} \rho \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma \vdash_{ty} Q \Rightarrow \rho \rightsquigarrow Q \Rightarrow \sigma}$$

$$\textsc{osTy-scheme}\quad \frac{\begin{array}{c}a \notin \Gamma \\ \Gamma_C;\Gamma, a \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma \vdash_{ty} \forall a.\sigma \rightsquigarrow \forall a.\sigma}$$

$\lambda_{\mathbf{TC}}$ Term Typing

$\boxed{P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e}$ ($\lambda_{\mathbf{TC}}$ *Term Inference*)

$$\textsc{osTm-inf-true}\quad \frac{\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}}$$

$$\textsc{osTm-inf-false}\quad \frac{\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}}$$

$$\textsc{osTm-inf-let}\quad \frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_k \\ \Gamma_C; \Gamma \vdash_{ty} \forall \overline{a}_j . \overline{Q}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j . \overline{Q}_k \Rightarrow \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \\ P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{Q}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{Q}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ e = \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{Q}_k \Rightarrow \sigma = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{Q}_k . e_1 \ \mathbf{in}\ \ e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e}$$

$$\textsc{osTm-inf-ArrE}\quad \frac{\begin{array}{c} P; \Gamma_C; \Gamma \vdash_{tm} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2}$$

$$\textsc{osTm-inf-Ann}\quad \frac{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm} e :: \tau \Rightarrow \tau \rightsquigarrow e}$$

$\boxed{P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e}$ ($\lambda_{\mathbf{TC}}$ *Term Checking*)

$$\textsc{osTm-check-var}\quad \frac{\begin{array}{c} (x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau) \in \Gamma \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau) \\ \overline{P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j / \overline{a}_j] Q_i \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} x \Leftarrow [\overline{\tau}_j / \overline{a}_j] \tau \rightsquigarrow x\, \overline{\sigma}_j\, \overline{d}_i}$$

osTm-check-meth

$$\frac{\begin{array}{c}(m : \overline{Q}'_k \Rightarrow TC\,a : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a . \overline{Q}_i \Rightarrow \tau') \\ P; \Gamma_C; \Gamma \vDash TC\,\tau \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \\ \overline{P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j / \overline{a}_j][\tau / a] Q_i \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma\end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j / \overline{a}_j][\tau / a]\tau' \rightsquigarrow d.m\,\overline{\sigma}_j\,\overline{d}_i}$$

osTm-check-ArrI

$$\frac{\begin{array}{c}x \notin \mathbf{dom}(\Gamma) \\ P; \Gamma_C; \Gamma, x : \tau_1 \vdash_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma\end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} \lambda x . e \Leftarrow \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \sigma . e}$$

osTm-check-Inf

$$\frac{P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

$\boxed{\Gamma_C \vdash^{M}_{cls} cls : \Gamma_C{}'}$ *(Class Decl Typing)*

osCls-cls

$$\frac{\begin{array}{c}m \notin \mathbf{dom}(\Gamma_C) \\ \mathbf{closure}(\Gamma_C; \overline{Q}_k) = \overline{Q}_p \\ \Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_j . \overline{Q}_p \Rightarrow \tau \rightsquigarrow \sigma \\ \mathbf{unambig}(\forall \overline{a}_j, a . \overline{Q}_p \Rightarrow \tau) \\ \overline{\Gamma_C; \bullet, a \vdash_Q TC_i\,a \rightsquigarrow Q_i}^{\,i} \\ \nexists TC' : (m : \overline{Q}'_m \Rightarrow TC'\ b : \sigma') \in \Gamma_C \\ \nexists m' : (m' : \overline{Q}'_m \Rightarrow TC\ a : \sigma') \in \Gamma_C \\ \Gamma_C{}' = m : \overline{TC_i\ a} \Rightarrow TC\,a : \forall \overline{a}_j . \overline{Q}_p \Rightarrow \tau\end{array}}{\Gamma_C \vdash^{M}_{cls} \mathbf{class}\ \ \overline{TC_i\ a} \Rightarrow TC\,a\,\mathbf{where}\ \{m : \forall \overline{a}_j . \overline{Q}_k \Rightarrow \tau\} : \Gamma_C{}'}$$

$\boxed{P; \Gamma_C \vdash^{M}_{inst} inst : P'}$ *(Instance Decl Typing)*

osInst-inst

$$\frac{\begin{array}{c}(m : \overline{Q}'_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{Q}_y \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau \rightsquigarrow \sigma \\ \mathbf{closure}(\Gamma_C; \overline{Q}_p) = \overline{Q}_q \\ \mathbf{unambig}(\forall \overline{b}_k . \overline{Q}_q \Rightarrow TC\, \tau) \\ \overline{\Gamma_C; \bullet, \overline{b}_k \vdash_Q Q_q \rightsquigarrow Q_q}^{\,q} \\ \overline{P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q \vDash [\tau/a]Q'_i \rightsquigarrow d_i}^{\,i} \\ P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{Q}_y \vdash_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ D\ \mathbf{fresh} \\ \overline{\delta}_y\ \mathbf{fresh} \qquad \overline{\delta}_q\ \mathbf{fresh} \\ (D' : \forall \overline{b}'_m . \overline{Q}'_n \Rightarrow TC\, \tau_2).m' \mapsto \Gamma' : e' \notin P\ \mathit{where}\ [\overline{\tau}'_m / \overline{b}'_m]\tau_2 = [\overline{\tau}'_k / \overline{b}_k]\tau \\ P' = (D : \forall \overline{b}_k . \overline{Q}_q \Rightarrow TC\, \tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{Q}_y : e\end{array}}{P; \Gamma_C \vdash^{M}_{inst} \mathbf{instance}\ \overline{Q}_p \Rightarrow TC\, \tau\, \mathbf{where}\ \{m = e\} : P'}$$

$\boxed{P; \Gamma_C \vdash^{M}_{pgm} pgm : \tau; P'; \Gamma_C' \rightsquigarrow e}$ *($\lambda_{\mathbf{TC}}$ Program Typing)*

osPgm-cls

$$\frac{\begin{array}{c}\Gamma_C \vdash^{M}_{cls} cls : \Gamma_C' \\ P; \Gamma_C, \Gamma_C' \vdash^{M}_{pgm} pgm : \tau; P'; \Gamma_C'' \rightsquigarrow e\end{array}}{P; \Gamma_C \vdash^{M}_{pgm} cls; pgm : \tau; P'; \Gamma_C', \Gamma_C'' \rightsquigarrow e}$$

osPgm-inst

$$\frac{\begin{array}{c}P; \Gamma_C \vdash^{M}_{inst} inst : P' \\ P, P'; \Gamma_C \vdash^{M}_{pgm} pgm : \tau; P''; \Gamma_C' \rightsquigarrow e\end{array}}{P; \Gamma_C \vdash^{M}_{pgm} inst; pgm : \tau; P', P''; \Gamma_C' \rightsquigarrow e}$$

osPgm-expr

$$\frac{P; \Gamma_C; \bullet \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C \vdash^{M}_{pgm} e : \tau; \bullet; \bullet \rightsquigarrow e}$$

Constraint Proving

$\boxed{P;\Gamma_C;\Gamma \vDash^{M} Q \rightsquigarrow d}$ *(Constraint Entailment)*

osEntail-inst

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q').m \mapsto \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i, \overline{b}_k, \overline{\delta}_y : \overline{Q}_y : e, P_2 \\ Q = [\overline{\tau}_j/\overline{a}_j]Q' \\ \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma \\ \overline{\Gamma_C;\Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \overline{P;\Gamma_C;\Gamma \vDash [\overline{\tau}_j/\overline{a}_j]Q_i \rightsquigarrow d_i}^{i} \end{array}}{P;\Gamma_C;\Gamma \vDash Q \rightsquigarrow D\,\overline{\sigma}_j\,\overline{d}_i}$$

osEntail-local

$$\frac{\begin{array}{c} (\delta : Q) \in \Gamma \\ \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma \end{array}}{P;\Gamma_C;\Gamma \vDash Q \rightsquigarrow \delta}$$

$\lambda_{\mathbf{TC}}$ Environment Well-Formedness

$\boxed{\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma}$ *($\lambda_{\mathbf{TC}}$ Environment Well-Formedness)*

osCtx-empty

$$\frac{}{\vdash_{ctx} \bullet;\bullet;\bullet \rightsquigarrow \bullet;\bullet;\bullet}$$

osCtx-clsEnv

$$\frac{\begin{array}{c} \Gamma_C;\bullet,a \vdash_{ty} \forall \overline{a}_j.\overline{TC_i\,a}^{i} \Rightarrow \tau \rightsquigarrow \sigma \\ \overline{a}_j, a = \mathbf{fv}(\tau) \\ \overline{\Gamma_C;\bullet,a \vdash_Q TC_i\,a \rightsquigarrow Q_i}^{i} \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\,b \notin \mathbf{dom}(\Gamma_C) \\ \vdash_{ctx} \bullet;\Gamma_C;\bullet \rightsquigarrow \bullet;\Gamma_C;\bullet \end{array}}{\vdash_{ctx} \bullet;\Gamma_C, m : \overline{TC_i\ a} \Rightarrow TC\,a : \forall \overline{a}_j.\overline{TC_i\,a}^{i} \Rightarrow \tau;\bullet \rightsquigarrow \bullet;\Gamma_C, m : TC\,a : \sigma;\bullet}$$

osCtx-tyEnvTm

$$\frac{\begin{array}{c} \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \bullet;\Gamma_C;\Gamma \end{array}}{\vdash_{ctx} \bullet;\Gamma_C;\Gamma, x : \sigma \rightsquigarrow \bullet;\Gamma_C;\Gamma, x : \sigma}$$

osCtx-tyEnvTy

$$\frac{\begin{array}{c} a \notin \Gamma \\ \vdash_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \bullet;\Gamma_C;\Gamma \end{array}}{\vdash_{ctx} \bullet;\Gamma_C;\Gamma, a \rightsquigarrow \bullet;\Gamma_C;\Gamma, a}$$

$$\textsc{osCtx-tyEnvD}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_Q TC\,\tau \rightsquigarrow Q\\ \delta \notin \mathbf{dom}(\Gamma)\\ \vdash_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \bullet;\Gamma_C;\Gamma\end{array}}{\vdash_{ctx} \bullet;\Gamma_C;\Gamma,\delta : TC\,\tau \rightsquigarrow \bullet;\Gamma_C;\Gamma,\delta : Q}$$

$$\textsc{osCtx-pgmInst}\quad \frac{\begin{array}{c}\mathbf{unambig}(\forall \bar{b}_j.\overline{Q}_i \Rightarrow TC\,\tau)\\ \Gamma_C;\bullet \vdash_C \forall \bar{b}_j.\overline{Q}_i \Rightarrow TC\,\tau \rightsquigarrow \forall \bar{b}_j.\overline{Q}_i \Rightarrow TC\,\sigma\\ (m : \overline{Q}'_m \Rightarrow TC\,a : \forall \bar{a}_k.\overline{Q}_y \Rightarrow \tau') \in \Gamma_C\\ P;\Gamma_C;\bullet,\bar{b}_j,\bar{\delta}_i : \overline{Q}_i,\bar{a}_k,\bar{\delta}_y : [\tau/a]\overline{Q}_y \vdash_{tm} e \Leftarrow [\tau/a]\tau' \rightsquigarrow e\\ \Gamma_C;\bullet,a \vdash_{ty} \forall \bar{a}_k.\overline{Q}_y \Rightarrow \tau' \rightsquigarrow \forall \bar{a}_k.\overline{Q}_y \Rightarrow \sigma'\\ D \notin \mathbf{dom}(P)\\ (D' : \forall \bar{b}'_k.\overline{Q}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P\\ \mathbf{where}[\bar{\tau}_j/\bar{b}_j]\tau = [\bar{\tau}'_k/\bar{b}'_k]\tau''\\ \vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma\\ \Sigma' = \Sigma,(D : \forall \bar{b}_j.\overline{Q}_i \Rightarrow TC\,\sigma).m \mapsto \Lambda \bar{b}_j.\lambda \bar{\delta}_i : \overline{Q}_i.\Lambda \bar{a}_k.\lambda \bar{\delta}_y : [\sigma/a]\overline{Q}_y.e\end{array}}{\vdash_{ctx} P,(D : \forall \bar{b}_j.\overline{Q}_i \Rightarrow TC\,\tau).m \mapsto \bullet,\bar{b}_j,\bar{\delta}_i : \overline{Q}_i,\bar{a}_k,\bar{\delta}_y : [\tau/a]\overline{Q}_y : e;\Gamma_C;\Gamma \rightsquigarrow \Sigma';\Gamma_C;\Gamma}$$

$\lambda_{\mathbf{TC}}$ Context Typing and Elaboration

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Inference - Inference)*

$$\textsc{osM-inf-infT-empty}\quad \frac{}{[\bullet] : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]}$$

$$\textsc{osM-inf-infT-appL}\quad \frac{\begin{array}{c}M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M\\ P;\Gamma_C;\Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2\end{array}}{M\,e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M\,e_2}$$

$$\textsc{osM-inf-infT-appR}\quad \frac{\begin{array}{c}M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \rightsquigarrow M\\ P;\Gamma_C;\Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1\end{array}}{e_1\,M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\,M}$$

osM-inf-infT-letL

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \Leftarrow \tau_1) \leadsto M \\ \overline{\delta}_i \ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \leadsto e_2 \\ \Gamma_C; \Gamma' \vdash_{ty} \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 \leadsto \forall \overline{a}_j . \overline{\sigma}_i \rightarrow \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{\sigma}_i \rightarrow \sigma_1 = \Lambda \overline{a}_j . \lambda \overline{\delta_i : \sigma_i}^{i} . M \ \mathbf{in}\ \ e_2 \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 = M \ \mathbf{in}\ \ e_2 : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \leadsto M'}$$

osM-inf-infT-letR

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \leadsto M \\ \overline{\delta}_i \ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \leadsto e_1 \\ \Gamma_C; \Gamma' \vdash_{ty} \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 \leadsto \forall \overline{a}_j . \overline{\sigma}_i \rightarrow \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{\sigma}_i \rightarrow \sigma_1 = \Lambda \overline{a}_j . \lambda \overline{\delta_i : \sigma_i}^{i} . e_1 \ \mathbf{in}\ \ M \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j . \overline{Q}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in}\ \ M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \leadsto M'}$$

osM-inf-infT-ann

$$\frac{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \leadsto M}{M :: \tau' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \leadsto M}$$

$\boxed{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \leadsto M}$ *($\lambda_{\mathbf{TC}}$ Context Inference - Checking)*

osM-inf-checkT-abs

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Rightarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma', x : \tau \Leftarrow \tau_2) \leadsto M \\ \Gamma_C; \Gamma' \vdash_{ty} \tau \leadsto \sigma \end{array}}{\lambda x . M : (P; \Gamma_C; \Gamma \Rightarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau \rightarrow \tau_2) \leadsto \lambda x : \sigma . M}$$

osM-inf-checkT-inf

$$\frac{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \leadsto M}{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \leadsto M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Checking - Inference)*

osM-check-infT-appL

$$\dfrac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{M\ e_2 : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M\ e_2}$$

osM-check-infT-appR

$$\dfrac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \end{array}}{e_1\ M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\ M}$$

osM-check-infT-letL

$$\dfrac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \Leftarrow \tau_1) \rightsquigarrow M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 = \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma_i}^{\,i}.M\ \mathbf{in}\ \ e_2 \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = M\ \mathbf{in}\ \ e_2 : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

osM-check-infT-letR

$$\dfrac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 = \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma_i}^{\,i}.e_1\ \mathbf{in}\ \ M \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = e_1\ \mathbf{in}\ \ M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

osM-check-infT-ann

$$\dfrac{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Checking - Checking)*

osM-check-checkT-empty

$$\dfrac{}{[\bullet] : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Leftarrow \tau) \rightsquigarrow [\bullet]}$$

osM-check-checkT-abs

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma', x : \tau \Leftarrow \tau_2) \leadsto M \\ \Gamma_C;\Gamma' \vdash_{ty} \tau \leadsto \sigma \end{array}}{\lambda x.M : (P;\Gamma_C;\Gamma \Leftarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau \to \tau_2) \leadsto \lambda x : \sigma.M}$$

osM-check-checkT-inf

$$\frac{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \leadsto M}{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \leadsto M}$$

λ_{TC} Context Typing and Elaboration through F_{D}

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \leadsto M}$ *($\lambda_{\mathbf{TC}}$ Context Inference - Inference)*

osM-inf-inf-empty

$$\frac{}{[\bullet] : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Rightarrow \tau) \leadsto [\bullet]}$$

osM-inf-inf-appL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \leadsto M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \leadsto e_2 \end{array}}{M\, e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto M\, e_2}$$

osM-inf-inf-appR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \leadsto M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \leadsto e_1 \end{array}}{e_1\, M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto e_1\, M}$$

osM-inf-inf-letL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \Leftarrow \tau_1) \leadsto M \\ \overline{\delta}_i \ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \leadsto e_2 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \leadsto \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{Q}_i.M \ \mathbf{in}\ \ e_2 \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = M \ \mathbf{in}\ \ e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto M'}$$

osM-inf-inf-letR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i \ \textbf{fresh} \\ x \notin \textbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma',\overline{a}_j,\overline{\delta}_i : \overline{Q}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 \\ M' = \textbf{let} \ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{Q}_i.e_1 \ \textbf{in} \ \ M \end{array}}{\textbf{let} \ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = e_1 \ \textbf{in} \ \ M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

osM-inf-inf-ann

$$\frac{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Inference - Checking)*

osM-inf-check-abs

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C;\Gamma' \vdash_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x.M : (P;\Gamma_C;\Gamma \Rightarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau \to \tau_2) \rightsquigarrow \lambda x : \sigma.M}$$

osM-inf-check-inf

$$\frac{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Checking - Inference)*

osM-check-inf-appL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{M\ e_2 : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M\ e_2}$$

osM-check-inf-appR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \end{array}}{e_1\ M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\ M}$$

osM-check-inf-letL

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \Leftarrow \tau_1) \rightsquigarrow M \\ \overline{\delta}_i \ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 \\ M' = \mathbf{let} \ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{Q}_i.M \ \mathbf{in} \ \ e_2 \end{array}}{\mathbf{let} \ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = M \ \mathbf{in} \ \ e_2 : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

osM-check-inf-letR

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i \ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C; \Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 \\ M' = \mathbf{let} \ \ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{Q}_i.e_1 \ \mathbf{in} \ \ M \end{array}}{\mathbf{let} \ \ x : \sigma_1 = e_1 \ \mathbf{in} \ \ M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

osM-check-inf-ann

$$\frac{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathbf{TC}}$ Context Checking - Checking)*

osM-check-check-empty

$$\frac{}{[\bullet] : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Leftarrow \tau) \rightsquigarrow [\bullet]}$$

osM-check-check-abs

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x.M : (P; \Gamma_C; \Gamma \Leftarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau \to \tau_2) \rightsquigarrow \lambda x : \sigma.M}$$

osM-check-check-inf

$$\frac{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

A.4 $\lambda^{\Rightarrow}_{\mathsf{TC}}$ Declarative Type System Additional Judgments

A.4.1 Well-formedness of Types & Constraints

Well-formedness of types takes the form $\Gamma \vdash_{\mathsf{ty}} \sigma$ and is given by the following rules:

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a}\ \textsc{TyVar} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2}\ \textsc{TyArr}$$

$$\frac{\Gamma \vdash_{\mathsf{ct}} C \qquad \Gamma \vdash_{\mathsf{ty}} \rho}{\Gamma \vdash_{\mathsf{ty}} C \Rightarrow \rho}\ \textsc{TyQual} \qquad \frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\mathsf{ty}} \sigma}{\Gamma \vdash_{\mathsf{ty}} \forall a.\sigma}\ \textsc{TyAll}$$

It is entirely straightforward and ensures that type terms are well-scoped. Rule TyQual requires checking the well-formedness of our new form of constraints C, via relation $\Gamma \vdash_{\mathsf{ct}} C$, given by the following rules:

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau}{\Gamma \vdash_{\mathsf{ct}} TC\ \tau}\ (\mathrm{C}Q) \qquad \frac{\Gamma \vdash_{\mathsf{ct}} C_1 \qquad \Gamma \vdash_{\mathsf{ct}} C_2}{\Gamma \vdash_{\mathsf{ct}} C_1 \Rightarrow C_2}\ (\mathrm{C}\Rightarrow) \qquad \frac{\begin{array}{c} a \notin \Gamma \\ \Gamma, a \vdash_{\mathsf{ct}} C \end{array}}{\Gamma \vdash_{\mathsf{ct}} \forall a.C}\ (\mathrm{C}\forall)$$

Finally, an axiom set A is well-formed if all constraints it contains are well-formed:

$$\frac{}{\Gamma \vdash_{\mathsf{ax}} \bullet}\ \textsc{AxNil} \qquad \frac{\Gamma \vdash_{\mathsf{ax}} A \qquad \Gamma \vdash_{\mathsf{ct}} C}{\Gamma \vdash_{\mathsf{ax}} A, C}\ \textsc{AxCons}$$

A.4.2 Program Typing

The judgment for program typing takes the form $P; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma$ and is given by the following rules:

$$\frac{\Gamma \vdash_{\mathsf{cls}} cls : A_S; \Gamma_c \qquad P ,_{\mathrm{S}} A_S; \Gamma, \Gamma_c \vdash_{\mathsf{pgm}} pgm : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} (cls; pgm) : \sigma}\ \textsc{PgmCls}$$

$$\frac{P; \Gamma \vdash_{\mathsf{inst}} inst : A_I \qquad P ,_{\mathrm{I}} A_I; \Gamma \vdash_{\mathsf{pgm}} pgm : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} (inst; pgm) : \sigma}\ \textsc{PgmInst}$$

$$\frac{P; \Gamma \vdash_{\mathsf{tm}} e : \sigma}{P; \Gamma \vdash_{\mathsf{pgm}} e : \sigma}\ \textsc{PgmExpr}$$

For brevity, if $P = \bullet$ and $\Gamma = \bullet$ we denote program typing as $\vdash_{\mathsf{pgm}} pgm : \sigma$.

A.4.3 Elaboration of Programs

Elaboration of programs is given by judgment $\mathcal{P}; \Gamma \vdash_{\mathtt{pgm}} pgm : \sigma \rightsquigarrow \mathit{fpgm}$:

$\boxed{\mathcal{P}; \Gamma \vdash_{\mathtt{pgm}} pgm : \sigma \rightsquigarrow \mathit{fpgm}}$ (Program Elaboration)

$$\frac{\Gamma \vdash_{\mathtt{cls}} cls : \mathcal{A}_S; \Gamma_c \rightsquigarrow \mathit{fdata}; \overline{\mathit{fval}} \qquad \mathcal{P},_{\mathrm{S}} \mathcal{A}_S; \Gamma, \Gamma_c \vdash_{\mathtt{pgm}} pgm : \sigma \rightsquigarrow \mathit{fpgm}}{\mathcal{P}; \Gamma \vdash_{\mathtt{pgm}} (cls; pgm) : \sigma \rightsquigarrow \mathit{fdata}; \overline{\mathit{fval}}; \mathit{fpgm}} \; \textsc{PCls}$$

$$\frac{\mathcal{P}; \Gamma \vdash_{\mathtt{inst}} inst : \mathcal{A}_I \rightsquigarrow \mathit{fval} \qquad \mathcal{P},_{\mathrm{I}} \mathcal{A}_I; \Gamma \vdash_{\mathtt{pgm}} pgm : \sigma \rightsquigarrow \mathit{fpgm}}{\mathcal{P}; \Gamma \vdash_{\mathtt{pgm}} (inst; pgm) : \sigma \rightsquigarrow \mathit{fval}; \mathit{fpgm}} \; \textsc{PIns}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{tm}} e : \tau \rightsquigarrow t \mid \mathcal{A}; E \qquad \theta = \mathit{unify}(\bullet; E) \qquad \overline{a} = \mathit{fv}(\theta(\mathcal{A})) \cup \mathit{fv}(\theta(\tau)) \\ \overline{a}; \langle \bullet, \mathcal{A}_I, \mathcal{A}_L \rangle \models \theta(\mathcal{A}) \rightsquigarrow \overline{d : C}; \eta \qquad \vdash_{\mathtt{ct}} C_i \rightsquigarrow \upsilon_i\end{array}}{\langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{A}_L \rangle; \Gamma \vdash_{\mathtt{pgm}} e : \forall \overline{a}. \overline{C} \Rightarrow \theta(\tau) \rightsquigarrow \Lambda \overline{a}. \lambda(\overline{d : \upsilon}). \eta(\theta(t))} \; \textsc{PExp}$$

Rules PCls and PIns handle class and instance declarations, respectively, and they are entirely standard. Rule PExp performs standard type-inference, simplification [43] and generalization for a top-level expression e. For simplicity, we do not utilize *interaction rules* (e.g. we do not simplify the constraints $\{\mathit{Eq}\ a, \mathit{Ord}\ a\}$ to $\{\mathit{Ord}\ a\}$), but it is straightforward to do so. Finally, observe that superclass axioms $\mathcal{A}_S$ are not used for the simplification of wanted constraints. This is standard practice for Haskell but our distinction between the axioms within the program theory allows us to express this explicitly.

A.5 $\lambda^{\Rrightarrow}_{\mathbf{TC}}$ Additional Definitions

A.5.1 Syntax

$$
\begin{array}{lcll}
pgm & ::= & e \mid cls; pgm \mid inst; pgm & \textit{spgm} \\
cls & ::= & \mathbf{class}\ \overline{C} \Rightarrow TC\, a\, \mathbf{where}\ \{m : \sigma\} & \textit{class decl.} \\
inst & ::= & \mathbf{instance}\ \overline{C} \Rightarrow TC\, \tau\, \mathbf{where}\ \{m = e\} & \textit{instance decl.} \\
\\
e & ::= & True \mid False \mid x \mid m \mid \lambda x.e \mid e_1\, e_2 \mid \mathbf{let}\ \ x : \sigma = e_1\, \mathbf{in}\ \ e_2 \mid e :: \tau & \textit{term} \\
\\
\tau & ::= & Bool \mid a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid C \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme} \\
\\
Q & ::= & TC\, \tau & \textit{class constraint} \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a.C & \textit{constraint} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : C & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \overline{C} \Rightarrow TC\, a : \sigma & \textit{class environment} \\
P & ::= & \bullet \mid P, (D : C).m \mapsto \Gamma : e & \textit{program context} \\
M & ::= & [\bullet] \mid \lambda x.M \mid M\, e \mid e\, M \mid M :: \tau & \textit{evaluation context} \\
 & & \mid \mathbf{let}\ \ x : \sigma = M\, \mathbf{in}\ \ e \mid \mathbf{let}\ \ x : \sigma = e\, \mathbf{in}\ \ M &
\end{array}
$$

A.5.2 $\lambda^{\Rrightarrow}_{\mathbf{TC}}$ Judgments and Elaboration

$\lambda^{\Rrightarrow}_{\mathbf{TC}}$ Type & Constraint Well-Formedness

$\boxed{\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma}$ *($\lambda^{\Rrightarrow}_{\mathbf{TC}}$ Class Constraint Well-Formedness)*

sQT-TC

$$
\frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma' \\ \Gamma_C = \Gamma_{C1}, m : \overline{C}_i \Rightarrow TC\, a : \sigma, \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{\Gamma_C; \Gamma \vdash_Q TC\, \tau \rightsquigarrow [\sigma'/a]\{m : \sigma\}}
$$

$\boxed{\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma}$ $(\lambda^{\Rightarrow}_{\mathbf{TC}}$ *Constraint Well-Formedness)*

$$\textsc{sCT-forall}\quad \frac{\Gamma_C ; \Gamma , a \vdash_C C \rightsquigarrow \sigma \qquad a \notin \Gamma}{\Gamma_C ; \Gamma \vdash_C \forall a . C \rightsquigarrow \forall a . \sigma}$$

$$\textsc{sCT-arrow}\quad \frac{\Gamma_C ; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1 \qquad \Gamma_C ; \Gamma \vdash_C C_2 \rightsquigarrow \sigma_2}{\Gamma_C ; \Gamma \vdash_C C_1 \Rightarrow C_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{sCT-classConstr}\quad \frac{\Gamma_C ; \Gamma \vdash_Q Q \rightsquigarrow \sigma}{\Gamma_C ; \Gamma \vdash_C Q \rightsquigarrow \sigma}$$

$\boxed{\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}$ $(\lambda^{\Rightarrow}_{\mathbf{TC}}$ *Type Well-Formedness)*

$$\textsc{sTyT-bool}\quad \frac{}{\Gamma_C ; \Gamma \vdash_{ty} \mathit{Bool} \rightsquigarrow \mathit{Bool}}$$

$$\textsc{sTyT-var}\quad \frac{a \in \Gamma}{\Gamma_C ; \Gamma \vdash_{ty} a \rightsquigarrow a}$$

$$\textsc{sTyT-arrow}\quad \frac{\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma_1 \qquad \Gamma_C ; \Gamma \vdash_{ty} \tau_2 \rightsquigarrow \sigma_2}{\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \rightarrow \tau_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{sTyT-qual}\quad \frac{\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_1 \qquad \Gamma_C ; \Gamma \vdash_{ty} \rho \rightsquigarrow \sigma_2}{\Gamma_C ; \Gamma \vdash_{ty} C \Rightarrow \rho \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{sTyT-scheme}\quad \frac{a \notin \Gamma \qquad \Gamma_C ; \Gamma , a \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Gamma_C ; \Gamma \vdash_{ty} \forall a . \sigma \rightsquigarrow \forall a . \sigma}$$

$\lambda^{\Rightarrow}_{\mathbf{TC}}$ Term Typing

$\boxed{P ; \Gamma_C ; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}$ $(\lambda^{\Rightarrow}_{\mathbf{TC}}$ *Term Inference)*

$$\textsc{sTm-infT-true}\quad \frac{\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma}{P ; \Gamma_C ; \Gamma \vdash_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}}$$

$$\textsc{sTm-infT-false}\quad \frac{\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma}{P ; \Gamma_C ; \Gamma \vdash_{tm} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}}$$

sTm-infT-let

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C ; \overline{C}_i) = \overline{C}_k \\ \overline{\Gamma_C ; \Gamma \vdash_C C_k \rightsquigarrow \sigma_k}^{k} \\ \Gamma_C ; \Gamma \vdash_{ty} \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j . \overline{\sigma}_k \rightarrow \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \\ P ; \Gamma_C ; \Gamma , \overline{a}_j , \overline{\delta}_k : \overline{C}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P ; \Gamma_C ; \Gamma , x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ e = \mathbf{let} \ \ x : \forall \overline{a}_j . \overline{\sigma}_k \rightarrow \sigma = \Lambda \overline{a}_j . \lambda \, \overline{\delta_k : \sigma_k}^{k} . e_1 \ \mathbf{in} \ \ e_2 \end{array}}{P ; \Gamma_C ; \Gamma \vdash_{tm} \mathbf{let} \ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in} \ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e}$$

sTm-infT-ArrE

$$\frac{\begin{array}{c} P ; \Gamma_C ; \Gamma \vdash_{tm} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \\ P ; \Gamma_C ; \Gamma \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{P ; \Gamma_C ; \Gamma \vdash_{tm} e_1 \, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1 \, e_2}$$

sTm-infT-Ann

$$\frac{P ; \Gamma_C ; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}{P ; \Gamma_C ; \Gamma \vdash_{tm} e :: \tau \Rightarrow \tau \rightsquigarrow e}$$

$\boxed{P ; \Gamma_C ; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Term Checking)*

sTm-checkT-var

$$\frac{\begin{array}{c} (x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \\ \overline{P ; \Gamma_C ; \Gamma \vDash [[\overline{\tau}_j / \overline{a}_j] C_i] \rightsquigarrow e_i}^{i} \\ \overline{\Gamma_C ; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma \end{array}}{P ; \Gamma_C ; \Gamma \vdash_{tm} x \Leftarrow [\overline{\tau}_j / \overline{a}_j] \tau \rightsquigarrow x \, \overline{\sigma}_j \, \overline{e}_i}$$

sTm-checkT-meth

$$\frac{\begin{array}{c} (m : \overline{C}'_k \Rightarrow TC \, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j , a . \overline{C}_i \Rightarrow \tau') \\ P ; \Gamma_C ; \Gamma \vDash [TC \, \tau] \rightsquigarrow e \\ \Gamma_C ; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \\ \overline{P ; \Gamma_C ; \Gamma \vDash [[\overline{\tau}_j / \overline{a}_j][\tau / a] C_i] \rightsquigarrow e_i}^{i} \\ \overline{\Gamma_C ; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \\ \vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma \end{array}}{P ; \Gamma_C ; \Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j / \overline{a}_j][\tau / a] \tau' \rightsquigarrow e . m \, \overline{\sigma}_j \, \overline{e}_i}$$

sTm-checkT-ArrI

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ P;\Gamma_C;\Gamma, x : \tau_1 \vdash_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \\ \Gamma_C;\Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma \end{array}}{P;\Gamma_C;\Gamma \vdash_{tm} \lambda x.e \Leftarrow \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \sigma.e}$$

sTm-checkT-Inf

$$\frac{P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}{P;\Gamma_C;\Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

$\boxed{\Gamma_C \vdash_{cls} cls : \Gamma_C{}'}$ *(Class Decl Typing)*

sClsT-cls

$$\frac{\begin{array}{c} m \notin \mathbf{dom}(\Gamma_C) \\ \mathbf{closure}(\Gamma_C;\overline{C}_m) = \overline{C}_n \\ \Gamma_C;\bullet, a \vdash_{ty} \forall \overline{a}_j.\overline{C}_n \Rightarrow \tau \rightsquigarrow \sigma \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{C}_n \Rightarrow \tau) \\ \overline{\Gamma_C;\bullet, a \vdash_C C_i \rightsquigarrow \sigma_i}^{\,i<q} \\ \nexists TC' : (m : \overline{C}'_w \Rightarrow TC'\ b : \sigma') \in \Gamma_C \\ \nexists m' : (m' : \overline{C}'_w \Rightarrow TC\ a : \sigma') \in \Gamma_C \\ \Gamma_C{}' = m : \overline{C}_q \Rightarrow TC\, a : \forall \overline{a}_j.\overline{C}_n \Rightarrow \tau \end{array}}{\Gamma_C \vdash_{cls} \mathbf{class}\ \ \overline{C}_q \Rightarrow TC\, a\, \mathbf{where}\ \{m : \forall \overline{a}_j.\overline{C}_m \Rightarrow \tau\} : \Gamma_C{}'}$$

$\boxed{P;\Gamma_C \vdash_{inst} inst : P'}$ *(Instance Decl Typing)*

sInstT-inst

$$\frac{\begin{array}{c} (m : \overline{C}'_i \Rightarrow TC\, a : \forall \overline{a}_j.\overline{C}'_y \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C;\bullet, \overline{b}_k \vdash_{ty} \tau \rightsquigarrow \sigma \\ \mathbf{closure}(\Gamma_C;\overline{C}_p) = \overline{C}_q \\ \mathbf{unambig}(\forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau) \\ \overline{\Gamma_C;\bullet, \overline{b}_k \vdash_C C_q \rightsquigarrow \sigma_q}^{\,q} \\ \overline{P;\Gamma_C;\bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q \vDash [[\tau/a]C'_i] \rightsquigarrow e_i}^{\,i} \\ P;\Gamma_C;\bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}'_y : [\tau/a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ D\ \mathbf{fresh} \\ \overline{\delta}_q\ \mathbf{fresh} \qquad \overline{\delta}'_y\ \mathbf{fresh} \\ (D' : \forall \overline{b}'_w.\overline{C}'_n \Rightarrow TC\,\tau_2).m' \mapsto \Gamma' : e' \notin P\ \ \mathit{where}\ [\overline{\tau}'_w/\overline{b}'_w]\tau_2 = [\overline{\tau}'_k/\overline{b}_k]\tau \\ P' = (D : \forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}'_y : [\tau/a]\overline{C}'_y : e \end{array}}{P;\Gamma_C \vdash_{inst} \mathbf{instance}\ \ \overline{C}_p \Rightarrow TC\,\tau\, \mathbf{where}\ \{m = e\} : P'}$$

$\boxed{P;\Gamma_C \vdash_{pgm} pgm : \tau; P';\Gamma_C{}' \leadsto e}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Program Typing)*

sPgmT-cls

$$\frac{\Gamma_C \vdash_{cls} cls : \Gamma_C{}' \qquad P;\Gamma_C,\Gamma_C{}' \vdash_{pgm} pgm : \tau; P';\Gamma_C{}'' \leadsto e}{P;\Gamma_C \vdash_{pgm} cls; pgm : \tau; P';\Gamma_C{}',\Gamma_C{}'' \leadsto e}$$

sPgmT-inst

$$\frac{P;\Gamma_C \vdash_{inst} inst : P' \qquad P,P';\Gamma_C \vdash_{pgm} pgm : \tau; P'';\Gamma_C{}' \leadsto e}{P;\Gamma_C \vdash_{pgm} inst; pgm : \tau; P',P'';\Gamma_C{}' \leadsto e}$$

sPgmT-expr

$$\frac{P;\Gamma_C;\bullet \vdash_{tm} e \Rightarrow \tau \leadsto e}{P;\Gamma_C \vdash_{pgm} e : \tau; \bullet; \bullet \leadsto e}$$

$\boxed{\mathbf{closure}(\Gamma_C;\overline{C}_i) = \overline{C}_j}$ *(Closure over Superclass Relation)*

sClosure-empty

$$\frac{}{\mathbf{closure}(\Gamma_C;\bullet) = \bullet}$$

sClosure-TC

$$\frac{TC\ a = head(C) \qquad (m : \overline{C}_m \Rightarrow TC\ a : \sigma) \in \Gamma_C \qquad \mathbf{closure}(\Gamma_C;\overline{C}_i,\overline{C}_m) = \overline{C}_j}{\mathbf{closure}(\Gamma_C;\overline{C}_i,C) = \overline{C}_j,C}$$

$\boxed{\mathbf{unambig}(\sigma)}$ *(Unambiguity for Type Schemes)*

sUnambig-scheme

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j.\overline{C}_i \Rightarrow \tau)}$$

$\boxed{\mathbf{unambig}(C)}$ *(Unambiguity for Constraints)*

sUnambig-constraint

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j.\overline{C}_i \Rightarrow TC\ \tau)}$$

Constraint Proving

$$\boxed{P; \Gamma_C; \Gamma \vDash [C] \rightsquigarrow e}$$

(Constraint Entailment)

sEntailT-arrow

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash [C_2] \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1 \end{array}}{P; \Gamma_C; \Gamma \vDash [C_1 \Rightarrow C_2] \rightsquigarrow \lambda \delta_1 : \sigma_1 . e}$$

sEntailT-forall

$$\frac{P; \Gamma_C; \Gamma, a \vDash [C] \rightsquigarrow e}{P; \Gamma_C; \Gamma \vDash [\forall a . C] \rightsquigarrow \Lambda a . e}$$

sEntailT-inst

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q').m \mapsto \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y : e, P_2 \\ P_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_0 \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C'_i \rightsquigarrow \sigma'_i}^{i} \\ \overline{\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_C C_y \rightsquigarrow \sigma''_y}^{y} \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ e'_0 = \Lambda \overline{a}_j . \lambda \, \overline{\delta'_i : \sigma'_i}^{i} . \{ m = \Lambda \overline{b}_k . \lambda \, \overline{\delta_y : \sigma''_y}^{y} . e_0 \} \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash e'_0 : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q'] \vDash Q \rightsquigarrow \bullet \vdash e_1 \end{array}}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e_1}$$

sEntailT-local

$$\frac{\begin{array}{c} (\delta : C) \in \Gamma \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta : C] \vDash Q \rightsquigarrow \bullet \vdash e \end{array}}{P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e}$$

$$\boxed{P; \Gamma_C; \Gamma; [\overline{a}; \bullet \vdash e_0 : C] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_1}$$

(Constraint Matching)

sMatchT-arrow

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C}, \delta_1 : C_1 \vdash e_0 \, \delta_1 : C_2] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_2 \\ P; \Gamma_C; \Gamma \vDash [[\overline{\tau}/\overline{a}] C_1] \rightsquigarrow e_1 \end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C_1 \Rightarrow C_2] \vDash Q \rightsquigarrow \overline{\tau} \vdash [e_1/\delta_1] e_2}$$

sMatchT-forall

$$\frac{P; \Gamma_C; \Gamma; [\overline{a}, a; \overline{\delta} : \overline{C} \vdash e_0 \, a : C] \vDash Q \rightsquigarrow \overline{\tau}, \tau \vdash e_1}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : \forall a . C] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_1}$$

sMatchT-classconstr

$$\frac{\begin{array}{c} \tau_1 = [\overline{\tau}/\overline{a}] \tau_0 \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \tau_i \rightsquigarrow \sigma_i}^{i} \end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : TC \, \tau_0] \vDash TC \, \tau_1 \rightsquigarrow \overline{\tau} \vdash [\overline{\sigma}/\overline{a}] e_0}$$

$\lambda^{\Rightarrow}_{\mathbf{TC}}$ Environment Well-Formedness

$\boxed{\vdash_{ctx} P; \Gamma_C; \Gamma \leadsto \Gamma}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Environment Well-Formedness)*

sCtxT-empty

$$\frac{}{\vdash_{ctx} \bullet; \bullet; \bullet \leadsto \bullet}$$

sCtxT-clsEnv

$$\frac{\begin{array}{c} \Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \leadsto \sigma \\ \overline{a}_j, a = \mathbf{fv}(\tau) \\ \overline{\Gamma_C; \bullet, a \vdash_C C_i \leadsto \sigma_i}^{\,i} \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\, b \notin \mathbf{dom}(\Gamma_C) \\ \vdash_{ctx} \bullet; \Gamma_C; \bullet \leadsto \bullet \end{array}}{\vdash_{ctx} \bullet; \Gamma_C, m : \overline{C}_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau; \bullet \leadsto \bullet}$$

sCtxT-tyEnvTm

$$\frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_{ty} \sigma \leadsto \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma \leadsto \Gamma \end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma \leadsto \Gamma, x : \sigma}$$

sCtxT-tyEnvTy

$$\frac{\begin{array}{c} a \notin \Gamma \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma \leadsto \Gamma \end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, a \leadsto \Gamma, a}$$

sCtxT-tyEnvD

$$\frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_C C \leadsto \sigma \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma \leadsto \Gamma \end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, \delta : C \leadsto \Gamma, \delta : \sigma}$$

sCtxT-pgmInst

$$\frac{\begin{array}{c} \mathbf{unambig}(\forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau) \\ \Gamma_C; \bullet \vdash_C \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau \leadsto \forall \overline{b}_j . \overline{\sigma}_i \rightarrow [\sigma / a]\{m : \forall \overline{a}_k . \overline{\sigma}'_y \rightarrow \sigma'\} \\ (m : \overline{C}'_m \Rightarrow TC\, a : \forall \overline{a}_k . \overline{C}'_y \Rightarrow \tau') \in \Gamma_C \\ \Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_k . \overline{C}'_y \Rightarrow \tau' \leadsto \forall \overline{a}_k . \overline{\sigma}'_y \rightarrow \sigma' \\ \Gamma_C; \bullet, \overline{b}_j \vdash_{ty} \tau \leadsto \sigma \\ P; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau / a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau / a]\tau' \leadsto e \\ D \notin \mathbf{dom}(P) \\ (D' : \forall \overline{b}'_k . \overline{C}''_y \Rightarrow TC\, \tau'').m' \mapsto \Gamma' : e' \notin P \\ \mathbf{where} [\overline{\tau}_j / \overline{b}_j]\tau = [\overline{\tau}'_k / \overline{b}'_k]\tau'' \\ \vdash_{ctx} P; \Gamma_C; \Gamma \leadsto \Gamma \end{array}}{\vdash_{ctx} P, (D : \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau).m \mapsto \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau / a]\overline{C}'_y : e; \Gamma_C; \Gamma \leadsto \Gamma}$$

A.5.3 $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Judgments and Elaboration through $F_{\mathbf{D}}^{\Rightarrow}$

$\lambda_{\mathsf{TC}}^{\Rightarrow}$ Type & Constraint Well-Formedness

$\boxed{\Gamma_C;\Gamma \vdash_{Q}^{M} Q \rightsquigarrow Q}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Class Constraint Well-Formedness)*

sQ-TC

$$\frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \\ \Gamma_C = \Gamma_{C1}, m : \overline{C}_i \Rightarrow TC\, a : \sigma, \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty}^{M} \sigma \rightsquigarrow \sigma'\end{array}}{\Gamma_C;\Gamma \vdash_{Q}^{M} TC\,\tau \rightsquigarrow TC\,\sigma}$$

$\boxed{\Gamma_C;\Gamma \vdash_{C}^{M} C \rightsquigarrow C}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Constraint Well-Formedness)*

sC-forall

$$\frac{\begin{array}{c}\Gamma_C;\Gamma, a \vdash_{C}^{M} C \rightsquigarrow C \\ a \notin \Gamma\end{array}}{\Gamma_C;\Gamma \vdash_{C}^{M} \forall a.C \rightsquigarrow \forall a.C}$$

sC-arrow

$$\frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{C}^{M} C_1 \rightsquigarrow C_1 \\ \Gamma_C;\Gamma \vdash_{C}^{M} C_2 \rightsquigarrow C_2\end{array}}{\Gamma_C;\Gamma \vdash_{C}^{M} C_1 \Rightarrow C_2 \rightsquigarrow C_1 \Rightarrow C_2}$$

sC-classConstr

$$\frac{\Gamma_C;\Gamma \vdash_{Q}^{M} Q \rightsquigarrow Q}{\Gamma_C;\Gamma \vdash_{C}^{M} Q \rightsquigarrow Q}$$

$\boxed{\Gamma_C;\Gamma \vdash_{ty}^{M} \sigma \rightsquigarrow \sigma}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Type Well-Formedness)*

sTy-bool

$$\frac{}{\Gamma_C;\Gamma \vdash_{ty}^{M} Bool \rightsquigarrow Bool}$$

sTy-var

$$\frac{a \in \Gamma}{\Gamma_C;\Gamma \vdash_{ty}^{M} a \rightsquigarrow a}$$

sTy-arrow

$$\frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty}^{M} \tau_1 \rightsquigarrow \sigma_1 \\ \Gamma_C;\Gamma \vdash_{ty}^{M} \tau_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C;\Gamma \vdash_{ty}^{M} \tau_1 \rightarrow \tau_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

sTy-qual

$$\frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{C}^{M} C \rightsquigarrow C \\ \Gamma_C;\Gamma \vdash_{ty}^{M} \rho \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma \vdash_{ty}^{M} C \Rightarrow \rho \rightsquigarrow C \Rightarrow \sigma}$$

sTy-scheme

$$\frac{\begin{array}{c}a \notin \Gamma \\ \Gamma_C;\Gamma, a \vdash_{ty}^{M} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma \vdash_{ty}^{M} \forall a.\sigma \rightsquigarrow \forall a.\sigma}$$

$\lambda_{\mathbf{TC}}^{\Rightarrow}$ Term Typing

$\boxed{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Rightarrow \tau \rightsquigarrow e}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Term Inference)*

$$\textsc{sTm-inf-True}\quad \frac{\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}}$$

$$\textsc{sTm-inf-False}\quad \frac{\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}}$$

$$\textsc{sTm-inf-Let}\quad \frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C; \overline{C}_i) = \overline{C}_k \\ \Gamma_C; \Gamma \vdash_{ty}^{M} \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \\ P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash_{tm}^{M} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \vdash_{tm}^{M} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ e = \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1 \ \mathbf{in}\ \ e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} \mathbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e}$$

$$\textsc{sTm-inf-ArrE}\quad \frac{\begin{array}{c} P; \Gamma_C; \Gamma \vdash_{tm}^{M} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma \vdash_{tm}^{M} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2}$$

$$\textsc{sTm-inf-Ann}\quad \frac{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e}$$

$\boxed{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Term Checking)*

$$\textsc{sTm-check-Var}\quad \frac{\begin{array}{c} (x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma \\ \mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \\ \overline{P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j / \overline{a}_j] C_i] \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j / \overline{a}_j] \tau \rightsquigarrow x\, \overline{\sigma}_j\, \overline{d}_i}$$

sTm-check-meth

$$\frac{\begin{array}{c} (m : \overline{C}'_k \Rightarrow TC\,a : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{C}_i \Rightarrow \tau') \\ \overline{\begin{array}{c} P; \Gamma_C; \Gamma \vDash^{M} [TC\,\tau] \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \end{array}} \\ \overline{P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j/\overline{a}_j][\tau/a]C_i] \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash^{M}_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\,\overline{\sigma}_j\,\overline{d}_i}$$

sTm-check-ArrI

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ P; \Gamma_C; \Gamma, x : \tau_1 \vdash^{M}_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightsquigarrow \sigma \end{array}}{P; \Gamma_C; \Gamma \vdash^{M}_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma.e}$$

sTm-check-Inf

$$\frac{P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

$\boxed{\Gamma_C \vdash^{M}_{cls} cls : \Gamma_C{}'}$ *(Class Decl Typing)*

sCls-cls

$$\frac{\begin{array}{c} m \notin \mathbf{dom}(\Gamma_C) \\ \mathbf{closure}(\Gamma_C; \overline{C}_m) = \overline{C}_n \\ \Gamma_C; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j.\overline{C}_n \Rightarrow \tau \rightsquigarrow \sigma \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{C}_n \Rightarrow \tau) \\ \overline{\Gamma_C; \bullet, a \vdash^{M}_{C} C_i \rightsquigarrow C_i}^{\,i<q} \\ \nexists TC' : (m : \overline{C}'_w \Rightarrow TC'\ b : \sigma') \in \Gamma_C \\ \nexists m' : (m' : \overline{C}'_w \Rightarrow TC\ a : \sigma') \in \Gamma_C \\ \Gamma_C{}' = m : \overline{C}_q \Rightarrow TC\,a : \forall \overline{a}_j.\overline{C}_n \Rightarrow \tau \end{array}}{\Gamma_C \vdash^{M}_{cls} \mathbf{class}\ \overline{C}_q \Rightarrow TC\,a\,\mathbf{where}\ \{m : \forall \overline{a}_j.\overline{C}_m \Rightarrow \tau\} : \Gamma_C{}'}$$

$\boxed{P; \Gamma_C \vdash^{M}_{inst} inst : P'}$ *(Instance Decl Typing)*

sInst-inst

$$\frac{\begin{array}{c}(m : \overline{C}'_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_y \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C; \bullet, \overline{b}_k \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \\ \mathbf{closure}(\Gamma_C; \overline{C}_p) = \overline{C}_q \\ \mathbf{unambig}(\forall \overline{b}_k . \overline{C}_q \Rightarrow TC\, \tau) \\ \overline{\overline{\Gamma_C; \bullet, \overline{b}_k \vdash^{M}_{C} C_q \rightsquigarrow C_q}^{\,q}}^{\,i} \\ P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q \vDash^{M} [[\tau/a]C'_i] \rightsquigarrow d_i \\ P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y \vdash^{M}_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ D\ \mathbf{fresh} \\ \overline{\delta}_y\ \mathbf{fresh} \qquad \overline{\delta}_q\ \mathbf{fresh} \\ (D' : \forall \overline{b}'_m . \overline{C}'_n \Rightarrow TC\, \tau_2).m' \mapsto \Gamma' : e' \notin P\ \mathit{where}\ [\overline{\tau}'_m / \overline{b}'_m]\tau_2 = [\overline{\tau}'_k / \overline{b}_k]\tau \\ P' = (D : \forall \overline{b}_k . \overline{C}_q \Rightarrow TC\, \tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y : e\end{array}}{P; \Gamma_C \vdash^{M}_{inst} \mathbf{instance}\ \overline{C}_p \Rightarrow TC\, \tau\, \mathbf{where}\ \{m = e\} : P'}$$

$\boxed{P; \Gamma_C \vdash^{M}_{pgm} pgm : \tau; P'; {\Gamma_C}' \rightsquigarrow e}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Program Typing)*

sPgm-cls

$$\frac{\begin{array}{c}\Gamma_C \vdash^{M}_{cls} cls : {\Gamma_C}' \\ P; \Gamma_C, {\Gamma_C}' \vdash^{M}_{pgm} pgm : \tau; P'; {\Gamma_C}'' \rightsquigarrow e\end{array}}{P; \Gamma_C \vdash^{M}_{pgm} cls; pgm : \tau; P'; {\Gamma_C}', {\Gamma_C}'' \rightsquigarrow e}$$

sPgm-inst

$$\frac{\begin{array}{c}P; \Gamma_C \vdash^{M}_{inst} inst : P' \\ P, P'; \Gamma_C \vdash^{M}_{pgm} pgm : \tau; P''; {\Gamma_C}' \rightsquigarrow e\end{array}}{P; \Gamma_C \vdash^{M}_{pgm} inst; pgm : \tau; P', P''; {\Gamma_C}' \rightsquigarrow e}$$

sPgm-expr

$$\frac{P; \Gamma_C; \bullet \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C \vdash^{M}_{pgm} e : \tau; \bullet; \bullet \rightsquigarrow e}$$

Constraint Proving

$\boxed{P; \Gamma_C; \Gamma \vDash^{M} [C] \rightsquigarrow d}$ *(Constraint Entailment)*

sEntail-arrow

$$\frac{\begin{array}{c}P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash^{M} [C_2] \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash^{M}_{C} C_1 \rightsquigarrow C_1\end{array}}{P; \Gamma_C; \Gamma \vDash^{M} [C_1 \Rightarrow C_2] \rightsquigarrow \lambda \delta_1 : C_1 . d}$$

sEntail-forall

$$\frac{P; \Gamma_C; \Gamma, a \vDash^{M} [C] \rightsquigarrow d}{P; \Gamma_C; \Gamma \vDash^{M} [\forall a . C] \rightsquigarrow \Lambda a . d}$$

sEntail-inst

$$\frac{\begin{array}{c} P = P_1, (D : C).m \mapsto \Gamma' : e, P_2 \\ \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash D : C] \vDash^M Q \rightsquigarrow \bullet \vdash d \end{array}}{P; \Gamma_C; \Gamma \vDash^M [Q] \rightsquigarrow d}$$

sEntail-local

$$\frac{\begin{array}{c} (\delta : C) \in \Gamma \\ \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta : C] \vDash^M Q \rightsquigarrow \bullet \vdash d \end{array}}{P; \Gamma_C; \Gamma \vDash^M [Q] \rightsquigarrow d}$$

$\boxed{P; \Gamma_C; \Gamma; [\overline{a}; \bullet \vdash d_0 : C] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash d_1}$ *(Constraint Matching)*

sMatch-arrow

$$\frac{\begin{array}{c} P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C}, \delta_1 : C_1 \vdash d_0\, \delta_1 : C_2] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash d_2 \\ P; \Gamma_C; \Gamma \vDash^M [[\overline{\tau}/\overline{a}]C_1] \rightsquigarrow d_1 \end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : C_1 \Rightarrow C_2] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash [d_1/\delta_1]d_2}$$

sMatch-forall

$$\frac{P; \Gamma_C; \Gamma; [\overline{a}, a; \overline{\delta} : \overline{C} \vdash d_0\, a : C] \vDash^M Q \rightsquigarrow \overline{\tau}, \tau \vdash d_1}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : \forall a.C] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash d_1}$$

sMatch-classconstr

$$\frac{\dfrac{\tau_1 = [\overline{\tau}/\overline{a}]\tau_0}{\Gamma_C; \Gamma \vdash^M_{ty} \tau_i \rightsquigarrow \sigma_i}\,^{i}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : TC\, \tau_0] \vDash^M TC\, \tau_1 \rightsquigarrow \overline{\tau} \vdash [\overline{\sigma}/\overline{a}]d_0}$$

$\lambda_{\mathsf{TC}}^{\Rightarrow}$ Environment Well-Formedness

$\boxed{\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}$ *($\lambda_{\mathbf{TC}}^{\Rightarrow}$ Environment Well-Formedness)*

sCtx-empty

$$\frac{}{\vdash^M_{ctx} \bullet; \bullet; \bullet \rightsquigarrow \bullet; \bullet; \bullet}$$

sCtx-clsEnv

$$\frac{\begin{array}{c}\dfrac{\begin{array}{c}\Gamma_C ; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \rightsquigarrow \sigma \\ \overline{a}_j, a = \mathbf{fv}(\tau)\end{array}}{\Gamma_C ; \bullet, a \vdash^{M}_{C} C_i \rightsquigarrow Q_i}^{\,i} \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\, b \notin \mathbf{dom}(\Gamma_C) \\ \vdash^{M}_{ctx} \bullet ; \Gamma_C ; \bullet \rightsquigarrow \bullet ; \Gamma_C ; \bullet\end{array}}{\vdash^{M}_{ctx} \bullet ; \Gamma_C , m : \overline{C}_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau ; \bullet \rightsquigarrow \bullet ; \Gamma_C , m : TC\, a : \sigma ; \bullet}$$

sCtx-tyEnvTm

$$\frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash^{M}_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \bullet ; \Gamma_C ; \Gamma\end{array}}{\vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma , x : \sigma \rightsquigarrow \bullet ; \Gamma_C ; \Gamma , x : \sigma}$$

sCtx-tyEnvTy

$$\frac{\begin{array}{c}a \notin \Gamma \\ \vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \bullet ; \Gamma_C ; \Gamma\end{array}}{\vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma , a \rightsquigarrow \bullet ; \Gamma_C ; \Gamma , a}$$

sCtx-tyEnvD

$$\frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash^{M}_{C} C \rightsquigarrow C \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \bullet ; \Gamma_C ; \Gamma\end{array}}{\vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma , \delta : C \rightsquigarrow \bullet ; \Gamma_C ; \Gamma , \delta : C}$$

sCtx-pgmInst

$$\frac{\begin{array}{c}\mathbf{unambig}(\forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau) \\ \Gamma_C ; \bullet \vdash^{M}_{C} \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau \rightsquigarrow \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \sigma \\ (m : \overline{C}'_m \Rightarrow TC\, a : \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau') \in \Gamma_C \\ P ; \Gamma_C ; \bullet , \overline{b}_j , \overline{\delta}_i : \overline{C}_i , \overline{a}_k , \overline{\delta}_y : [\tau / a] \overline{C}_y \vdash^{M}_{tm} e \Leftarrow [\tau / a] \tau' \rightsquigarrow e \\ \Gamma_C ; \bullet , a \vdash^{M}_{ty} \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau' \rightsquigarrow \forall \overline{a}_k . \overline{C}_y \Rightarrow \sigma' \\ D \notin \mathbf{dom}(P) \\ (D' : \forall \overline{b}'_k . \overline{C}''_y \Rightarrow TC\, \tau'').m' \mapsto \Gamma' : e' \notin P \\ \mathbf{where} [\overline{\tau}_j / \overline{b}_j] \tau = [\overline{\tau}'_k / \overline{b}'_k] \tau'' \\ \vdash^{M}_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Sigma ; \Gamma_C ; \Gamma \\ \Sigma' = \Sigma , (D : \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \sigma).m \mapsto \Lambda \overline{b}_j . \lambda \overline{\delta}_i : \overline{C}_i . \Lambda \overline{a}_k . \lambda \overline{\delta}_y : [\sigma / a] \overline{C}_y . e\end{array}}{\vdash^{M}_{ctx} P , (D : \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau).m \mapsto \bullet , \overline{b}_j , \overline{\delta}_i : \overline{C}_i , \overline{a}_k , \overline{\delta}_y : [\tau / a] \overline{C}_y : e ; \Gamma_C ; \Gamma \rightsquigarrow \Sigma' ; \Gamma_C ; \Gamma}$$

$\lambda^{\Rightarrow}_{\mathsf{TC}}$ Context Typing and Elaboration

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \leadsto M}$ *($\lambda^{\Rightarrow}_{\mathsf{TC}}$ Context Inference - Inference)*

sM-inf-infT-empty

$$\frac{}{[\bullet] : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Rightarrow \tau) \leadsto [\bullet]}$$

sM-inf-infT-appL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \leadsto M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \leadsto e_2 \end{array}}{M\ e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto M\ e_2}$$

sM-inf-infT-appR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \leadsto M \\ P;\Gamma_C;\Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \leadsto e_1 \end{array}}{e_1\ M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto e_1\ M}$$

sM-inf-infT-letL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \leadsto M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \leadsto e_2 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \leadsto \forall\overline{a}_j.\overline{\sigma}_i \to \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall\overline{a}_j.\overline{\sigma}_i \to \sigma_1 = \Lambda\overline{a}_j.\lambda\overline{\delta_i : \sigma_i}^{\,i}.M\ \mathbf{in}\ \ e_2 \end{array}}{\mathbf{let}\ \ x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M\ \mathbf{in}\ \ e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto M'}$$

sM-inf-infT-letR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \leadsto M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \leadsto e_1 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \leadsto \forall\overline{a}_j.\overline{\sigma}_i \to \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall\overline{a}_j.\overline{\sigma}_i \to \sigma_1 = \Lambda\overline{a}_j.\lambda\overline{\delta_i : \sigma_i}^{\,i}.e_1\ \mathbf{in}\ \ M \end{array}}{\mathbf{let}\ \ x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1\ \mathbf{in}\ \ M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \leadsto M'}$$

sM-inf-infT-ann

$$\frac{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \leadsto M}{M :: \tau' : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \leadsto M}$$

$\boxed{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Context Inference - Checking)*

sM-inf-checkT-abs

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Rightarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x.M : (P; \Gamma_C; \Gamma \Rightarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau \to \tau_2) \rightsquigarrow \lambda x : \sigma.M}$$

sM-inf-checkT-inf

$$\frac{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Context Checking - Inference)*

sM-check-infT-appL

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M \\ P; \Gamma_C; \Gamma' \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{M\, e_2 : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M\, e_2}$$

sM-check-infT-appR

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1) \rightsquigarrow M \\ P; \Gamma_C; \Gamma' \vdash_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \end{array}}{e_1\, M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\, M}$$

sM-check-infT-letL

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \rightsquigarrow M \\ \overline{\delta}_i \textbf{ fresh} \\ x \notin \textbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 \\ M' = \textbf{let } \ x : \forall \overline{a}_j.\overline{\sigma}_i \to \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta_i : \sigma_i}^{\,i}.M \textbf{ in } \ e_2 \end{array}}{\textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M \textbf{ in } \ e_2 : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-check-infT-letR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C;\Gamma' \vdash_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_i \rightarrow \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_i \rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda\, \overline{\delta_i : \sigma_i}^{\,i}.e_1\, \mathbf{in}\ \ M \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1\, \mathbf{in}\ \ M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-check-infT-ann

$$\frac{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathsf{TC}}^{\Rightarrow}$ Context Checking - Checking)*

sM-check-checkT-empty

$$\frac{}{[\bullet] : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Leftarrow \tau) \rightsquigarrow [\bullet]}$$

sM-check-checkT-abs

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C;\Gamma' \vdash_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x.M : (P;\Gamma_C;\Gamma \Leftarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau \rightarrow \tau_2) \rightsquigarrow \lambda x : \sigma.M}$$

sM-check-checkT-inf

$$\frac{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

$\lambda_{\mathsf{TC}}^{\Rightarrow}$ Context Typing and Elaboration through $F_{\mathsf{D}}^{\Rightarrow}$

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda_{\mathsf{TC}}^{\Rightarrow}$ Context Inference - Inference)*

sM-inf-inf-empty

$$\frac{}{[\bullet] : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]}$$

sM-inf-inf-appL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \rightarrow \tau_2) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash^{M}_{\mathit{tm}} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{M\, e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M\, e_2}$$

sM-inf-inf-appR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash^{M}_{\mathit{tm}} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \end{array}}{e_1\, M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\, M}$$

sM-inf-inf-letL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \rightsquigarrow M \\ \overline{\delta}_i \ \textbf{fresh} \\ x \notin \textbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash^{M}_{\mathit{tm}} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ \Gamma_C;\Gamma' \vdash^{M}_{\mathit{ty}} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \\ M' = \textbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.M\ \textbf{in}\ \ e_2 \end{array}}{\textbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M\ \textbf{in}\ \ e_2 : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-inf-inf-letR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i \ \textbf{fresh} \\ x \notin \textbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash^{M}_{\mathit{tm}} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C;\Gamma' \vdash^{M}_{\mathit{ty}} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \\ M' = \textbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e_1\ \textbf{in}\ \ M \end{array}}{\textbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1\ \textbf{in}\ \ M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-inf-inf-ann

$$\frac{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Context Inference - Checking)*

sM-inf-check-abs

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Rightarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C;\Gamma' \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x.M : (P;\Gamma_C;\Gamma \Rightarrow \tau_1) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau \to \tau_2) \rightsquigarrow \lambda x : \sigma.M}$$

sM-inf-check-inf

$$\frac{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau') \rightsquigarrow M}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Context Checking - Inference)*

sM-check-inf-appL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash^{M}_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \end{array}}{M\, e_2 : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M\, e_2}$$

sM-check-inf-appR

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Leftarrow \tau_1) \rightsquigarrow M \\ P;\Gamma_C;\Gamma' \vdash^{M}_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \end{array}}{e_1\, M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow e_1\, M}$$

sM-check-inf-letL

$$\frac{\begin{array}{c} M : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \rightsquigarrow M \\ \overline{\delta}_i\ \mathbf{fresh} \\ x \notin \mathbf{dom}(\Gamma') \\ P;\Gamma_C;\Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash^{M}_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ \Gamma_C;\Gamma' \vdash^{M}_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \\ M' = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.M\ \mathbf{in}\ \ e_2 \end{array}}{\mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M\ \mathbf{in}\ \ e_2 : (P;\Gamma_C;\Gamma \Leftarrow \tau) \mapsto (P;\Gamma_C;\Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-check-inf-letR

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M \\ \overline{\delta}_i \ \textbf{fresh} \\ x \notin \textbf{dom}(\Gamma') \\ P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash^{M}_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ \Gamma_C; \Gamma' \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \\ M' = \textbf{let} \ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e_1 \ \textbf{in} \ \ M \end{array}}{\textbf{let} \ \ x : \sigma_1 = e_1 \ \textbf{in} \ \ M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M'}$$

sM-check-inf-ann

$$\frac{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}{M :: \tau' : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}$$

$\boxed{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$ *($\lambda^{\Rightarrow}_{\mathbf{TC}}$ Context Checking - Checking)*

sM-check-check-empty

$$\frac{}{[\bullet] : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Leftarrow \tau) \rightsquigarrow [\bullet]}$$

sM-check-check-abs

$$\frac{\begin{array}{c} M : (P; \Gamma_C; \Gamma \Leftarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma', x : \tau \Leftarrow \tau_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \end{array}}{\lambda x . M : (P; \Gamma_C; \Gamma \Leftarrow \tau_1) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau \to \tau_2) \rightsquigarrow \lambda x : \sigma . M}$$

sM-check-check-inf

$$\frac{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M}{M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M}$$

A.5.4 Unification Algorithm

The unification algorithm takes the form $\mathit{unify}(\overline{a}; E) = \theta_{\perp}$ and is given by the following equations:

$$\begin{array}{lcl} \mathit{unify}(\overline{a}; \bullet) & = & \bullet \\ \mathit{unify}(\overline{a}; E, b \sim b) & = & \mathit{unify}(\overline{a}; E) \\ \mathit{unify}(\overline{a}; E, b \sim \tau) & = & \mathit{unify}(\overline{a}; \theta(E)) \cdot \theta \\ \quad \text{where } b \notin \overline{a} \wedge b \notin \mathit{fv}(\tau) \wedge \theta = [\tau / b] & & \\ \mathit{unify}(\overline{a}; E, \tau \sim b) & = & \mathit{unify}(\overline{a}; \theta(E)) \cdot \theta \\ \quad \text{where } b \notin \overline{a} \wedge b \notin \mathit{fv}(\tau) \wedge \theta = [\tau / b] & & \\ \mathit{unify}(\overline{a}; E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4)) & = & \mathit{unify}(\overline{a}; E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4) \end{array}$$

Function *unify* is a straightforward extension of the standard first-order unification algorithm [17]. The only difference between the two lies in the additional argument: the *untouchable* variables $\overline{a}$. These variables are treated by the algorithm as skolem constants and therefore can not be substituted (they can be unified with themselves though).

A.6 F_D Additional Definitions

A.6.1 Syntax

$$\begin{array}{lcll}
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma.e \mid e_1\,e_2 \mid \lambda\delta : Q.e \mid e\,d & \textit{expression} \\
 & & \Lambda a.e \mid e\,\sigma \mid d.m \mid \mathbf{let}\ \ x : \sigma = e_1\,\mathbf{in}\ \ e_2 & \\
v & ::= & \mathit{True} \mid \mathit{False} \mid \lambda x : \sigma.e \mid \lambda\delta : Q.e \mid \Lambda a.e & \textit{value}
\end{array}$$

$$\begin{array}{lcll}
\sigma & ::= & \mathit{Bool} \mid a \mid \sigma_1 \to \sigma_2 \mid Q \Rightarrow \sigma \mid \forall a.\sigma & \textit{type} \\
Q & ::= & \mathit{TC}\,\sigma & \textit{class constraint} \\
C & ::= & \forall\overline{a}.\overline{Q} \Rightarrow Q & \textit{constraint}
\end{array}$$

$$\begin{array}{lcll}
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \mathit{TC}\,a : \sigma & \textit{class environment} \\
\Sigma & ::= & \bullet \mid \Sigma, (D : C).m \mapsto e & \textit{method environment} \\
M & ::= & [\bullet] \mid \lambda x : \sigma.M \mid \lambda\delta : Q.M \mid e\,M \mid M\,e \mid M\,d & \textit{evaluation context} \\
 & & \mid \Lambda a.M \mid M\,\sigma \mid \mathbf{let}\ \ x : \sigma = M\,\mathbf{in}\ \ e & \\
 & & \mid \mathbf{let}\ \ x : \sigma = e\,\mathbf{in}\ \ M &
\end{array}$$

Dictionaries

$$\begin{array}{lcll}
d & ::= & \delta \mid D\,\overline{\sigma}\,\overline{d} & \textit{dictionary} \\
dv & ::= & D\,\overline{\sigma}\,\overline{dv} & \textit{dictionary value}
\end{array}$$

A.6.2 $F_{\mathbf{D}}$ Judgments and Elaboration

$F_{\mathbf{D}}$ Type & Constraint Well-Formedness

$\boxed{\Gamma_C;\Gamma \vdash_Q Q \rightsquigarrow \sigma}$ *($F_{\mathbf{D}}$ Dictionary Type Well-Formedness)*

$$\textsc{oiQ-TC}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ \Gamma_C = \Gamma_{C1}, m : TC\, a : \sigma', \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma' \rightsquigarrow \sigma'\end{array}}{\Gamma_C;\Gamma \vdash_Q TC\,\sigma \rightsquigarrow [\sigma/a]\{m : \sigma'\}}$$

$\boxed{\Gamma_C;\Gamma \vdash_C C}$ *($F_{\mathbf{D}}$ Constraint Well-Formedness)*

$$\textsc{oiC-abs}\quad \frac{\begin{array}{c}\overline{\Gamma_C;\Gamma,\overline{a}_j \vdash_Q Q_i \rightsquigarrow \sigma_i}^{\,i\in 1..n} \\ \Gamma_C;\Gamma,\overline{a}_j \vdash_Q Q \rightsquigarrow \sigma \\ \overline{a}_j \notin \Gamma\end{array}}{\Gamma_C;\Gamma \vdash_C \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q}$$

$\boxed{\Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}$ *($F_{\mathbf{D}}$ Type Well-Formedness)*

$$\textsc{oiTy-bool}\quad \frac{}{\Gamma_C;\Gamma \vdash_{ty} Bool \rightsquigarrow Bool} \qquad \textsc{oiTy-var}\quad \frac{a \in \Gamma}{\Gamma_C;\Gamma \vdash_{ty} a \rightsquigarrow a}$$

$$\textsc{oiTy-arrow}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \\ \Gamma_C;\Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C;\Gamma \vdash_{ty} \sigma_1 \to \sigma_2 \rightsquigarrow \sigma_1 \to \sigma_2} \qquad \textsc{oiTy-qual}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \vdash_Q Q \rightsquigarrow \sigma' \\ \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma \vdash_{ty} Q \Rightarrow \sigma \rightsquigarrow \sigma' \to \sigma}$$

$$\textsc{oiTy-scheme}\quad \frac{\Gamma_C;\Gamma, a \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Gamma_C;\Gamma \vdash_{ty} \forall a.\sigma \rightsquigarrow \forall a.\sigma}$$

Dictionary Typing

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \ \rightsquigarrow e}$ *(Dictionary Typing)*

oD-var

$$\frac{\begin{array}{c}(\delta : Q) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \ \rightsquigarrow \delta}$$

oD-con

$$\frac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{Q}_i . e, \Sigma_2 \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_Q Q_i \ \rightsquigarrow \sigma'_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j}^{\,j} \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e : [\sigma_q / a]\sigma_m \rightsquigarrow e \\ \overline{\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_j / \overline{a}_j] Q_i \ \rightsquigarrow e_i}^{\,i}\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D\,\overline{\sigma}_j\,\overline{d}_i : TC\,[\overline{\sigma}_j / \overline{a}_j]\sigma_q \ \rightsquigarrow (\Lambda \overline{a}_j . \lambda\,\overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\})\,\overline{\sigma}_j\,\overline{e}_i}$$

F_{D} Term Typing

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e}$ *(F_{D} Term Typing)*

oiTm-true

$$\frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \rightsquigarrow \mathit{True}}$$

oiTm-false

$$\frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \rightsquigarrow \mathit{False}}$$

oiTm-var

$$\frac{\begin{array}{c}(x : \sigma) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma \rightsquigarrow x}$$

oiTm-let

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathbf{let}\ \ x : \sigma_1 = e_1\,\mathbf{in}\ \ e_2 : \sigma_2 \rightsquigarrow \mathbf{let}\ \ x : \sigma_1 = e_1\,\mathbf{in}\ \ e_2}$$

oiTm-method

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma \rightsquigarrow e \\ (m : TC\,a : \sigma') \in \Gamma_C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma' \rightsquigarrow e.m}$$

oiTm-arrI

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \lambda x : \sigma_1.e}$$

oiTm-arrE

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2}$$

oiTm-constrI

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : Q \vdash_{tm} e : \sigma \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda\delta : Q.e : Q \Rightarrow \sigma \rightsquigarrow \lambda\delta : \sigma.e}$$

oiTm-constrE

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : Q \Rightarrow \sigma \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1\, e_2}$$

oiTm-forallI

$$\frac{\Sigma; \Gamma_C; \Gamma, a \vdash_{tm} e : \sigma \rightsquigarrow e}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma \rightsquigarrow \Lambda a.e}$$

oiTm-forallE

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\,\sigma : [\sigma/a]\sigma' \rightsquigarrow e\,\sigma}$$

$F_{\mathbf{D}}$ Environment Well-Formedness

$\boxed{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}$ *($F_{\mathbf{D}}$ Environment Well-Formedness)*

oiCtx-empty

$$\frac{}{\vdash_{ctx} \bullet; \bullet; \bullet}$$

oiCtx-clsEnv

$$\frac{\begin{array}{c}\Gamma_C; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\, b \notin \mathbf{dom}(\Gamma_C) \\ \vdash_{ctx} \bullet; \Gamma_C; \bullet\end{array}}{\vdash_{ctx} \bullet; \Gamma_C, m : TC\,a : \sigma; \bullet}$$

oiCtx-tyEnvTm

$$\frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma\end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma}$$

oiCtx-tyEnvTy

$$\frac{a \notin \Gamma \qquad \vdash_{ctx} \bullet; \Gamma_C; \Gamma}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, a}$$

oiCtx-tyEnvD

$$\frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_Q TC\,\sigma \rightsquigarrow \sigma \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma\end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, \delta : TC\,\sigma}$$

$$\textsc{oiCtx-MEnv}\quad \frac{\begin{array}{c}\mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma)\\ \Gamma_C;\bullet \vdash_C \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma\\ (m : TC\,a : \sigma') \in \Gamma_C\\ \Sigma;\Gamma_C;\bullet \vdash_{tm} e : \forall \overline{a}_j.\overline{Q}_i \Rightarrow [\sigma/a]\sigma' \rightsquigarrow e\\ D \notin \mathbf{dom}(\Sigma)\\ (D' : \forall \overline{a}'_m.\overline{Q}''_n \Rightarrow TC\,\sigma'').m' \mapsto e' \notin \Sigma\\ \mathbf{where}[\overline{\sigma}_j/\overline{a}_j]\sigma = [\overline{\sigma}'_m/\overline{a}'_m]\sigma''\\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma\end{array}}{\vdash_{ctx} \Sigma, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma).m \mapsto e;\Gamma_C;\Gamma}$$

$\boxed{\mathbf{unambig}(C)}$ *(Unambiguity for Constraints)*

$$\textsc{oiUnambig-constraint}\quad \frac{\overline{a}_j \in \mathbf{fv}(\sigma)}{\mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma)}$$

F_{D} Environment Elaboration

$\boxed{\Gamma_C;\Gamma \rightsquigarrow \Gamma}$ *(F_{D}-to-$F_{\{\}}$ environment translation)*

$$\textsc{Ctx-Empty}\quad \frac{}{\Gamma_C;\bullet \rightsquigarrow \bullet}$$

$$\textsc{Ctx-Var}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \rightsquigarrow \Gamma\\ \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma, x : \sigma \rightsquigarrow \Gamma, x : \sigma}$$

$$\textsc{Ctx-DVar}\quad \frac{\begin{array}{c}\Gamma_C;\Gamma \rightsquigarrow \Gamma\\ \Gamma_C;\Gamma \vdash_C C \rightsquigarrow \sigma\end{array}}{\Gamma_C;\Gamma, \delta : C \rightsquigarrow \Gamma, \delta : \sigma}$$

$$\textsc{Ctx-TVar}\quad \frac{\Gamma_C;\Gamma \rightsquigarrow \Gamma}{\Gamma_C;\Gamma, a \rightsquigarrow \Gamma, a}$$

In the translation mechanism, we have assumed namespace translation functions which take a F_{D} type, term or dictionary-variable name and return the same identifier representing a $F_{\{\}}$ type or term variable. There are four such functions, each with a different namespace as domain:

Type variables: It translates a type variable of the $F_{\mathbf{D}}$ language, a, to the $F_{\{\}}$ type variable with the same name, a.

Term variables: Similar to type variables, but for the term sort.

Dictionary variables: It translates a dictionary variable, δ, to a $F_{\{\}}$ term variable with the same name.

Dictionary labels: It translates a dictionary method, m, to a record-field label, m, with the same name.

This identifier translation is assumed in all judgments that involve elaboration, such as the $F_{\mathbf{D}}$ term typing. When we regard identifiers, the font-color change implies such a translation. However, this convention is not used in other language sorts (types, non-variable terms, etc.). For example, two types with the same identifier but of different color mean only two types, a $F_{\mathbf{D}}$ and a $F_{\{\}}$ type, that are not related to each other. Any specification of the relation between the two types is given by the judgments they appear in.

$F_{\mathbf{D}}$ Operational Semantics

$\boxed{\Sigma \vdash e \longrightarrow e'}$ *($F_{\mathbf{D}}$ Evaluation)*

oiEval-app
$$\frac{\Sigma \vdash e_1 \longrightarrow e_1'}{\Sigma \vdash e_1\, e_2 \longrightarrow e_1'\, e_2}$$

oiEval-appAbs
$$\frac{}{\Sigma \vdash (\lambda x : \sigma . e_1)\, e_2 \longrightarrow [e_2/x]e_1}$$

oiEval-tyApp
$$\frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, \sigma \longrightarrow e'\, \sigma}$$

oiEval-tyAppAbs
$$\frac{}{\Sigma \vdash (\Lambda a . e)\, \sigma \longrightarrow [\sigma/a]e}$$

oiEval-DApp
$$\frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, d \longrightarrow e'\, d}$$

oiEval-DAppAbs
$$\frac{}{\Sigma \vdash (\lambda \delta : Q . e)\, d \longrightarrow [d/\delta]e}$$

oiEval-method
$$\frac{(D : C).m \mapsto e \in \Sigma}{\Sigma \vdash (D\, \overline{\sigma}\, \overline{d}).m \longrightarrow e\, \overline{\sigma}\, \overline{d}}$$

oiEval-let
$$\frac{}{\Sigma \vdash \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e_2 \longrightarrow [e_1/x]e_2}$$

$F_{\mathbf{D}}$ Context Typing and Elaboration

$\boxed{M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M}$ *($F_{\mathbf{D}}$ Context Typing)*

oiM-empty
$$\frac{}{[\bullet] : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \rightsquigarrow [\bullet]}$$

oiM-abs

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma', x : \sigma \Rightarrow \sigma_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{\lambda x : \sigma.M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma \to \sigma_2) \rightsquigarrow \lambda x : \sigma.M}$$

oiM-appL

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1 \to \sigma_2) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2 \end{array}}{M\, e_2 : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow M\, e_2}$$

oiM-appR

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1 \end{array}}{e_1\, M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow e_1\, M}$$

oiM-dictAbs

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', \delta : Q \Rightarrow \sigma_1) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{Q} Q \rightsquigarrow \sigma \end{array}}{\lambda \delta : Q.M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow Q \Rightarrow \sigma_1) \rightsquigarrow \lambda \delta : \sigma.M}$$

oiM-dictApp

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow Q \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma' \vdash_{d} d : Q \rightsquigarrow e \end{array}}{M\, d : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M\, e}$$

oiM-tyAbs

$$\frac{M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma', a \Rightarrow \sigma_2) \rightsquigarrow M}{\Lambda a.M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \forall a.\sigma_2) \rightsquigarrow \Lambda a.M}$$

oiM-tyApp

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \forall a.\sigma_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{M\, \sigma : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow [\sigma/a]\sigma_2) \rightsquigarrow M\, \sigma}$$

oiM-letL

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma', x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \end{array}}{\textbf{let } x : \sigma_1 = M \textbf{ in } e_2 : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow \textbf{let } x : \sigma_1 = M \textbf{ in } e_2}$$

oiM-letR

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \Rightarrow \sigma_2) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \end{array}}{\mathbf{let}\ \ x : \sigma_1 = e_1\ \mathbf{in}\ \ M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow \mathbf{let}\ \ x : \sigma_1 = e_1\ \mathbf{in}\ \ M}$$

A.7 $F_{\mathbf{D}}^{\Rightarrow}$ Additional Definitions

A.7.1 Syntax

$$\begin{array}{lcll}
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma . e \mid e_1\, e_2 \mid \lambda \delta : C . e \mid e\, d & \textit{expression} \\
 & & \Lambda a . e \mid e\, \sigma \mid d . m \mid \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e_2 & \\
v & ::= & \mathit{True} \mid \mathit{False} \mid \lambda x : \sigma . e \mid \lambda \delta : C . e \mid \Lambda a . e & \textit{value}
\end{array}$$

$$\begin{array}{lcll}
\sigma & ::= & \mathit{Bool} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid C \Rightarrow \sigma \mid \forall a . \sigma & \textit{type} \\
Q & ::= & \mathit{TC}\, \sigma & \textit{class constraint} \\
C & ::= & Q \mid C_1 \Rightarrow C_2 \mid \forall a . C & \textit{constraint}
\end{array}$$

$$\begin{array}{lcll}
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : C & \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \mathit{TC}\, a : \sigma & \textit{class environment} \\
\Sigma & ::= & \bullet \mid \Sigma, (D : C) . m \mapsto e & \textit{method environment} \\
M & ::= & [\bullet] \mid \lambda x : \sigma . M \mid \lambda \delta : C . M \mid e\, M \mid M\, e \mid M\, d & \textit{evaluation context} \\
 & & \mid \Lambda a . M \mid M\, \sigma \mid \mathbf{let}\ \ x : \sigma = M\ \mathbf{in}\ \ e & \\
 & & \mid \mathbf{let}\ \ x : \sigma = e\, \mathbf{in}\ \ M &
\end{array}$$

Dictionaries

$$\begin{array}{lcll}
d & ::= & \delta \mid D \mid \lambda \delta : C . d \mid d_1\, d_2 \mid \Lambda a . d \mid d\, \sigma & \textit{dictionary} \\
dv & ::= & D\, \overline{\sigma}\, \overline{d} \mid \lambda \delta : C . d \mid \Lambda a . d & \textit{dictionary value}
\end{array}$$

A.7.2 F_{D} Judgments and Elaboration

F_{D} Type & Constraint Well-Formedness

$\boxed{\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma}$ *(F_{D} Dictionary Type Well-Formedness)*

$$\textsc{iQ-TC}\quad \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ \Gamma_C = \Gamma_{C1}, m : TC\, a : \sigma', \Gamma_{C2} \\ \Gamma_{C1}; \bullet, a \vdash_{ty} \sigma' \rightsquigarrow \sigma'\end{array}}{\Gamma_C; \Gamma \vdash_Q TC\, \sigma \rightsquigarrow [\sigma/a]\{m : \sigma'\}}$$

$\boxed{\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma}$ *(F_{D} Constraint Well-Formedness)*

$$\textsc{iC-forall}\quad \frac{\begin{array}{c}\Gamma_C; \Gamma, a \vdash_C C \rightsquigarrow \sigma \\ a \notin \Gamma\end{array}}{\Gamma_C; \Gamma \vdash_C \forall a.C \rightsquigarrow \forall a.\sigma}$$

$$\textsc{iC-arrow}\quad \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1 \\ \Gamma_C; \Gamma \vdash_C C_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C; \Gamma \vdash_C C_1 \Rightarrow C_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{iC-classConstr}\quad \frac{\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma}{\Gamma_C; \Gamma \vdash_C Q \rightsquigarrow \sigma}$$

$\boxed{\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}$ *(F_{D} Type Well-Formedness)*

$$\textsc{iTy-bool}\quad \frac{}{\Gamma_C; \Gamma \vdash_{ty} Bool \rightsquigarrow Bool}$$

$$\textsc{iTy-var}\quad \frac{a \in \Gamma}{\Gamma_C; \Gamma \vdash_{ty} a \rightsquigarrow a}$$

$$\textsc{iTy-arrow}\quad \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2\end{array}}{\Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{iTy-qual}\quad \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma' \\ \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Gamma_C; \Gamma \vdash_{ty} C \Rightarrow \sigma \rightsquigarrow \sigma' \rightarrow \sigma}$$

$$\textsc{iTy-scheme}\quad \frac{\Gamma_C; \Gamma, a \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Gamma_C; \Gamma \vdash_{ty} \forall a.\sigma \rightsquigarrow \forall a.\sigma}$$

Dictionary Typing

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : C \rightsquigarrow e}$ *(Dictionary Typing)*

D-VAR
$$\frac{(\delta : C) \in \Gamma \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C \rightsquigarrow \delta}$$

D-CON
$$\frac{\begin{array}{c} \Sigma = \Sigma_1, (D : \forall \bar{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \bar{a}_j.\lambda \bar{\delta}_i : \overline{C}_i.e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \bar{a}_j \vdash_C C_i \rightsquigarrow \sigma'_i}^{\,i} \\ \Sigma_1; \Gamma_C; \bullet, \bar{a}_j, \bar{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q / a]\sigma_m \rightsquigarrow e \end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \bar{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \rightsquigarrow \Lambda \bar{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\}}$$

D-DABS
$$\frac{\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e \qquad \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1}{\Sigma; \Gamma_C; \Gamma \vdash_d \lambda \delta : C_1.d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda \delta : \sigma_1.e}$$

D-DAPP
$$\frac{\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \qquad \Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2}{\Sigma; \Gamma_C; \Gamma \vdash_d d_1\,d_2 : C_2 \rightsquigarrow e_1\,e_2}$$

D-TYABS
$$\frac{\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e}{\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e}$$

D-TYAPP
$$\frac{\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow e \qquad \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Sigma; \Gamma_C; \Gamma \vdash_d d\,\sigma : [\sigma / a]C \rightsquigarrow e\,\sigma}$$

$F_{\mathbf{D}}$ Term Typing

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e}$ *($F_{\mathbf{D}}$ Term Typing)*

ITM-TRUE
$$\frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \rightsquigarrow \mathit{True}}$$

ITM-FALSE
$$\frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \rightsquigarrow \mathit{False}}$$

iTm-var

$$\frac{\begin{array}{c}(x : \sigma) \in \Gamma \\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} x : \sigma \rightsquigarrow x}$$

iTm-let

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \\ \Sigma;\Gamma_C;\Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \\ \Gamma_C;\Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathbf{let}\ \ x : \sigma_1 = e_1\, \mathbf{in}\ \ e_2 : \sigma_2 \rightsquigarrow \mathbf{let}\ \ x : \sigma_1 = e_1\, \mathbf{in}\ \ e_2}$$

iTm-method

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d : TC\,\sigma\ \rightsquigarrow e \\ (m : TC\,a : \sigma') \in \Gamma_C\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} d.m : [\sigma/a]\sigma' \rightsquigarrow e.m}$$

iTm-arrI

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \rightsquigarrow e \\ \Gamma_C;\Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.e}$$

iTm-arrE

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1 \\ \Sigma;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2}$$

iTm-constrI

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma, \delta : C \vdash_{tm} e : \sigma \rightsquigarrow e \\ \Gamma_C;\Gamma \vdash_C C\ \rightsquigarrow \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma \rightsquigarrow \lambda\delta : \sigma.e}$$

iTm-constrE

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : C \Rightarrow \sigma \rightsquigarrow e_1 \\ \Sigma;\Gamma_C;\Gamma \vdash_d d : C\ \rightsquigarrow e_2\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1\, e_2}$$

iTm-forallI

$$\frac{\Sigma;\Gamma_C;\Gamma, a \vdash_{tm} e : \sigma \rightsquigarrow e}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma \rightsquigarrow \Lambda a.e}$$

iTm-forallE

$$\frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \forall a.\sigma' \rightsquigarrow e \\ \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\,\sigma : [\sigma/a]\sigma' \rightsquigarrow e\,\sigma}$$

F_{D} Environment Well-Formedness

$\boxed{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}$ *(F_{D} Environment Well-Formedness)*

$$\textsc{iCtx-empty}\quad \frac{}{\vdash_{ctx} \bullet; \bullet; \bullet}$$

$$\textsc{iCtx-clsEnv}\quad \frac{\begin{array}{c} \Gamma_C; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\, b \notin \mathbf{dom}(\Gamma_C) \\ \vdash_{ctx} \bullet; \Gamma_C; \bullet \end{array}}{\vdash_{ctx} \bullet; \Gamma_C, m : TC\, a : \sigma; \bullet}$$

$$\textsc{iCtx-tyEnvTm}\quad \frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma \end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma}$$

$$\textsc{iCtx-tyEnvTy}\quad \frac{a \notin \Gamma \qquad \vdash_{ctx} \bullet; \Gamma_C; \Gamma}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, a}$$

$$\textsc{iCtx-tyEnvD}\quad \frac{\begin{array}{c} \Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash_{ctx} \bullet; \Gamma_C; \Gamma \end{array}}{\vdash_{ctx} \bullet; \Gamma_C; \Gamma, \delta : C}$$

$$\textsc{iCtx-MEnv}\quad \frac{\begin{array}{c} \mathbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma) \\ \Gamma_C; \bullet \vdash_C \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma \rightsquigarrow \sigma \\ (m : TC\, a : \sigma') \in \Gamma_C \\ \Sigma; \Gamma_C; \bullet \vdash_{tm} e : \forall \overline{a}_j . \overline{C}_i \Rightarrow [\sigma / a]\sigma' \rightsquigarrow e \\ D \notin \mathbf{dom}(\Sigma) \\ (D' : \forall \overline{a}'_m . \overline{C}''_n \Rightarrow TC\, \sigma'').m' \mapsto e' \notin \Sigma \\ \mathbf{where}[\overline{\sigma}_j / \overline{a}_j]\sigma = [\overline{\sigma}'_m / \overline{a}'_m]\sigma'' \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \end{array}}{\vdash_{ctx} \Sigma, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma).m \mapsto e; \Gamma_C; \Gamma}$$

$\boxed{\mathbf{unambig}(C)}$ *(Unambiguity for Constraints)*

$$\textsc{iUnambig-constraint}\quad \frac{\overline{a}_j \in \mathbf{fv}(\sigma)}{\mathbf{unambig}(\forall \overline{a}_j . \overline{Q}_i \Rightarrow TC\, \sigma)}$$

$F_{\mathbf{D}}$ Environment Elaboration

$\boxed{\Gamma_C ; \Gamma \rightsquigarrow \Gamma}$ *($F_{\mathbf{D}}$-to-$F_{\{\}}$ environment translation)*

$$\textsc{Ctx-Empty}\quad \frac{}{\Gamma_C ; \bullet \rightsquigarrow \bullet}$$

$$\textsc{Ctx-Var}\quad \frac{\Gamma_C ; \Gamma \rightsquigarrow \Gamma \qquad \Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Gamma_C ; \Gamma, x : \sigma \rightsquigarrow \Gamma, x : \sigma}$$

$$\textsc{Ctx-DVar}\quad \frac{\Gamma_C ; \Gamma \rightsquigarrow \Gamma \qquad \Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma}{\Gamma_C ; \Gamma, \delta : C \rightsquigarrow \Gamma, \delta : \sigma}$$

$$\textsc{Ctx-TVar}\quad \frac{\Gamma_C ; \Gamma \rightsquigarrow \Gamma}{\Gamma_C ; \Gamma, a \rightsquigarrow \Gamma, a}$$

In the translation mechanism, we have assumed namespace translation functions which take a $F_{\mathbf{D}}$ type, term or dictionary-variable name and return the same identifier representing a $F_{\{\}}$ type or term variable. There are four such functions, each with a different namespace as domain:

Type variables: It translates a type variable of the $F_{\mathbf{D}}$ language, a, to the $F_{\{\}}$ type variable with the same name, a.

Term variables: Similar to type variables, but for the term sort.

Dictionary variables: It translates a dictionary variable, δ, to a $F_{\{\}}$ term variable with the same name.

Dictionary labels: It translates a dictionary method, m, to a record-field label, m, with the same name.

This identifier translation is assumed in all judgments that involve elaboration, such as the $F_{\mathbf{D}}$ term typing. When we regard identifiers, the font-color change implies such a translation. However, this convention is not used in other language sorts (types, non-variable terms, etc.). For example, two types with the same identifier but of different color mean only two types, a $F_{\mathbf{D}}$ and a $F_{\{\}}$ type, that are not related to each other. Any specification of the relation between the two types is given by the judgments they appear in.

$F_{\mathbf{D}}$ Operational Semantics

$\boxed{\Sigma \vdash e \longrightarrow e'}$ *($F_{\mathbf{D}}$ Evaluation)*

iEval-app
$$\frac{\Sigma \vdash e_1 \longrightarrow e_1'}{\Sigma \vdash e_1\, e_2 \longrightarrow e_1'\, e_2}$$

iEval-appAbs
$$\frac{}{\Sigma \vdash (\lambda x : \sigma.e_1)\, e_2 \longrightarrow [e_2/x]e_1}$$

iEval-tyApp
$$\frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, \sigma \longrightarrow e'\, \sigma}$$

iEval-tyAppAbs
$$\frac{}{\Sigma \vdash (\Lambda a.e)\, \sigma \longrightarrow [\sigma/a]e}$$

iEval-DApp
$$\frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, d \longrightarrow e'\, d}$$

iEval-DAppAbs
$$\frac{}{\Sigma \vdash (\lambda \delta : C.e)\, d \longrightarrow [d/\delta]e}$$

iEval-method
$$\frac{d \longrightarrow d'}{\Sigma \vdash d.m \longrightarrow d'.m}$$

iEval-methodVal
$$\frac{(D : C).m \mapsto e \in \Sigma}{\Sigma \vdash (D\, \overline{\sigma}_m\, \overline{d}_n).m \longrightarrow e\, \overline{\sigma}_m\, \overline{d}_n}$$

iEval-let
$$\frac{}{\Sigma \vdash \mathbf{let}\ \ x : \sigma = e_1\, \mathbf{in}\ \ e_2 \longrightarrow [e_1/x]e_2}$$

$\boxed{\Sigma \vdash e \longrightarrow^* e'}$ *($F_{\mathbf{D}}$ Reduction)*

iReduce-step
$$\frac{\begin{array}{c}\Sigma \vdash e_1 \longrightarrow e_2 \\ \Sigma \vdash e_2 \longrightarrow^* e_3\end{array}}{\Sigma \vdash e_1 \longrightarrow^* e_3}$$

iReduce-stop
$$\frac{}{\Sigma \vdash e_1 \longrightarrow^* e_1}$$

$\boxed{d \longrightarrow d'}$ *($F_{\mathbf{D}}$ Dictionary Evaluation)*

iDictEval-app
$$\frac{d_1 \longrightarrow d_1'}{d_1\, d_2 \longrightarrow d_1'\, d_2}$$

iDictEval-appAbs
$$\frac{}{(\lambda \delta : C.d_1)\, d_2 \longrightarrow [d_2/\delta]d_1}$$

iDictEval-tyApp
$$\frac{d \longrightarrow d'}{d\, \sigma \longrightarrow d'\, \sigma}$$

iDictEval-tyAppAbs
$$\frac{}{(\Lambda a.d)\, \sigma \longrightarrow [\sigma/a]d}$$

$\boxed{d \longrightarrow^{*} d'}$ *(F_{D} Dictionary Reduction)*

iDictReduce-step

$$\frac{\begin{array}{c} d_1 \longrightarrow d_2 \\ d_2 \longrightarrow^{*} d_3 \end{array}}{d_1 \longrightarrow^{*} d_3}$$

iDictReduce-stop

$$\frac{}{d_1 \longrightarrow^{*} d_1}$$

F_{D} Context Typing and Elaboration

$\boxed{M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma') \rightsquigarrow M}$ *(F_{D} Context Typing)*

iM-empty

$$\frac{}{[\bullet] : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \rightsquigarrow [\bullet]}$$

iM-abs

$$\frac{\begin{array}{c} M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma_1) \mapsto (\Sigma;\Gamma_C;\Gamma', x : \sigma \Rightarrow \sigma_2) \rightsquigarrow M \\ \Gamma_C;\Gamma' \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{\lambda x : \sigma.M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma_1) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma \to \sigma_2) \rightsquigarrow \lambda x : \sigma.M}$$

iM-appL

$$\frac{\begin{array}{c} M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma_1 \to \sigma_2) \rightsquigarrow M \\ \Sigma;\Gamma_C;\Gamma' \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2 \end{array}}{M\ e_2 : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma_2) \rightsquigarrow M\ e_2}$$

iM-appR

$$\frac{\begin{array}{c} M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma;\Gamma_C;\Gamma' \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1 \end{array}}{e_1\ M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma_2) \rightsquigarrow e_1\ M}$$

iM-dictAbs

$$\frac{\begin{array}{c} M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma', \delta : C \Rightarrow \sigma_1) \rightsquigarrow M \\ \Gamma_C;\Gamma' \vdash_{C} C \rightsquigarrow \sigma \end{array}}{\lambda \delta : C.M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow C \Rightarrow \sigma_1) \rightsquigarrow \lambda \delta : \sigma.M}$$

iM-dictApp

$$\frac{\begin{array}{c} M : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow C \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma;\Gamma_C;\Gamma' \vdash_{d} d : C \rightsquigarrow e \end{array}}{M\ d : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma_1) \rightsquigarrow M\ e}$$

iM-tyAbs

$$\frac{M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma', a \Rightarrow \sigma_2) \rightsquigarrow M}{\Lambda a.M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \forall a.\sigma_2) \rightsquigarrow \Lambda a.M}$$

iM-tyApp

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \forall a.\sigma_2) \rightsquigarrow M \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma \rightsquigarrow \sigma \end{array}}{M\,\sigma : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma_1) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow [\sigma/a]\sigma_2) \rightsquigarrow M\,\sigma}$$

iM-letL

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma', x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \end{array}}{\textbf{let } x : \sigma_1 = M \textbf{ in } e_2 : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow \textbf{let } x : \sigma_1 = M \textbf{ in } e_2}$$

iM-letR

$$\frac{\begin{array}{c} M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \Rightarrow \sigma_2) \rightsquigarrow M \\ \Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \\ \Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \end{array}}{\textbf{let } x : \sigma_1 = e_1 \textbf{ in } M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_2) \rightsquigarrow \textbf{let } x : \sigma_1 = e_1 \textbf{ in } M}$$

A.8 $F_{\{\}}$ Additional Definitions

A.8.1 Syntax

$$\begin{array}{lcll}
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma.e \mid e_1\, e_2 \mid \Lambda a.e \mid e\,\sigma & \textit{target term} \\
 & & \mid \{\, \overline{m_i = e_i}^{\,i<n} \,\} \mid e.m \mid \textbf{let } x : \sigma = e_1 \textbf{ in } e_2 & \\
v & ::= & \mathit{True} \mid \mathit{False} \mid \lambda x : \sigma.e \mid \Lambda a.e \mid \{\, \overline{m_i = e_i}^{\,i<n} \,\} & \textit{target value} \\
\sigma & ::= & \mathit{Bool} \mid a \mid \forall a.\sigma \mid \sigma_1 \to \sigma_2 \mid \{\, \overline{m_i : \sigma_i}^{\,i<n} \,\} & \textit{target type} \\
\Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & \textit{target context} \\
M & ::= & [\bullet] \mid \lambda x : \sigma.M \mid e\,M \mid M\,e \mid \Lambda a.M \mid M\,\sigma & \textit{target evaluation context} \\
 & & \mid \{\, \overline{m_i = M_i}^{\,i \in 1..n} \,\} \mid M.m & \\
 & & \mid \textbf{let } x : \sigma = M \textbf{ in } e \mid \textbf{let } x : \sigma = e \textbf{ in } M &
\end{array}$$

A.8.2 $F_{\{\}}$ Judgments

$F_{\{\}}$ Type Well-Formedness

$\boxed{\Gamma \vdash_{ty} \sigma}$ *(Well-formed $F_{\{\}}$ types)*

$$\textsc{tTy-Bool}\quad \frac{}{\Gamma \vdash_{ty} \mathit{Bool}}$$

$$\textsc{tTy-Var}\quad \frac{}{\Gamma \vdash_{ty} a}$$

$$\textsc{tTy-Abs}\quad \frac{\Gamma, a \vdash_{ty} \sigma}{\Gamma \vdash_{ty} \forall a.\sigma}$$

$$\textsc{tTy-Arr}\quad \frac{\Gamma \vdash_{ty} \sigma_1 \qquad \Gamma \vdash_{ty} \sigma_2}{\Gamma \vdash_{ty} \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{tTy-Rec}\quad \frac{\overline{\Gamma \vdash_{ty} \sigma_i}^{\,i<n}}{\Gamma \vdash_{ty} \{\,\overline{m_i : \sigma_i}^{\,i<n}\,\}}$$

$F_{\{\}}$ Term Typing

$\boxed{\Gamma \vdash_{tm} e : \sigma}$ *(Well typed $F_{\{\}}$ terms)*

$$\textsc{tTm-True}\quad \frac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}}$$

$$\textsc{tTm-False}\quad \frac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} \mathit{False} : \mathit{Bool}}$$

$$\textsc{tTm-Var}\quad \frac{(x : \sigma) \in \Gamma \qquad \vdash_{ctx} \Gamma}{\Gamma \vdash_{tm} x : \sigma}$$

$$\textsc{tTm-Abs}\quad \frac{\Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2}{\Gamma \vdash_{tm} \lambda x : \sigma_1 . e : \sigma_1 \rightarrow \sigma_2}$$

$$\textsc{tTm-App}\quad \frac{\Gamma \vdash_{tm} e_1 : \sigma \rightarrow \sigma' \qquad \Gamma \vdash_{tm} e_2 : \sigma}{\Gamma \vdash_{tm} e_1\, e_2 : \sigma'}$$

$$\textsc{tTm-TAbs}\quad \frac{\Gamma, a \vdash_{tm} e : \sigma}{\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma}$$

$$\textsc{tTm-TApp}\quad \frac{\Gamma \vdash_{tm} e : \forall a.\sigma_1 \qquad \Gamma \vdash_{ty} \sigma_2}{\Gamma \vdash_{tm} e\, \sigma : [\sigma_2/a]\sigma_1}$$

$$\textsc{tTm-Rec}\quad \frac{\overline{\Gamma \vdash_{tm} e_i : \sigma_i}^{\,i<n}}{\Gamma \vdash_{tm} \{\,\overline{m_i = e_i}^{\,i<n}\,\} : \{\,\overline{m_i : \sigma_i}^{\,i<n}\,\}}$$

$$\textsc{tTm-Proj}\quad \frac{\Gamma \vdash_{tm} e : \{\,\overline{m_i : \sigma_i}^{\,i<n}\,\}}{\Gamma \vdash_{tm} e.m_j : \sigma_j}$$

$$\textsc{tTm-Let}\quad \frac{\Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \qquad \Gamma \vdash_{tm} e_1 : \sigma_1}{\Gamma \vdash_{tm} \textbf{let}\ \ x : \sigma_1 = e_1\ \textbf{in}\ \ e_2 : \sigma_2}$$

$F_{\{\}}$ Environment Well-Formedness

$\boxed{\vdash_{ctx} \Gamma}$ *(Well-formed $F_{\{\}}$ environment)*

$$\textsc{tCx-Empty}\quad \frac{}{\vdash_{ctx} \bullet}$$

$$\textsc{tCx-TVar}\quad \frac{\vdash_{ctx} \Gamma \qquad a \notin \Gamma}{\vdash_{ctx} \Gamma, a}$$

$$\textsc{tCx-Var}\quad \frac{\begin{array}{c}\vdash_{ctx} \Gamma \\ \Gamma \vdash_{ty} \sigma \qquad x \notin \Gamma\end{array}}{\vdash_{ctx} \Gamma, x : \sigma}$$

$F_{\{\}}$ Operational Semantics

$\boxed{e \longrightarrow e'}$ *($F_{\{\}}$ evaluation)*

$$\textsc{tEval-AppAbs}\quad \frac{}{(\lambda x : \sigma . e_1)\, e_2 \longrightarrow [e_1/x]e_2}$$

$$\textsc{tEval-TappTabs}\quad \frac{}{(\Lambda a . e)\, \sigma \longrightarrow [\sigma/a]e}$$

$$\textsc{tEval-Proj}\quad \frac{}{\{\, \overline{m_i = e_i}^{\,i \in 1..n} \,\}.m_j \longrightarrow e_j}$$

$$\textsc{tEval-Let}\quad \frac{}{\textbf{let}\ \ x : \sigma = e_1\ \textbf{in}\ \ e_2 \longrightarrow [e_1/x]e_2}$$

$$\textsc{tEval-App}\quad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}$$

$$\textsc{tEval-Tapp}\quad \frac{e_1 \longrightarrow e_1'}{e_1\, \sigma \longrightarrow e_1'\, \sigma}$$

$$\textsc{tEval-Rec}\quad \frac{e \longrightarrow e'}{e.m_j \longrightarrow e'.m_j}$$

$\boxed{e \longrightarrow^* e'}$ *($F_{\{\}}$ Reduction)*

$$\textsc{tReduce-step}\quad \frac{\begin{array}{c}e_1 \longrightarrow e_2 \\ e_2 \longrightarrow^* e_3\end{array}}{e_1 \longrightarrow^* e_3}$$

$$\textsc{tReduce-stop}\quad \frac{}{e_1 \longrightarrow^* e_1}$$

$F_{\{\}}$ Context Typing

$\boxed{M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma')}$ *($F_{\{\}}$ Context Typing)*

tM-empty

$$\frac{}{[\bullet] : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma \Rightarrow \sigma)}$$

tM-abs

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma', x : \sigma \Rightarrow \sigma_2) \\ \Gamma' \vdash_{ty} \sigma \end{array}}{\lambda x : \sigma . M : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma' \Rightarrow \sigma \to \sigma_2)}$$

tM-appL

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_1 \to \sigma_2) \\ \Gamma' \vdash_{tm} e_2 : \sigma_1 \end{array}}{M\ e_2 : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_2)}$$

tM-appR

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_1) \\ \Gamma' \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \end{array}}{e_1\ M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_2)}$$

tM-tyAbs

$$\frac{M : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma', a \Rightarrow \sigma_2)}{\Lambda a . M : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma' \Rightarrow \forall a . \sigma_2)}$$

tM-tyApp

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma' \Rightarrow \forall a . \sigma_2) \\ \Gamma' \vdash_{ty} \sigma \end{array}}{M\ \sigma : (\Gamma \Rightarrow \sigma_1) \mapsto (\Gamma' \Rightarrow [\sigma / a] \sigma_2)}$$

tM-letL

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_1) \\ \Gamma', x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \\ \Gamma' \vdash_{ty} \sigma_1 \end{array}}{\textbf{let}\ \ x : \sigma_1 = M\ \textbf{in}\ \ e_2 : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_2)}$$

tM-letR

$$\frac{\begin{array}{c} M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma', x : \sigma_1 \Rightarrow \sigma_2) \\ \Gamma' \vdash_{tm} e_1 : \sigma_1 \\ \Gamma' \vdash_{ty} \sigma_1 \end{array}}{\textbf{let}\ \ x : \sigma_1 = e_1\ \textbf{in}\ \ M : (\Gamma \Rightarrow \sigma) \mapsto (\Gamma' \Rightarrow \sigma_2)}$$

A.9 System F with Data Types Definitions

Both the typing rules and call-by-name operational semantics for System F are entirely standard and can be found elsewhere, we include them here to keep the presentation self-contained. In the following, we denote System F typing environments by Δ:

$\Delta \quad ::= \quad \bullet \mid \Delta, T \mid \Delta, K : \upsilon \mid \Delta, a \mid \Delta, x : \upsilon$ *typing environment*

A.9.1 Term Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \upsilon}$ (Term Typing)

$$\frac{(x : \upsilon) \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} x : \upsilon}\ \textsc{TmVar} \qquad \frac{\begin{array}{c} x \notin \mathit{dom}(\Delta) \\ \Delta, x : \upsilon_1 \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \upsilon_2 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1 \end{array}}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} \lambda(x : \upsilon_1).t : \upsilon_1 \to \upsilon_2}\ (\to\mathrm{I})$$

$$\frac{(K : \upsilon) \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} K : \upsilon}\ \textsc{TmCon} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t_1 : \upsilon_1 \to \upsilon_2 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t_2 : \upsilon_1}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t_1\ t_2 : \upsilon_2}\ (\to\mathrm{E})$$

$$\frac{a \notin \Delta \qquad \Delta, a \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} \Lambda a.t : \forall a.\upsilon}\ (\forall\mathrm{I}) \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \forall a.\upsilon \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t\ \upsilon_1 : [\upsilon_1/a]\upsilon}\ (\forall\mathrm{E})$$

$$\frac{\begin{array}{c} x \notin \mathit{dom}(\Delta) \qquad \Delta, x : \upsilon_1 \vdash^{\mathsf{F}}_{\mathtt{tm}} t_1 : \upsilon_1 \\ \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1 \qquad \Delta, x : \upsilon_1 \vdash^{\mathsf{F}}_{\mathtt{tm}} t_2 : \upsilon_2 \end{array}}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} (\mathbf{let}\ x : \upsilon_1 = t_1\ \mathbf{in}\ t_2) : \upsilon_2}\ \textsc{TmLet}$$

$$\frac{\begin{array}{c} \Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t_1 : T\ \upsilon \\ \overline{x} \notin \mathit{dom}(\Delta) \qquad (K : \forall a.\overline{\upsilon} \to T\ a) \in \Delta \qquad \Delta, \overline{x : [\upsilon/a]\upsilon} \vdash^{\mathsf{F}}_{\mathtt{tm}} t_2 : \upsilon_2 \end{array}}{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} (\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2) : \upsilon_2}\ \textsc{TmCase}$$

A.9.2 Well-formedness of Types

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon}$ (Type Well-formedness)

$$\frac{a \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} a}\ \textsc{TyVar} \qquad \frac{T \in \Delta}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} T}\ \textsc{TyCon} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_2}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1 \to \upsilon_2}\ \textsc{TyArr}$$

$$\frac{a \notin \Delta \qquad \Delta, a \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \forall a.\upsilon}\ \textsc{TyAll} \qquad \frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1 \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_2}{\Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon_1\ \upsilon_2}\ \textsc{TyApp}$$

A.9.3 Program Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathtt{pgm}} \mathit{fpgm} : \upsilon}$ (Program Typing)

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{pgm}} t : \upsilon}\ \textsc{PgmExpr}$$

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{val}} \mathit{fval} : \Delta_{\upsilon} \qquad \Delta, \Delta_{\upsilon} \vdash^{\mathsf{F}}_{\mathtt{pgm}} \mathit{fpgm} : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{pgm}} (\mathit{fval}; \mathit{fpgm}) : \upsilon}\ \textsc{PgmVal}$$

$$\frac{\Delta \vdash^{\mathsf{F}}_{\mathtt{data}} \mathit{fdata} : \Delta_{d} \qquad \Delta, \Delta_{d} \vdash^{\mathsf{F}}_{\mathtt{pgm}} \mathit{fpgm} : \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{pgm}} (\mathit{fdata}; \mathit{fpgm}) : \upsilon}\ \textsc{PgmData}$$

For brevity, if $\Delta = \bullet$ we denote System F program typing as $\vdash^{\mathsf{F}}_{\mathtt{pgm}} \mathit{fpgm} : \upsilon$.

A.9.4 Value Binding Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathtt{val}} \mathit{fval} : \Delta_{\mathit{fval}}}$ (Value Binding Typing)

$$\frac{x \notin \mathit{dom}(\Delta) \qquad \Delta, x : \upsilon \vdash^{\mathsf{F}}_{\mathtt{tm}} t : \upsilon \qquad \Delta \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon}{\Delta \vdash^{\mathsf{F}}_{\mathtt{val}} (\mathbf{let}\ x : \upsilon = t) : [x : \upsilon]}\ \textsc{Val}$$

A.9.5 Datatype Declaration Typing

$\boxed{\Delta \vdash^{\mathsf{F}}_{\mathtt{data}} \mathit{fdata} : \Delta_{\mathit{fdata}}}$ (Datatype Declaration Typing)

$$\frac{\overline{\Delta, a \vdash^{\mathsf{F}}_{\mathtt{ty}} \upsilon}}{\Delta \vdash^{\mathsf{F}}_{\mathtt{val}} (\mathbf{data}\ T\ a = K\ \overline{\upsilon}) : [T, K : \forall a. \overline{\upsilon} \rightarrow T\ a]}\ \textsc{Data}$$

A.9.6 Call-by-name Operational Semantics

The small-step, call-by-name operational semantics of System F are presented below:

$\boxed{t \longrightarrow t'}$ (Operational Semantics (Small-step))

$$\frac{}{(\Lambda a.t)\ \upsilon \longrightarrow [\upsilon/a]t}\ \textsc{TyBeta} \qquad \frac{}{(\lambda(x:\upsilon).t)\ t' \longrightarrow [t'/x]t}\ \textsc{TmBeta}$$

$$\frac{t_1 \longrightarrow t_1'}{(\mathbf{case}\ t_1\ \mathbf{of}\ K\ \overline{x} \to t_2) \longrightarrow (\mathbf{case}\ t_1'\ \mathbf{of}\ K\ \overline{x} \to t_2)}\ \textsc{CaseStep}$$

$$\frac{}{(\mathbf{case}\ K\ \bar{t}\ \mathbf{of}\ K\ \overline{x} \to t) \longrightarrow [\bar{t}/\overline{x}]t}\ \textsc{CaseBeta}$$

$$\frac{}{(\mathbf{let}\ x:\upsilon = t_1\ \mathbf{in}\ t_2) \longrightarrow [\mathbf{let}\ x:\upsilon = t_1\ \mathbf{in}\ t_1/x]t_2}\ \textsc{LetBeta}$$

Appendix B

Stability Proofs

This chapter provides the proofs for the properties discussed in Section 5.3.

B.1 Let-Inlining and Extraction

Property 1 (Let Inlining is Type Preserving)**.**

- *If* $\Gamma \vdash \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2 \Rightarrow \eta^\epsilon$ *then* $\Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta^\epsilon$
- *If* $\Gamma \vdash \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2 \Leftarrow \sigma$ *then* $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma$

Before proving Property 1, we first introduce a number of helper lemmas:

Lemma 1 (Expression Inlining is Type Preserving (Synthesis))**.**
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Rightarrow \eta_2^\epsilon$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, e_2 \Rightarrow \eta_2^\epsilon$

Lemma 2 (Expression Inlining is Type Preserving (Checking))**.**
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Leftarrow \sigma_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, e_2 \Leftarrow \sigma_2$

Lemma 3 (Head Inlining is Type Preserving)**.**
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^H h \Rightarrow \sigma_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $\Gamma_1, \Gamma_2 \vdash^H [e_1/x]\, h \Rightarrow \sigma_2$

Lemma 4 (Argument Inlining is Type Preserving)**.**
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma_1, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$
where $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $\Gamma_1, \Gamma_2 \vdash^A [e_1/x]\, \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$

$$\begin{tikzcd}
 & \Gamma \vdash e \Rightarrow \eta^{\epsilon} \arrow[loop above] \arrow[r] \arrow[d, shift left] \arrow[dl, bend left] \arrow[dr, shift left] & \Gamma \vdash e \Leftarrow \sigma \arrow[loop above] \arrow[d, shift left] \arrow[dl] \\
\Gamma \vdash^{H} h \Rightarrow \sigma \arrow[ur, bend left] & & \\
 & \Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2 \arrow[loop below] \arrow[u, shift left] & \Gamma \vdash decl \Rightarrow \Gamma' \arrow[u, shift left] \arrow[ul, shift left]
\end{tikzcd}$$

Figure B.1: Relation dependencies

Lemma 5 (Declaration Inlining is Type Preserving)**.**
If $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^{\epsilon}$ *and* $\Gamma_1, x : \forall \overline{\{a\}}.\eta_1^{\epsilon}, \Gamma_2 \vdash decl \Rightarrow \Gamma_3$
where $\overline{a} = \boldsymbol{fv}\,(\eta_1^{\epsilon}) \setminus \boldsymbol{dom}\,(\Gamma_1)$ *then* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, decl \Rightarrow \Gamma_3$

Figure B.1 shows the dependencies between the different relations, and by extension the different helper lemmas. An arrow from A to B denotes that B depends on A. Note that these 5 lemmas need to be proven through mutual induction. The proof proceeds by structural induction on the second typing derivation. While the number of cases gets quite large, each case is entirely trivial.

Using these additional lemmas, we then continue proving Property 1. By case analysis on the premise (rule Tm-InfLet or rule Tm-CheckLet, followed by rule Decl-NoAnnSingle), we learn that $\Gamma \vdash x = e_1 \Rightarrow \Gamma, x : \forall \overline{\{a\}}.\eta_1^{\epsilon}$, $\Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon}$, and either $\Gamma, x : \forall \overline{\{a\}}.\eta_1^{\epsilon} \vdash e_2 \Rightarrow \eta^{\epsilon}$ or $\Gamma, x : \forall \overline{\{a\}}.\eta_1^{\epsilon} \vdash e_2 \Leftarrow \sigma$. Both parts of the goal now follow trivially from Lemma 1 and 2 respectively. □

Property 2 (Let Extraction is Type Preserving)**.**

- *If* $\Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta_2^{\epsilon}$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon}$ *then* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Rightarrow \eta_2^{\epsilon}$
- *If* $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma_2$ *and* $\Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon}$ *then* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Leftarrow \sigma_2$

Similarly to before, we start by introducing a number of helper lemmas:

Lemma 6 (Expression Extraction is Type Preserving (Synthesis))**.**
If $\Gamma \vdash e_1 \Rightarrow \eta_1^{\epsilon}$ *and* $\Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta_2^{\epsilon}$
then $\Gamma, x : \forall \overline{\{a\}}.\eta_1^{\epsilon} \vdash e_2 \Rightarrow \eta_2^{\epsilon}$ *where* $\overline{a} = \boldsymbol{fv}\,(\eta_1^{\epsilon}) \setminus \boldsymbol{dom}\,(\Gamma)$

Lemma 7 (Expression Extraction is Type Preserving (Checking)).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Leftarrow \sigma_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma)$

Lemma 8 (Head Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash^H [e_1/x]\, h \Rightarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash^H h \Rightarrow \sigma_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma)$

Lemma 9 (Argument Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash^A [e_1/x]\, \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma)$

Lemma 10 (Declaration Extraction is Type Preserving).
If $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ *and* $\Gamma \vdash [e_1/x]\, decl \Rightarrow \Gamma, \Gamma'$
then $\Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon \vdash decl \Rightarrow \Gamma, x : \forall\, \overline{\{a\}}.\eta_1^\epsilon, \Gamma'$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma)$

In addition to these helper lemmas, we also introduce two typing context lemmas:

Lemma 11 (Environment Variable Shifting is Type Preserving).

- *If* $\Gamma_1, x_1 : \sigma_1, x_2 : \sigma_2, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma_1, x_2 : \sigma_2, x_1 : \sigma_1, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, x_1 : \sigma_1, x_2 : \sigma_2, \Gamma_2 \vdash e \Leftarrow \sigma$ *then* $\Gamma_1, x_2 : \sigma_2, x_1 : \sigma_1, \Gamma_2 \vdash e \Leftarrow \sigma$

Lemma 12 (Environment Type Variable Shifting is Type Preserving).

- *If* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *and* $\bullet = \boldsymbol{fv}(\sigma) \setminus \boldsymbol{dom}(\Gamma_1)$
  then $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Leftarrow \sigma$ *and* $\bullet = \boldsymbol{fv}(\sigma) \setminus \boldsymbol{dom}(\Gamma_1)$
  then $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Leftarrow \sigma$
- *If* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Rightarrow \eta^\epsilon$
- *If* $\Gamma_1, x : \sigma, a, \Gamma_2 \vdash e \Leftarrow \sigma$ *then* $\Gamma_1, a, x : \sigma, \Gamma_2 \vdash e \Leftarrow \sigma$

Lemmas 11 and 12 are folklore, and can be proven through straightforward induction.

Now we can go about proving Lemmas 6 till 10. Similarly to the Property 1 helper lemmas, they have to be proven using mutual induction. Most cases are quite straightforward, and we will focus only on Lemma 8. We start by performing case analysis on h:

Case $h = y$ **where** $y = x$

By evaluating the substitution, we know from the premise that $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $\Gamma \vdash^H e_1 \Rightarrow \sigma_2$, while the goal remains $\Gamma, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon \vdash^H x \Rightarrow \sigma_2$. It is clear from rule H-Var that in order for the goal to hold, $\sigma_2 = \forall\,\overline{\{a\}}.\eta_1^\epsilon$. We proceed by case analysis on the second derivation:

case rule H-Var $e_1 = x'$ **:** The rule premise tells us that $x' : \sigma_2 \in \Gamma$. The goal follows directly under lazy instantiation. However, under eager instantiation, rule Tm-InfApp instantiates the type $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \eta_1^\epsilon$ making the goal invalid.

case rule H-Con $e_1 = K$**, rule** H-Ann $e_1 = e_3 : \sigma_3$**, rule** H-Inf $e_1 = e_1$**, rule** H-Undef $e_1 = \mathit{undefined}$**, or rule** H-Seq $e_1 = e_q$ **:**

Similarly to the previous case, the goal is only valid under eager instantiation.

Case $h = y$ **where** $y \neq x$

This case is trivial, as the substitution $[e_1/x]$ does not alter h. The result thus follows from weakening.

Case $h = K$**,** $h = \mathit{undefined}$**, or** $h = e_q$

Similarly to the previous case, as the substitution does not alter h, the result thus follows from weakening.

Case $h = e : \sigma$

The result follows by applying Lemma 7.

Case $h = e$

The result follows by applying Lemma 6. □

Using these lemmas, both Property 2 goals follow straightforwardly using rule Decl-NoAnnSingle, in combination with rule Tm-InfLet and Lemma 6 or rule Tm-CheckLet and Lemma 7, respectively. □

B.2 Contextual Equivalence

As we've now arrived at properties involving the runtime semantics of the language, we first need to formalise our definition of contextual equivalence, and introduce a number of useful lemmas.

Definition 13 (Contextual Equavalence)**.**

$$
\begin{aligned}
e_1 \simeq e_2 \equiv \Gamma \vdash_{tm} e_1 : \sigma_1 \quad &\wedge \quad \Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1 \\
\wedge\, \Gamma \vdash_{tm} e_2 : \sigma_2 \quad &\wedge \quad \Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_2 \\
\wedge\, \forall M : \Gamma; \sigma_3 \mapsto \bullet; &Bool, \\
\exists v : M[\dot{t}_1[e_1]] \hookrightarrow^{\Downarrow} v \quad &\wedge \quad M[\dot{t}_2[e_2]] \hookrightarrow^{\Downarrow} v
\end{aligned}
$$

This definition for contextual equivalence is modified from [33, Chapter 46]. Two core expressions are thus contextually equivalent, if a common type exists to which both their types instantiate, and if no (closed) context can distinguish between them. This can either mean that both applied expressions evaluate to the same value v or both diverge. Note that while we require the context to map to a closed, Boolean expression, other base types, like **Int**, would have been valid alternatives as well.

We first introduce reflexivity, commutativity and transitivity lemmas:

Lemma 13 (Contextual Equivalence Reflexivity)**.**
If $\Gamma \vdash_{tm} e : \sigma$ *then* $e \simeq e$

The proof follows directly from the definition of contextual equivalence, along with the determinism of System F evaluation.

Lemma 14 (Contextual Equivalence Commutativity)**.**
If $e_1 \simeq e_2$ *then* $e_2 \simeq e_1$

Trivial proof by unfolding the definition of contextual equivalence.

Lemma 15 (Contextual Equivalence Transitivity)**.**
If $e_1 \simeq e_2$ *and* $e_2 \simeq e_3$ *then* $e_1 \simeq e_3$

Trivial proof by unfolding the definition of contextual equivalence.

Furthermore, we also introduce a number of compatibility lemmas for the contextual equivalence relation, along with two helper lemmas:

Lemma 16 (Compatibility Term Abstraction)**.**
If $e_1 \simeq e_2$ *then* $\lambda x : \sigma.e_1 \simeq \lambda x : \sigma.e_2$

Lemma 17 (Compatibility Term Application)**.**
If $e_1 \simeq e_2$ *and* $e'_1 \simeq e'_2$ *then* $e_1\, e'_1 \simeq e_2\, e'_2$

Lemma 18 (Compatibility Type Abstraction)**.**
If $e_1 \simeq e_2$ *then* $\Lambda a.e_1 \simeq \Lambda a.e_2$

Lemma 19 (Compatibility Type Application).
If $e_1 \simeq e_2$ *then* $e_1\,\sigma \simeq e_2\,\sigma$

Lemma 20 (Compatibility Case Abstraction).
If $\forall\ i : e_{1\,i} \simeq e_{2\,i}$ *then* $\mathbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \rightarrow e_{1\,i}}^{\,i} \simeq \mathbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \rightarrow e_{2\,i}}^{\,i}$

Lemma 21 (Compatibility Expression Wrapper).
If $e_1 \simeq e_2$ *then* $\dot{t}[e_1] \simeq \dot{t}[e_2]$

Lemma 22 (Compatibility Helper Forwards).
If $M[e_1] \hookrightarrow^{\Downarrow} v$ *and* $e_1 \hookrightarrow e_2$ *then* $M[e_2] \hookrightarrow^{\Downarrow} v$

Lemma 23 (Compatibility Helper Backwards).
If $M[e_2] \hookrightarrow^{\Downarrow} v$ *and* $e_1 \hookrightarrow e_2$ *then* $M[e_1] \hookrightarrow^{\Downarrow} v$

The helper lemmas are proven by straightforward induction on the evaluation step derivation. We will prove Lemma 18 as an example, as it is non-trivial. The other compatibility lemmas are proven similarly.

We start by unfolding the definition of contextual equivalence in both the premise: $\Gamma \vdash_{tm} e_1 : \sigma_1$, $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1$, $\Gamma \vdash_{tm} e_2 : \sigma_2$, $\Gamma \vdash \sigma_2 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_2$, $\forall M : \Gamma; \sigma_3 \mapsto \bullet; Bool$, $\exists v : M[\dot{t}_1[e_1]] \hookrightarrow^{\Downarrow} v$ and $M[\dot{t}_2[e_2]] \hookrightarrow^{\Downarrow} v$. Unfolding the definition reduces the goal to be proven to $\Gamma' \vdash_{tm} \Lambda a.e_1 : \sigma_1'$, $\Gamma' \vdash \sigma_1' \xrightarrow{inst\ \delta} \rho_3' \rightsquigarrow \dot{t}_1'$, $\Gamma' \vdash_{tm} \Lambda a.e_2 : \sigma_2'$, $\Gamma' \vdash \sigma_2' \xrightarrow{inst\ \delta} \rho_3' \rightsquigarrow \dot{t}_2'$, $\forall M' : \Gamma'; \sigma_3' \mapsto \bullet; Bool$, $\exists v' : M'[\dot{t}_1'[\Lambda a.e_1]] \hookrightarrow^{\Downarrow} v'$ and $M'[\dot{t}_2'[\Lambda a.e_2]] \hookrightarrow^{\Downarrow} v'$.

The typing judgement goals follow directly from rule FTm-TyAbs, where we take $\sigma_1' = \forall\, a.\sigma_1$, $\sigma_2' = \forall\, a.\sigma_2$ and $\Gamma' = [\tau/a]\,\Gamma$ for some τ.

As we know $\Gamma \vdash \sigma_1 \xrightarrow{inst\ \delta} \rho_3 \rightsquigarrow \dot{t}_1$, it is easy to see that $[\tau/a]\,\Gamma \vdash [\tau/a]\,\sigma_1 \xrightarrow{inst\ \delta} [\tau/a]\,\rho_3 \rightsquigarrow [\tau/a]\,\dot{t}_1$, and similarly for $[\tau/a]\,\sigma_2$. Using this, the instantiation goals follow from rule InstT-SForall and rule InstT-Forall with $\rho_3' = [\tau/a]\,\rho_3$, $\dot{t}_1' = \lambda e.([\sigma/a]\dot{t}_1[e\,\sigma])$ and $\dot{t}_2' = \lambda e.([\sigma/a]\dot{t}_2[e\,\sigma])$.

Finally, by inlining the definitions, the first halve of the third goal becomes $M'[(\lambda e.([\sigma/a]\dot{t}_1[e\,\sigma]))[\Lambda a.e_1]] \hookrightarrow^{\Downarrow} v'$. This reduces to $M'[[\sigma/a]\dot{t}_1[(\Lambda a.e_1)\,\sigma]] \hookrightarrow^{\Downarrow} v'$. By lemma 22 (note that we can consider the combination of a context and an expression wrapper as a new context): $M'[[\sigma/a]\dot{t}_1[[\sigma/a]e_1]] \hookrightarrow^{\Downarrow} v'$. We can now bring the substitutions to the front, and reduce the goal (by Lemma 23) $M''[\dot{t}_1[e_1]] \hookrightarrow^{\Downarrow} v'$ where we define $M'' = \lambda e.M'[(\Lambda a.e)\,\sigma]$ (note that we use λtt as meta-notation here, to simplify our definition of M''). We perform the same derivation for the second halve of the goal: $M''[\dot{t}_2[e_2]] \hookrightarrow^{\Downarrow} v'$. As $M'' : \Gamma; \sigma_3 \mapsto \bullet; Bool$, the goal follows directly from the unfolded premise, where $v' = v$. □

We introduce an additional lemma stating that instantiating the type of expressions does not alter their behaviour:

Lemma 24 (Type Instantiation is Runtime Semantics Preserving)**.**
If $\Gamma \vdash_{tm} e : \sigma$ *and* $\Gamma \vdash \sigma \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}$ *then* $e \simeq \dot{t}[e]$

The proof proceeds by induction on the instantiation relation:

Case rule InstT-SInst $\dot{t} = []$ **:**

Trivial case, as $\dot{t}[e] = e$, the goal follows directly from Lemma 13.

Case rule InstT-SForall $\dot{t} = \lambda e_1.(\dot{t}'[e_1\,\sigma])$ **:**

We know from the first premise, along with rule FTm-TyApp that $\Gamma \vdash_{tm} e\,\sigma : [\sigma/a]\sigma'$ where $\sigma = \forall\, a.\sigma'$. By applying the induction hypothesis we get $e\,\sigma \simeq \dot{t}'[e\,\sigma]$. The goal to be proven is $e \simeq (\lambda e_1.(\dot{t}'[e_1\,\sigma]))[e]$, which reduces to $e \simeq \dot{t}'[e\,\sigma]$. By unfolding the definition of contextual equivalence in both the goal and the induction hypothesis result (using Lemma 15), the remaining goals are:

- $\Gamma \vdash_{tm} e : \sigma_1$: follows directly from the first premise.
- $\Gamma \vdash \forall\, a.\sigma' \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho' \rightsquigarrow \dot{t}_1$ and $\Gamma \vdash \rho' \overset{inst\ \mathcal{S}}{\dashrightarrow} \rho \rightsquigarrow \dot{t}_2$: follows directly from the premise if we take $\rho' = \rho$, $\dot{t}_1 = \dot{t}$ and $\dot{t}_2 = []$.
- $M[\dot{t}_1[e]] \hookrightarrow^{\Downarrow} v$ and $M[\dot{t}[e]] \hookrightarrow^{\Downarrow} v$: trivial as both sides are identical and evaluation is deterministic.

Case rule InstT-SInfForall $\dot{t} = \lambda e_1.(\dot{t}'[e_1\,\sigma])$ **:**

The proof follows analogously to the previous case. We have thus proven Lemma 24 under shallow instantiation.

Case rule InstT-Mono $\dot{t} = []$ **:**

Trivial case, as $\dot{t}[e] = e$, the goal follows directly from Lemma 13.

Case rule InstT-Function $\dot{t} = \lambda e_1.\lambda x : \sigma_1.(\dot{t}'[e_1\,x])$ **:**

It is clear that the goal does not hold in this case. Under deep instantiation, full eta expansion is performed, which alters the evaluation behaviour. Consider for example *undefined* and its expansion $\lambda x : \sigma.\mathit{undefined}\,x$. □

Finally, we introduce a lemma stating that evaluation preserves contextual equivalence. However, in order to prove it, we first need to introduce the common preservation lemma:

Lemma 25 (Preservation).
If $\Gamma \vdash_{tm} e : \sigma$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash_{tm} e' : \sigma$

The preservation proof for System F is folklore, and proceeds by straightforward induction on the evaluation relation.

Lemma 26 (Evaluation is Contextual Equivalence Preserving).
If $e_1 \simeq e_2$ *and* $e_2 \hookrightarrow e_2'$ *then* $e_1 \simeq e_2'$

The proof follows by Lemma 25 (to cover type preservation) and Lemma 22 (to cover the evaluation aspect).

B.3 Let-Inlining and Extraction, Continued

Property 3 (Let Inlining is Runtime Semantics Preserving).

- *If* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow e_1$ *and* $\Gamma \vdash [e_1/x]\, e_2 \Rightarrow \eta^{\epsilon} \rightsquigarrow e_2$ *then* $e_1 \simeq e_2$
- *If* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \Leftarrow \sigma \rightsquigarrow e_1$ *and* $\Gamma \vdash [e_1/x]\, e_2 \Leftarrow \sigma \rightsquigarrow e_2$ *then* $e_1 \simeq e_2$

We first need typing preservation lemmas before we can prove Property 3.

Lemma 27 (Expression Typing Preservation (Synthesis)).
If $\Gamma \vdash e \Rightarrow \eta \rightsquigarrow e$ *then* $\Gamma \vdash_{tm} e : \sigma$

Lemma 28 (Expression Typing Preservation (Checking)).
If $\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow e$ *then* $\Gamma \vdash_{tm} e : \sigma$

Lemma 29 (Head Typing Preservation).
If $\Gamma \vdash^{H} h \Rightarrow \sigma \rightsquigarrow e$ *then* $\Gamma \vdash_{tm} e : \sigma$

Lemma 30 (Argument Typing Preservation).
If $\Gamma \vdash^{A} \overline{arg} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{arg_F}$ *then* $\forall e_i \in \overline{arg_F} : \Gamma \vdash_{tm} e_i : \sigma_i$

Lemma 31 (Declaration Typing Preservation).
If $\Gamma \vdash decl \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e$ *then* $\Gamma \vdash_{tm} e : \sigma$

Similarly to the helper lemmas for Property 1, these lemmas need to be proven using mutual induction. The proofs follow through straightforward induction on the typing derivation.

We continue by introducing another set of helper lemmas:

Lemma 32 (Expression Inlining is Runtime Semantics Preserving (Synthesis))**.**

If $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow e_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow e_3$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $e_3 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_2)\, e_1$

Lemma 33 (Expression Inlining is Runtime Semantics Preserving (Checking))**.**

If $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash e_2 \Leftarrow \sigma_2 \rightsquigarrow e_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, e_2 \Leftarrow \sigma_2 \rightsquigarrow e_3$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $e_3 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_2)\, e_1$

Lemma 34 (Head Inlining is Runtime Semantics Preserving)**.**
If $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^H h \Rightarrow \sigma \rightsquigarrow e_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$ *and* $\Gamma_1, \Gamma_2 \vdash^H [e_1/x]\, h \Rightarrow \sigma \rightsquigarrow e_3$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $e_3 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_2)\, e_1$

Lemma 35 (Argument Inlining is Runtime Semantics Preserving)**.**
If $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash^A \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2 \rightsquigarrow \overline{arg_F}_1$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$
and $\Gamma_1, \Gamma_2 \vdash^A [e_1/x]\, \overline{arg} \Leftarrow \sigma_1 \Rightarrow \sigma_2 \rightsquigarrow \overline{arg_F}_2$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$
then $\forall e_i \in \overline{arg_F}_1,\ e_i' \in \overline{arg_F}_2\ :\ e_i' \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_i)\, e_1$

Lemma 36 (Declaration Inlining is Runtime Semantics Preserving)**.**
If $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2 \vdash decl \Rightarrow \Gamma_3 \rightsquigarrow y : \sigma_2 = e_2$, $\Gamma_1 \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$ *and* $\Gamma_1, \Gamma_2 \vdash [e_1/x]\, decl \Rightarrow \Gamma_3 \rightsquigarrow y : \sigma_2 = e_3$ *where* $\overline{a} = \boldsymbol{fv}(\eta_1^\epsilon) \setminus \boldsymbol{dom}(\Gamma_1)$ *then* $e_3 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_2)\, e_1$

As is probably clear by now, these lemmas are proven through mutual induction. The proof proceeds by structural induction on the first typing derivation. We will focus on the non-trivial cases:

Case rule H-VAR $h = y$ **where** $y = x$ **:**

The goal reduces to $e_1 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.x)\, e_1$, which follows directly from Lemmas 13 and 26.

Case rule H-VAR $h = y$ **where** $y \neq x$ **:**

The goal reduces to $y \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.y)\, e_1$. Since $(\lambda x : \forall\,\overline{a}.\sigma_1.y)\, e_1 \hookrightarrow y$, the goal follows directly from Lemmas 13 and 26.

Case rule TM-INFABS $e_2 = \lambda y.e_4$ **:**

The premise tells us $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2, y : \tau_1 \vdash e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow e_4$ and $\Gamma_1, \Gamma_2, y : \tau_1 \vdash [e_1/x]\, e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow e_5$. Applying the induction hypothesis gives us $e_5 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_4)\, e_1$. The goal reduces to $\lambda y : \sigma_1.e_5 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.\lambda y : \sigma_1.e_4)\, e_1$. In order not to clutter the proof too much, we introduce an additional helper lemma 37. The goal then follows from Lemmas 16 and 37.

Case rule TM-INFTYABS $e_2 = \Lambda a.e_4$ **:**

The premise tells us $\Gamma_1, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon, \Gamma_2, a \vdash e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow e_4$, $\Gamma_1, \Gamma_2, a \vdash [e_1/x]\, e_4 \Rightarrow \eta_4^\epsilon \rightsquigarrow e_5$ and $\Gamma_1, \Gamma_2 \vdash \forall\, a.\eta_4^\epsilon \xrightarrow{inst\ \delta} \eta_5^\epsilon \rightsquigarrow \dot{t}$. Applying the induction hypothesis gives us $e_5 \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.e_4)\, e_1$. The goal reduces to $\dot{t}[\Lambda a.e_5] \simeq (\lambda x : \forall\,\overline{a}.\sigma_1.\dot{t}[\Lambda a.e_4])\, e_1$. Similarly to before, we avoid cluttering the proof by introducing an additional helper lemma 38. The goal then follows from Lemmas 18, 24 and 38. □

Lemma 37 (Property 3 Term Abstraction Helper).
If $\Gamma \vdash_{tm} \lambda x : \sigma_2.((\lambda y : \sigma_1.e_2)\, e_1) : \sigma_3$ *and* $\Gamma \vdash_{tm} e_1 : \sigma_1$ *then* $\lambda x : \sigma_2.((\lambda y : \sigma_1.e_2)\, e_1) \simeq (\lambda y : \sigma_1.\lambda x : \sigma_2.e_2)\, e_1$

Lemma 38 (Property 3 Type Abstraction Helper).
If $\Gamma \vdash_{tm} \Lambda a.((\lambda x : \sigma_1.e_2)\, e_1) : \sigma_2$ *and* $a \notin \boldsymbol{fv}(\sigma_1)$ *then* $\Lambda a.((\lambda x : \sigma_1.e_2)\, e_1) \simeq (\lambda x : \sigma_1.\Lambda a.e_2)\, e_1$

Both lemmas follow from the definition of contextual equivalence.

We now return to proving Property 3. By case analysis (Either rule Tm-InfLet or rule Tm-CheckLet, followed by rule Decl-NoAnnSingle) we know $\Gamma, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Rightarrow \eta^\epsilon \rightsquigarrow e_3$ or $\Gamma, x : \forall\,\overline{\{a\}}.\eta_1^\epsilon \vdash e_2 \Leftarrow \sigma \rightsquigarrow e_3$ where $e_1 = (\lambda x : \forall\,\overline{a}.\sigma_1.e_3)\, e_4$, $\Gamma \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_4$ and $\overline{a} = \mathbf{fv}\,(\eta_1^\epsilon) \setminus \mathbf{dom}\,(\Gamma)$. The goal thus follows directly from Lemma 32 or 33. However, as Lemma 24 only holds under shallow instantiation, we cannot prove Property 3 under deep instantiation. □

B.4 Type Signatures

Property 4b (Signature Property is Type Preserving).
If $\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma'$ *and* $x : \sigma \in \Gamma'$ *then* $\Gamma \vdash x : \sigma; x\,\overline{\pi} = e \Rightarrow \Gamma'$

Before proving Property 4b, we first introduce a number of helper lemmas:

Lemma 39 (Skolemisation Exists).
If $\boldsymbol{fv}\,(\sigma) \in \Gamma$ *then* $\exists r, \Gamma'$ *such that* $\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'$

The proof follows through careful examination of the skolemisation relation.

Lemma 40 (Skolemisation Implies Instantiation).
If $\Gamma \vdash \sigma \xrightarrow{skol\ \delta} \rho; \Gamma'$ *then* $\Gamma' \vdash \sigma \xrightarrow{inst\ \delta} \rho$

The proof follows by straightforward induction on the skolemisation relation. Note that as skolemisation binds all type variables in Γ', they can then be used for instantiation.

Lemma 41 (Inferred Type Binders Preserve Expression Checking).
If $\Gamma \vdash e \Leftarrow \sigma$ *then* $\Gamma \vdash e \Leftarrow \forall \overline{\{a\}}.\sigma$

The proof follows by straightforward induction on the typing derivation.

Lemma 42 (Pattern Synthesis Implies Checking).
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *then* $\forall \sigma', \exists \sigma : \Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta$ *where* $\mathit{type}\,(\overline{\psi}; \sigma'\ \sigma)$

The proof follows by straightforward induction on the pattern typing derivation.

Lemma 43 (Expression Synthesis Implies Checking).
If $\Gamma \vdash e \Rightarrow \eta^\epsilon$ *then* $\Gamma \vdash e \Leftarrow \eta^\epsilon$

The proof follows by induction on the typing derivation. We will focus on the non-trivial cases below:

Case rule Tm-InfAbs $e = \lambda x.e'$ **:**

We know from the premise of the typing rule that $\Gamma, x : \tau_1 \vdash e' \Rightarrow \eta_2^\epsilon$ where $\eta^\epsilon = \tau_1 \rightarrow \eta_2^\epsilon$. By rule Tm-CheckAbs, the goal reduces to $\Gamma \vdash \tau_1 \rightarrow \eta_2^\epsilon \xrightarrow{skol\ \mathcal{S}} \tau_1 \rightarrow \eta_2^\epsilon; \Gamma$ (which follows directly by rule SkolT-SInst) and $\Gamma, x : \tau_1 \vdash e' \Leftarrow \eta_2^\epsilon$ (which follows by the induction hypothesis).

Case rule Tm-InfTyAbs $e = \Lambda a.e'$ **:**

The typing rule premise tells us that $\Gamma, a \vdash e' \Rightarrow \eta_1^\epsilon$ and $\Gamma \vdash \forall a.\eta_1^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon$. By rule Tm-CheckTyAbs, the goal reduces to $\eta_2^\epsilon = \forall \overline{\{a\}}.\forall a.\sigma'$ and $\Gamma, \overline{\{a\}}, a \vdash e' \Leftarrow \sigma'$. It is now clear that this property can never hold under eager instantiation, as the forall type in $\forall a.\eta_1^\epsilon$ would always be instantiated away. We will thus focus solely on lazy instantiation from here on out, where $\eta_2^\epsilon = \forall a.\eta_1^\epsilon$. In this case, the goal follows directly from the induction hypothesis.

Case rule Tm-InfApp $e = h\,\overline{arg}$ **:**

We know from the typing rule premise that $\Gamma \vdash^H h \Rightarrow \sigma$, $\Gamma \vdash^A \overline{arg} \Leftarrow \sigma \Rightarrow \sigma'$ and $\Gamma \vdash \sigma' \xrightarrow{inst\ \delta} \eta^\epsilon$. Note that as we assume lazy instantiation, $\eta^\epsilon = \sigma'$. By rule Tm-CheckInf, the goal reduces to $\Gamma \vdash \eta^\epsilon \xrightarrow{skol\ \delta} \rho; \Gamma'$ (follows by Lemma 39), $\Gamma' \vdash h\,\overline{arg} \Rightarrow \eta_1^\epsilon$ (follows by performing environment weakening on the premise, with $\eta_1^\epsilon = \eta^\epsilon$) and $\Gamma' \vdash \eta_1^\epsilon \xrightarrow{inst\ \delta} \rho$ (given that $\eta_1^\epsilon = \eta^\epsilon$, this follows by Lemma 40). □

We now proceed with proving Property 4b, through case analysis on the declaration typing derivation (rule Decl-NoAnnSingle):

We know from the typing rule premise that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$, $\Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon$, $\mathit{type}\,(\overline{\psi}; \eta^\epsilon\ \sigma_1)$ and $\sigma = \forall \overline{\{a\}}.\sigma_1$ where $\overline{a} = \mathbf{fv}\,(\sigma_1) \setminus \mathbf{dom}\,(\Gamma)$. By rule Decl-Ann, the goal reduces to $\Gamma \vdash^P \overline{\pi} \Leftarrow \forall \overline{\{a\}}.\sigma_1 \Rightarrow \sigma_2; \Delta_2$ and $\Gamma, \Delta_2 \vdash e \Leftarrow \sigma_2$.

We know from Lemma 42 that $\Gamma \vdash^{P} \overline{\pi} \Leftarrow \sigma_1 \Rightarrow \sigma_3; \Delta$ where $\mathit{type}\,(\overline{\psi}; \sigma_3\ \sigma_1)$. Furthermore, from Lemma 43 we get $\Gamma, \Delta \vdash e \Leftarrow \eta^{\epsilon}$. Note that we thus only prove Property 4b under lazy instantiation. We now proceed by case analysis on $\overline{\pi}$:

Case $\overline{\pi} = \bullet$:

The first goal now follows trivially by rule Pat-CheckEmpty with $\sigma_2 = \forall \overline{\{a\}}.\sigma_1$, $\sigma_1 = \eta^{\epsilon}$ and $\Delta = \Delta_2 = \bullet$. The second goal follows by Lemma 41.

Case $\overline{\pi} \neq \bullet$:

The first goal follows by repeated application of rule Pat-CheckInfForall with $\sigma_2 = \sigma_3 = \eta^{\epsilon}$. The second goal then follows directly from Lemma 43. □

Property 5 (Signature Property is Runtime Semantics Preserving).
If $\Gamma \vdash \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_1$ *and* $\Gamma \vdash x : \sigma; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_2$ *then* $e_1 \simeq e_2$

We start by introducing a number of helper lemmas:

Lemma 44 (Pattern Typing Mode Preserves Translation).
If $\Gamma \vdash^{P} \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_{F1}} : \overline{\psi_{F1}}$ *and* $\Gamma \vdash^{P} \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_{F2}} : \overline{\psi_{F2}}$ *where* $\mathit{type}\,(\overline{\psi}; \sigma'\ \sigma)$
then $\overline{\pi_{F1}} = \overline{\pi_{F2}}$ *and* $\overline{\psi_{F1}} = \overline{\psi_{F2}}$

The proof follows by straightforward induction on the pattern type inference derivation.

Lemma 45 (Compatibility One-Sided Type Abstraction).
If $e_1 \simeq e_2$ *then* $e_1 \simeq \Lambda a.e_2$

The proof follows by the definition of contextual equivalence. Note that while the left and right hand sides have different types, they still instantiate to a single common type.

Lemma 46 (Partial Skolemisation Preserves Type Checking and Runtime Semantics).
If $\Gamma \vdash e \Leftarrow \forall \overline{\{a\}}.\sigma \rightsquigarrow e_1$ *then* $\Gamma, \overline{a} \vdash e \Leftarrow \sigma \rightsquigarrow e_2$ *where* $e_1 \simeq e_2$.

The proof proceeds by induction on the type checking derivation. Note that every case performs a (limited) form of skolemisation. Every case proceeds by applying the induction hypothesis, followed by Lemma 45.

Lemma 47 (Typing Mode Preserves Runtime Semantics).
If $\Gamma \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e_1$ *and* $\Gamma \vdash e \Leftarrow \sigma \rightsquigarrow e_2$ *where* $\Gamma \vdash \eta^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma \vdash$

$\sigma \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $e_1 \simeq e_2$

The proof proceeds by induction on the first typing derivation. Each case follows straightforwardly by applying the induction hypothesis, along with the corresponding compatibility lemma (Lemmas 16 till 20).

We now turn to proving property 5, through case analysis on the first declaration typing derivation:

Case rule Decl-NoAnnSingle **:**

We know from the premise of the first derivation that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F}_1 : \overline{\psi_F}_1$, $\Gamma, \Delta \vdash e \Rightarrow \eta^\epsilon \rightsquigarrow e'_1$, $\mathit{type}\,(\overline{\psi}; \eta^\epsilon\ \sigma_1)$, $e_1 = \textbf{case}\,\overline{\pi_F}_1 : \overline{\psi_F}_1 \rightarrow e'_1$ and $\sigma = \forall\,\overline{\{a\}}.\sigma_1$ where $\overline{a} = \textbf{fv}\,(\sigma_1) \backslash \textbf{dom}\,(\Gamma)$. By case analysis on the second derivation (rule Decl-Ann), we get $\Gamma \vdash^P \overline{\pi} \Leftarrow \forall\,\overline{\{a\}}.\sigma_1 \Rightarrow \sigma_2; \Delta \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}_2$, $\Gamma, \Delta \vdash e \Leftarrow \sigma_2 \rightsquigarrow e'_2$ and $e_2 = \textbf{case}\,\overline{\pi_F}_2 : \overline{\psi_F}_2 \rightarrow e'_2$.

We proceed by case analysis on the patterns $\overline{\pi}$:

case $\overline{\pi} = \bullet$ **:** We know from rule Pat-InfEmpty, rule Pat-CheckEmpty and rule Type-Empty that $\sigma_2 = \forall\,\overline{\{a\}}.\sigma_1 = \forall\,\overline{\{a\}}.\eta^\epsilon$. By applying Lemma 46, we get $\Gamma, \overline{a} \vdash e \Leftarrow \eta^\epsilon \rightsquigarrow e_3$ where $e'_2 \simeq e_3$. The goal now follows by Lemma 47 (after environment weakening, where $\sigma = \rho = \eta^\epsilon$), and Lemma 15.

case $\overline{\pi} \neq \bullet$ **:** By case analysis on the pattern checking derivation (rule Pat-CheckInfForall), we know that $\Gamma, \overline{a} \vdash^P \overline{\pi} \Leftarrow \sigma_1 \Rightarrow \sigma_2; \Delta' \rightsquigarrow \overline{\pi_F}_2 : \overline{\psi_F}'_2$ where $\Delta = \overline{a}, \Delta'$ and $\overline{\psi_F}_2 = @\overline{a}, \overline{\psi_F}'_2$. By Lemma 42 (where we take $\sigma = \sigma_1$), we know that $\mathit{type}\,(\overline{\psi}; \sigma_2\ \sigma_1)$. This thus means that $\sigma_2 = \eta^\epsilon$. By Lemma 44, the goal reduces to $\textbf{case}\,\overline{\pi_F}_1 : \overline{\psi_F}_1 \rightarrow e'_1 \simeq \textbf{case}\,\overline{\pi_F}_1 : \overline{\psi_F}_1 \rightarrow e'_2$. Applying Lemma 20 reduces this goal further to $e'_1 \simeq e'_2$. This follows directly from Lemma 47 (where $\sigma = \rho = \eta^\epsilon$).

Case rule Decl-NoAnnMulti **:**

We know from the premise of the first derivation that $\forall i : \Gamma \vdash^P \overline{\pi}_i \Rightarrow \overline{\psi}; \Delta_i \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}$, $\Gamma, \Delta_i \vdash e_i \Rightarrow \eta^\epsilon_i \rightsquigarrow e_i$ and $\Gamma, \Delta_i \vdash \eta^\epsilon_i \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_i$. Furthermore, $e_1 = \textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \rightarrow \dot{t}_i[e_i]}^{\,i}$, $\mathit{type}\,(\overline{\psi}; \rho'\ \sigma')$ and $\sigma = \forall\,\overline{\{a\}}.\sigma'$ where $\overline{a} = \textbf{fv}\,(\sigma') \backslash$ $\textbf{dom}\,(\Gamma)$. By case analysis on the second derivation (rule Decl-Ann), we know that $\forall i : \Gamma \vdash^P \overline{\pi}_i \Leftarrow \forall\,\overline{\{a\}}.\sigma' \Rightarrow \sigma_i; \Delta_i \rightsquigarrow \overline{\pi_F}'_i : \overline{\psi_F}'$, $\Gamma, \Delta_i \vdash e_i \Leftarrow \sigma_i \rightsquigarrow e'_i$ and $e_2 = \textbf{case}\,\overline{\overline{\pi_F}'_i : \overline{\psi_F}' \rightarrow e'_i}^{\,i}$.

We again perform case analysis on the patterns $\overline{\pi}$:

case $\overline{\pi} = \bullet$ **:** Similarly to last time, we know that $\sigma' = \rho'$ and $\forall i : \sigma_i =$

$\forall\,\overline{\{a\}}.\rho'$. We know by Lemma 46 that $\forall i : \Gamma, \overline{a} \vdash e_i \Leftarrow \rho' \rightsquigarrow e''_i$ where $e'_i \simeq e''_i$. The goal now follows by Lemma 47 (where we take $\sigma = \rho = \rho'$) and Lemma 15.

case $\overline{\pi} \neq \bullet$ **:** Similarly to the previous case, we can derive that $\forall i : \Gamma, \overline{a} \vdash^{P} \overline{\pi} \Leftarrow \sigma' \Rightarrow \sigma_i; \Delta'_i \rightsquigarrow \overline{\pi_F}'_i : \overline{\psi_F}''$ where $\Delta_i = \overline{a}, \Delta'_i$ and $\overline{\psi_F}' = @\overline{a}, \overline{\psi_F}''$. We again derive by Lemma 42 that $\mathit{type}\,(\overline{\psi}; \sigma_i\ \sigma')$ and thus that $\sigma_i = \rho'$. By Lemma 44, the goal reduces to $\textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \to \dot{t}_i[e_i]}^{\,i} \simeq \textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F} \to e'_i}^{\,i}$. We reduce this goal further by applying Lemma 20 to $\forall i : \dot{t}_i[e_i] \simeq e'_i$. This follows directly from Lemma 47 (where $\sigma = \rho = \rho'$).

Note however, that as Lemma 47 only holds under shallow instantiation, that the same holds true for Property 5. □

Property 6 (Type Signatures are Runtime Semantics Preserving)**.**
If $\Gamma \vdash x : \sigma_1; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_1 = e_1$ *and* $\Gamma \vdash x : \sigma_2; \overline{x\,\overline{\pi}_i = e_i}^{\,i} \Rightarrow \Gamma_1 \rightsquigarrow x : \sigma_2 = e_2$ *where* $\Gamma \vdash \sigma_1 \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma \vdash \sigma_2 \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_2$ *then* $\dot{t}_1[e_1] \simeq \dot{t}_2[e_2]$

We start by introducing a number of helper lemmas:

Lemma 48 (Substitution in Expressions is Type Preserving (Synthesis))**.**
If $\Gamma, a \vdash e \Rightarrow \eta^{\epsilon} \rightsquigarrow e$ *then* $\Gamma \vdash [\tau/a]\,e \Rightarrow [\tau/a]\,\eta^{\epsilon} \rightsquigarrow [\tau/a]\,e$

Lemma 49 (Substitution in Expressions is Type Preserving (Checking))**.**
If $\Gamma, a \vdash e \Leftarrow \sigma \rightsquigarrow e$ *then* $\Gamma \vdash [\tau/a]\,e \Leftarrow [\tau/a]\,\sigma \rightsquigarrow [\tau/a]\,e$

Lemma 50 (Substitution in Heads is Type Preserving)**.**
If $\Gamma, a \vdash^{H} h \Rightarrow \sigma \rightsquigarrow e$ *then* $\Gamma \vdash^{H} [\tau/a]\,h \Rightarrow [\tau/a]\,\sigma \rightsquigarrow [\tau/a]\,e$

Lemma 51 (Substitution in Arguments is Type Preserving)**.**
If $\Gamma, a \vdash^{A} \overline{\mathit{arg}} \Leftarrow \sigma \Rightarrow \sigma' \rightsquigarrow \overline{\mathit{arg}_F}$ *then* $\Gamma \vdash^{A} [\tau/a]\,\overline{\mathit{arg}} \Leftarrow [\tau/a]\,\sigma \Rightarrow [\tau/a]\,\sigma' \rightsquigarrow [\tau/a]\,\overline{\mathit{arg}_F}$

Lemma 52 (Substitution in Declarations is Type Preserving)**.**
If $\Gamma, a \vdash \mathit{decl} \Rightarrow \Gamma, a, x : \sigma \rightsquigarrow x : \sigma = e$ *then* $\Gamma \vdash [\tau/a]\,\mathit{decl} \Rightarrow \Gamma, x : [\tau/a]\,\sigma \rightsquigarrow x : \sigma = [\tau/a]\,e$

The proof proceeds by mutual induction on the typing derivation. While the number of cases gets pretty large, each is quite straightforward.

Lemma 53 (Type Instantiation Produces Equivalent Expressions (Synthesis))**.**

If $\Gamma_1 \vdash e \Rightarrow \eta_1^{\epsilon} \rightsquigarrow e_1,\ \Gamma_2 \vdash e \Rightarrow \eta_2^{\epsilon} \rightsquigarrow e_2$ *and* $\exists\ \overline{a} \subseteq \boldsymbol{fv}\,(\eta_1^{\epsilon}) \cup \boldsymbol{fv}\,(\eta_2^{\epsilon})$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\,\overline{a}.\eta_1^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\,\overline{a}.\eta_2^{\epsilon} \xrightarrow{\mathit{inst}\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda\overline{a}.e_1] \simeq \dot{t}_2[\Lambda\overline{a}.e_2]$

Lemma 54 (Type Instantiation Produces Equivalent Expressions (Checking))**.**

If $\Gamma_1 \vdash e \Leftarrow \sigma_1 \rightsquigarrow e_1$ *and* $\Gamma_2 \vdash e \Leftarrow \sigma_2 \rightsquigarrow e_2$ *and* $\exists\ \overline{a} \subseteq \boldsymbol{fv}(\sigma_1) \cup \boldsymbol{fv}(\sigma_2)$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\,\overline{a}.\sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\,\overline{a}.\sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda\overline{a}.e_1] \simeq \dot{t}_2[\Lambda\overline{a}.e_2]$

Lemma 55 (Type Instantiation Produces Equivalent Expressions (Head Judgement))**.**
If $\Gamma_1 \vdash^H h \Rightarrow \sigma_1 \rightsquigarrow e_1,\ \Gamma_2 \vdash^H h \Rightarrow \sigma_2 \rightsquigarrow e_2$ *and* $\exists\ \overline{a} \subseteq \boldsymbol{fv}(\eta_1^\epsilon) \cup \boldsymbol{fv}(\eta_2^\epsilon)$
such that $\Gamma' = [\overline{\tau}/\overline{a}]\,\Gamma_1 = [\overline{\tau}/\overline{a}]\,\Gamma_2$ *and* $\Gamma' \vdash \forall\,\overline{a}.\sigma_1 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1$ *and* $\Gamma' \vdash \forall\,\overline{a}.\sigma_2 \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2$
then $\dot{t}_1[\Lambda\overline{a}.e_1] \simeq \dot{t}_2[\Lambda\overline{a}.e_2]$

Note that we define $[\tau/a]\,\Gamma$ as removing a from the environment Γ and substituting any occurrence of a in types bound to term variables. Furthermore, we use $\overline{a}_1 \cup \overline{a}_2$ as a shorthand for list concatenation, removing duplicates. The proof proceeds by induction on the first typing derivation. Note that Lemmas 53, 54 and 55 have to be proven using mutual induction. However, the proof for Lemma 55 is trivial, as every case besides rule H-Inf is deterministic. As usual, we will focus on the non-trivial cases:

Case rule Tm-CheckAbs $e = \lambda x.e'$ **:**

We know from the premise of the first and second (as the relation is syntax directed) typing derivation that $\Gamma_1 \vdash \sigma_1 \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_4 \to \sigma_5; \Gamma_1' \rightsquigarrow \dot{t}_1',\ \Gamma_2 \vdash \sigma_2 \overset{skol\ \mathcal{S}}{\dashrightarrow} \sigma_4' \to \sigma_5'; \Gamma_2' \rightsquigarrow \dot{t}_2',\ \Gamma_1', x : \sigma_4 \vdash e' \Leftarrow \sigma_5 \rightsquigarrow e_3$ and $\Gamma_2', x : \sigma_4' \vdash e' \Leftarrow \sigma_5' \rightsquigarrow e_4$, where $e_1 = \dot{t}_1'[\lambda x : \sigma_4.e_3]$ and $e_2 = \dot{t}_2'[\lambda x : \sigma_4'.e_4]$.

At this point, it is already clear that Lemma 54 can not hold under deep instantiation, as instantiation performs full eta expansion. We will thus focus on shallow instantiation from here on out.

By case analysis on the skolemisation and instantiation premises, it is clear that $\Gamma_1' = \Gamma_1, \overline{a}_1$, $\Gamma_2' = \Gamma_2, \overline{a}_2$ and $\rho = [\overline{\tau}_1/\overline{a}_1]\,(\sigma_4 \to \sigma_5) = [\overline{\tau}_2/\overline{a}_2]\,(\sigma_4' \to \sigma_5') = \sigma_3 \to \sigma_3'$. In order to apply the induction hypothesis, we take $\overline{a}'$ as $\overline{a} \cup \overline{a}_1 \cup \overline{a}_2$. Note that this does not alter the instantiation to ρ in any way, as these variables would already have been instantiated. We apply the induction hypothesis with $\Gamma_1 \vdash \forall\,\overline{a}'.\sigma_5 \xrightarrow{inst\ \delta} \sigma_3' \rightsquigarrow \dot{t}_3$ and $\Gamma_2 \vdash \forall\,\overline{a}'.\sigma_5' \xrightarrow{inst\ \delta} \sigma_3' \rightsquigarrow \dot{t}_4$ (after weakening), producing $\dot{t}_3[\Lambda\overline{a}'.e_3] \simeq \dot{t}_4[\Lambda\overline{a}'.e_4]$. Under shallow instantiation, these two instantiations follow directly from the premise with $\dot{t}_3 = \dot{t}_1$ and $\dot{t}_4 = \dot{t}_2$.

The goal reduces to $\dot{t}_1[\Lambda\overline{a}.\dot{t}_1'[\lambda x : \sigma_4.e_3]] \simeq \dot{t}_2[\Lambda\overline{a}.\dot{t}_2'[\lambda x : \sigma_4'.e_4]]$. By the definition of skolemisation, this further reduces to $\dot{t}_1[\Lambda\overline{a}.\Lambda\overline{a}_1.\lambda x : \sigma_4.e_3] \simeq \dot{t}_2[\Lambda\overline{a}.\Lambda\overline{a}_2.\lambda x :$

$\sigma_4'.e_4]$. Finally, the goal follows by the induction hypothesis and compatibility Lemmas 18, 16 and 21, along with transitivity Lemma 15.

Case rule TM-CHECKTYABS $e = \Lambda a.e'$ **:**

We know the premise of the typing derivation that $\sigma_1 = \forall\,\overline{\{a\}}_1.\forall\, a.\sigma_1'$, $\sigma_2 = \forall\,\overline{\{a\}}.\forall\, a.\sigma_2'$, $\Gamma_1, \overline{a}_1, a \vdash e' \Leftarrow \sigma_1' \rightsquigarrow e_1'$, $\Gamma_2, \overline{a}_2, a \vdash e' \Leftarrow \sigma_2' \rightsquigarrow e_2'$, $e_1 = \Lambda\overline{a}_1.\Lambda a.e_1'$ and $e_2 = \Lambda\overline{a}_2.\Lambda a.e_2'$. By case analysis on the type instantiation (rule INSTT-SFORALL and rule INSTT-SINFFORALL), we get $\Gamma' \vdash [\overline{\tau}_1/\overline{a}]\,[\overline{\tau}_1'/\overline{a}_1]\,[\tau_1/a]\,\sigma_1' \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_1'$ and $\Gamma' \vdash [\overline{\tau}_2/\overline{a}]\,[\overline{\tau}_2'/\overline{a}_2]\,[\tau_2/a]\,\sigma_2' \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_2'$ where $\dot{t}_1 = \lambda e.(\dot{t}_1'[e\,\overline{\sigma}_1\,\overline{\sigma}_1'\,\sigma_1])$ and $\dot{t}_2 = \lambda e.(\dot{t}_2'[e\,\overline{\sigma}_2\,\overline{\sigma}_2'\,\sigma_2])$.

The goal to be proven is $\dot{t}_1[\Lambda\overline{a}.\Lambda\overline{a}_1.\Lambda a.e_1'] \simeq \dot{t}_2[\Lambda\overline{a}.\Lambda\overline{a}_2.\Lambda a.e_2']$. This reduces to $\dot{t}_1'[(\Lambda\overline{a}.\Lambda\overline{a}_1.\Lambda a.e_1')\,\overline{\sigma}_1\,\overline{\sigma}_1'\,\sigma_1] \simeq \dot{t}_2'[(\Lambda\overline{a}.\Lambda\overline{a}_2.\Lambda a.e_2')\,\overline{\sigma}_2\,\overline{\sigma}_2'\,\sigma_2]$.

We now define a substitution $\theta = [\overline{\tau}_1/\overline{a}].[\overline{\tau}_2/\overline{a}].[\overline{\tau}_1'/\overline{a}_1].[\overline{\tau}_2'/\overline{a}_2].[\tau_1/a].[\tau_2/a]$. From the instantiation relation (and the fact that both types instantiate to the same type r, we conclude that if $[\tau_i/a] \in \theta$ and $[\tau_j/a] \in \theta$ that $\tau_i = \tau_j$. By applying Lemma 49, we transform the premise to $[\overline{\tau}_1/\overline{a}]\,\Gamma_1 \vdash \theta\, e' \Leftarrow \theta\,\sigma_1' \rightsquigarrow \theta\, e_1'$ and $[\overline{\tau}_2/\overline{a}]\,\Gamma_2 \vdash \theta\, e' \Leftarrow \theta\,\sigma_2' \rightsquigarrow \theta\, e_2'$.

By applying the induction hypothesis, we get that $\dot{t}_1'[\theta\, e_1'] \simeq \dot{t}_2'[\theta\, e_2']$. The goal follows directly from the definition of θ.

Case rule TM-CHECKINF **:**

We know from the premise of the typing derivation that $\Gamma_1 \vdash \sigma_1 \xrightarrow{skol\ \delta} \rho_1; \Gamma_1' \rightsquigarrow \dot{t}_1'$, $\Gamma_2 \vdash \sigma_2 \xrightarrow{skol\ \delta} \rho_2; \Gamma_2' \rightsquigarrow \dot{t}_2'$, $\Gamma_1' \vdash e \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1'$, $\Gamma_2' \vdash e \Rightarrow \eta_2^\epsilon \rightsquigarrow e_2'$, $\Gamma_1' \vdash \eta_1^\epsilon \xrightarrow{inst\ \delta} \rho_1 \rightsquigarrow \dot{t}_1''$, $\Gamma_2' \vdash \eta_2^\epsilon \xrightarrow{inst\ \delta} \rho_2 \rightsquigarrow \dot{t}_2''$, $e_1 = \dot{t}_1'[\dot{t}_1''[e_1']]$ and $e_2 = \dot{t}_2''[\dot{t}_2'[e_2']]$. The goal to be proven is thus $\dot{t}_1[\Lambda\overline{a}.\dot{t}_1'[\dot{t}_1''[e_1']]] \simeq \dot{t}_2[\Lambda\overline{a}.\dot{t}_2'[\dot{t}_2''[e_2']]]$.

From the definition of shallow skolemisation, we know that $\Gamma_1' = \Gamma_1, \overline{a}_1$, $\Gamma_2' = \Gamma_2, \overline{a}_2$, $\dot{t}_1' = \lambda e.\Lambda\overline{a}_1.e$ and $\dot{t}_2' = \lambda e.\Lambda\overline{a}_2.e$. We now take $\overline{a}' = \overline{a} \cup \overline{a}_1 \cup \overline{a}_2$. As σ_1 and σ_2 instantiate to the same type ρ, it is not hard to see from the definition of skolemisation that $\Gamma_1' \vdash \forall\,\overline{a}'.\eta_1^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_3$ and $\Gamma_2' \vdash \forall\,\overline{a}'.\eta_2^\epsilon \xrightarrow{inst\ \delta} \rho \rightsquigarrow \dot{t}_4$. By applying Lemma 53, we thus get $\dot{t}_3[\Lambda\overline{a}'.e_1'] \simeq \dot{t}_4[\Lambda\overline{a}'.e_2']$. The goal follows through careful examination of the skolemisation and instantiation premises. □

Lemma 56 (Pattern Checking Implies Synthesis).
If $\Gamma \vdash^P \overline{\pi} \Leftarrow \sigma \Rightarrow \sigma'; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *then* $\exists\overline{\psi} : \Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *where* $type\,(\overline{\psi}; \sigma'\ \sigma)$

The proof follows by straightforward induction on the pattern typing derivation.

We now go back to proving Property 6, and proceed by case analysis on both typing derivations (rule DECL-ANN). We know from the premise that

$\Gamma \vdash^P \overline{\pi}_i \Leftarrow \sigma_1 \Rightarrow \sigma_{i\,1}; \Delta_{i\,1} \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}_1$, $\Gamma \vdash^P \overline{\pi}_i \Leftarrow \sigma_2 \Rightarrow \sigma_{i\,2}; \Delta_{i\,2} \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}_2$, $\Gamma, \Delta_{i\,1} \vdash e_i \Leftarrow \sigma_{i\,1} \rightsquigarrow e_{i\,1}$, $\Gamma, \Delta_{i\,2} \vdash e_i \Leftarrow \sigma_{i\,2} \rightsquigarrow e_{i\,2}$, $e_1 = \textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F}_1 \to e_{i\,1}}^{\,i}$ and $e_2 = \textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F}_2 \to e_{i\,2}}^{\,i}$. The goal to be proven is $\dot{t}_1[\textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F}_1 \to e_{i\,1}}^{\,i}] \simeq \dot{t}_2[\textbf{case}\,\overline{\overline{\pi_F}_i : \overline{\psi_F}_2 \to e_{i\,2}}^{\,i}]$. Lemma 20 reduces this to $\forall i : \dot{t}_1[e_{i\,1}] \simeq \dot{t}_2[e_{i\,2}]$.

We take $\overline{a}_i = \textbf{dom}\,(\Delta_{i\,1}) \cup \textbf{dom}\,(\Delta_{i\,2}) \setminus \textbf{dom}\,(\Gamma)$, and apply weakening to get $\Gamma, \overline{a}_i \vdash e_i \Leftarrow \sigma_{i\,1} \rightsquigarrow e_{i\,1}$ and $\Gamma, \overline{a}_i \vdash e_i \Leftarrow \sigma_{i\,2} \rightsquigarrow e_{i\,2}$. The goal now follows directly from Lemma 54 with $\overline{a}_i = \bullet$, if we can show that $\Gamma, \overline{a}_i \vdash \sigma_{i\,1} \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_1$ and $\Gamma, \overline{a}_i \vdash \sigma_{i\,2} \xrightarrow{inst\ \delta} \rho' \rightsquigarrow \dot{t}_2$ for some r' (Note that Lemma 54 only holds under shallow instantiation).

We know from Lemma 56 that $\exists \overline{\psi_F} : \Gamma \vdash^P \overline{\pi}_i \Rightarrow \overline{\psi}; \Delta_i \rightsquigarrow \overline{\pi_F}_i : \overline{\psi_F}$ such that $type\,(\overline{\psi}; \sigma_{i\,1}\ \sigma_1)$ and $type\,(\overline{\psi}; \sigma_{i\,2}\ \sigma_2)$. The remaining goal follows from the definition of the type relation, and shallow instantiation. □

B.5 Pattern Inlining and Extraction

Property 7 (Pattern Inlining is Type Preserving)**.**
If $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$ *and* $wrap\,(\overline{\pi}; e_1\ \ e_2)$ *then* $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$

We first introduce a helper lemma to prove pattern inlining in expressions preserves the type:

Lemma 57 (Pattern Inlining in Expressions is Type Preserving (Synthesis))**.**
If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$ *where* $wrap\,(\overline{\pi}; e_1\ \ e_2)$
then $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ *and* $type\,(\overline{\psi}; \eta_1^\epsilon\ \ \eta_2^\epsilon)$

The proof proceeds by induction on the pattern typing derivation. We will focus on the non-trivial cases below. Note that the rule Pat-InfCon is an impossible case as $wrap\,(K\,\overline{\pi}; e_1\ \ e_2)$ is undefined.

Case rule Pat-InfVar $\overline{\pi} = x, \overline{\pi}'$**,** $\overline{\psi} = \tau_1, \overline{\psi}'$ **and** $\Delta = x : \tau_1, \Delta'$ **:**

We know from the rule premise that $\Gamma, x : \tau_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$. Furthermore, by inlining the definitions of Δ and $\overline{\pi}$ in the lemma premise, we get $\Gamma, x : \tau_1, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $wrap\,(x, \overline{\pi}'; e_1\ \ \lambda x.e_2')$ and thus (by rule PatWrap-Var) $wrap\,(\overline{\pi}'; e_1\ \ e_2')$. By the induction hypothesis, we get $\Gamma, x : \tau_1 \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon\ \ \eta_3^\epsilon)$. The goal follows by rule Tm-InfAbs and rule Type-Var.

Case rule Pat-InfTyVar $\overline{\pi} = @a, \overline{\pi}'$**,** $\overline{\psi} = @a, \overline{\psi}'$ **and** $\Delta = a, \Delta'$ **:**

We know from the rule premise that $\Gamma, a \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$. Again, by inlining the definitions in the lemma premise, we get $\Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $wrap\,(@a, \overline{\pi}'; e_1\ \Lambda a.e_2')$ and thus (by rule PatWrap-TyVar) $wrap\,(\overline{\pi}'; e_1\ e_2')$. By the induction hypothesis, we get $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon\ \eta_3^\epsilon)$.

The goal to be proven is $\Gamma \vdash \Lambda a.e_2' \Rightarrow \forall\, a.\eta_3^\epsilon$ where $type\,(@a, \overline{\psi}'; \eta_1^\epsilon\ \forall\, a.\eta_3^\epsilon)$ (follows by rule Type-TyVar). However, under eager instantiation, this goal can never hold as rule Tm-InfTyAbs would instantiate the forall binder away. We can thus only prove this lemma under lazy instantiation, where the goal follows trivially from rule Tm-InfTyAbs. □

We now proceed with proving Property 7, through case analysis on the declaration typing relation (rule Decl-NoAnnSingle). We know from the premise of the first derivation that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta, \Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$, $type\,(\overline{\psi}; \eta_1^\epsilon\ \sigma)$ and $\Gamma' = \Gamma, x : \forall \overline{\{a\}}.\sigma$ where $\overline{a} = \mathbf{fv}\,(\sigma) \setminus \mathbf{dom}\,(\Gamma)$. The goal to be proven thus becomes $\Gamma \vdash^P \bullet \Rightarrow \bullet; \bullet$ (follows directly from rule Pat-InfEmpty) and $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ where $\eta_2^\epsilon = \sigma$ (follows from Lemma 57). Note that as we require Lemma 57, we can only prove Property 7 under lazy instantiation. □

Property 9 (Pattern Extraction is Type Preserving).
If $\Gamma \vdash x = e_2 \Rightarrow \Gamma'$ *and* $wrap\,(\overline{\pi}; e_1\ e_2)$ *then* $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma'$

We first introduce another helper lemma to prove that pattern extraction from expressions preserves the typing:

Lemma 58 (Pattern Extraction from Expressions is Type Preserving (Synthesis)).
If $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ *and* $\exists\ e_1, \overline{\pi}$ *such that* $wrap\,(\overline{\pi}; e_1\ e_2)$
then $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon$ *where* $type\,(\overline{\psi}; \eta_1^\epsilon\ \eta_2^\epsilon)$

The proof proceeds by induction on the e_2 typing derivation. As usual, we will focus on the non-trivial cases:

Case rule Tm-InfAbs $e_2 = \lambda x.e_2'$ **and** $\eta_2^\epsilon = \tau_2 \rightarrow \eta_3^\epsilon$ **:**

We know from the rule premise that $\Gamma, x : \tau_2 \vdash e_2' \Rightarrow \eta_3^\epsilon$. It is clear by case analysis on $wrap\,(\overline{\pi}; e_1\ \lambda x.e_2')$ that $\overline{\pi} = x, \overline{\pi}'$ and $wrap\,(\overline{\pi}'; e_1\ e_2')$. By applying the induction hypothesis, we get $\Gamma, x : \tau_2 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$, $\Gamma, x : \tau_2, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $type\,(\overline{\psi}'; \eta_1^\epsilon\ \eta_3^\epsilon)$. The goal thus follows straightforwardly by rule Pat-InfVar and rule Type-Var.

Case rule Tm-InfTyAbs $e_2 = \Lambda a.e_2'$ **:**

We know from the rule premise that $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon$ and $\Gamma \vdash \forall\, a.\eta_3^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon$. Furthermore, it is clear by case analysis on $wrap\,(\overline{\pi}; e_1\ \Lambda a.e_2')$ that $\overline{\pi} = @a, \overline{\pi}'$

and $\mathit{wrap}\,(\overline{\pi}'; e_1\ e_2')$. By the induction hypothesis, we get $\Gamma, a \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta'$, $\Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ and $\mathit{type}\,(\overline{\psi}'; \eta_1^\epsilon\ \eta_3^\epsilon)$.

The goal to be proven is $\Gamma \vdash^P @a, \overline{\pi}' \Rightarrow @a, \overline{\psi}'; a, \Delta'$ (follows by rule Pat-InfTyVar), $\Gamma, a, \Delta' \vdash e_1 \Rightarrow \eta_1^\epsilon$ (follows by the induction hypothesis) and $\mathit{type}\,(@a, \overline{\psi}'; \eta_1^\epsilon\ \eta_2^\epsilon)$. However, it is clear that this final goal does not hold under eager instantiation, as rule Tm-InfTyAbs instantiates the forall binder away. Under lazy instantiation, the remaining goal follows directly from the premise.

Case rule Tm-InfApp $e_2 = h\,\overline{arg}$ **and** $\overline{arg} = \bullet$ **and** $h = e$ **:**

The goal follows directly by the induction hypothesis.

Case rule Tm-InfApp $e_2 = h\,\overline{arg}$ **and** $\overline{arg} \neq \bullet$ **or** $h \neq e$ **:**

It is clear from the definition of $\mathit{wrap}\,(\overline{\pi}; e_1\ h\,\overline{arg})$ that $\overline{\pi} = \bullet$. The goal thus follows trivially. □

We now return to prove Property 9 by case analysis on the declaration typing derivation (rule Decl-NoAnnSingle). We know from the derivation premise that $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon$ and $\sigma = \forall \overline{\{a\}}.\eta_2^\epsilon$ where $\overline{a} = \mathbf{fv}\,(\eta_2^\epsilon) \setminus \mathbf{dom}\,(\Gamma)$. The goal follows directly from Lemma 58. Note that as Lemma 58 only holds under lazy instantiation, the same holds true for Property 9. □

Property 8 (Pattern Inlining / Extraction is Runtime Semantics Preserving)**.** *If* $\Gamma \vdash x\,\overline{\pi} = e_1 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_1$, $\mathit{wrap}\,(\overline{\pi}; e_1\ e_2)$, *and* $\Gamma \vdash x = e_2 \Rightarrow \Gamma' \rightsquigarrow x : \sigma = e_2$ *then* $e_1 \simeq e_2$

We start by introducing a helper lemma, proving pattern inlining preserves the runtime semantics for expressions.

Lemma 59 (Pattern Inlining in Expressions is Runtime Semantics Preserving)**.**

If $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$ *and* $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1$ *and* $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow e_2$ *where* $\mathit{wrap}\,(\overline{\pi}; e_1\ e_2)$
then $\mathbf{case}\,\overline{\pi_F} : \overline{\psi_F} \rightarrow e_1 \simeq e_2$

The proof proceeds by induction on the pattern typing derivation. We will focus on the non-trivial cases. Note that, as $\mathit{wrap}\,(K\,\overline{\pi}; e_1\ e_2)$ is undefined, rule Pat-InfCon is an impossible case.

Case rule Pat-InfVar $\overline{\pi} = x, \overline{\pi}'$**,** $\overline{\psi} = \tau_1, \overline{\psi}'$ **,** $\Delta = x : \tau_1, \Delta'$**,** $\overline{\pi_F} = x : \sigma_1, \overline{\pi_F}'$ **and** $\overline{\psi_F} = \sigma_1, \overline{\psi_F}'$ **:**

We know from the pattern typing derivation premise that $\Gamma, x : \tau_1 \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta' \rightsquigarrow \overline{\pi_F}' : \overline{\psi_F}'$. By inlining the definitions and rule PatWrap-Var, we get

$e_2 = \lambda x.e_2'$ and $\textit{wrap}\,(\overline{\pi}'; e_1\ e_2')$. By case analysis on the e_2 typing derivation (rule Tm-InfAbs), we know $\Gamma, x : \tau_1 \vdash e_2' \Rightarrow \eta_3^\epsilon \rightsquigarrow e_2'$ where $\eta_2^\epsilon = \tau_1 \rightarrow \eta_3^\epsilon$ and $e_2 = \lambda x : \sigma_1.e_2'$. By applying the induction hypothesis, we get $\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1 \simeq e_2'$. The goal to be proven is $\lambda x : \sigma_1.\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1 = \lambda x : \sigma_1.e_2'$, and follows directly from Lemma 16.

Case rule Pat-InfTyVar $\overline{\pi} = @a, \overline{\pi}'$**,** $\overline{\psi} = @a, \overline{\psi}'$ **,** $\Delta = a, \Delta'$**,** $\overline{\pi_F} = @a, \overline{\pi_F}'$ **and** $\overline{\psi_F} = @a, \overline{\psi_F}'$ **:**

We know from the pattern typing derivation premise that $\Gamma, a \vdash^P \overline{\pi}' \Rightarrow \overline{\psi}'; \Delta' \rightsquigarrow \overline{\pi_F}' : \overline{\psi_F}'$. Similarly to the previous case, by inlining and rule PatWrap-TyVar, we get $e_2 = \Lambda a.e_2'$ and $\textit{wrap}\,(\overline{\pi}'; e_1\ e_2')$. By case analysis on the e_2 typing derivation (rule Tm-InfTyAbs), we get $\Gamma, a \vdash e_2' \Rightarrow \eta_3^\epsilon \rightsquigarrow e_2'$, $\Gamma \vdash \forall\, a.\eta_3^\epsilon \xrightarrow{inst\ \delta} \eta_2^\epsilon \rightsquigarrow \dot{t}$ and $e_2 = \dot{t}[\Lambda a.e_2']$. Applying the induction hypothesis tells us that $\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1 \simeq e_2'$.

The goal to be proven is $\Lambda a.\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1 \simeq \dot{t}[\Lambda a.e_2']$. By applying Lemma 18 to the result of the induction hypothesis, we get $\Lambda a.\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1 \simeq \Lambda a.e_2'$. Under lazy instantiation, the goal follows directly from this result, as $\dot{t} = []$. Under eager deep instantiation, it is clear that the goal does not hold, as $\dot{t}$ might perform eta expansion, thus altering the runtime semantics. Under eager shallow instantiation, the goal follows straightforwardly, as $\dot{t}$ can only perform type applications. Note that this implies that $\Lambda a.\textbf{case}\,\overline{\pi_F}' : \overline{\psi_F}' \rightarrow e_1$ and $\dot{t}[\Lambda a.e_2']$ could thus have different types, but can always instantiate to the same type. □

We now return to proving Property 8, by case analysis on the first declaration typing relation (rule Decl-NoAnnSingle). We know from the derivation premise that $\Gamma \vdash^P \overline{\pi} \Rightarrow \overline{\psi}; \Delta \rightsquigarrow \overline{\pi_F} : \overline{\psi_F}$, $\Gamma, \Delta \vdash e_1 \Rightarrow \eta_1^\epsilon \rightsquigarrow e_1'$, $e_1 = \textbf{case}\,\overline{\pi_F} : \overline{\psi_F} \rightarrow e_1'$, $\textit{type}\,(\overline{\psi}; \eta_1^\epsilon\ \sigma')$, $\sigma = \forall\,\overline{\{a\}}.\sigma'$ where $\overline{a} = \textbf{fv}\,(\sigma') \backslash \textbf{dom}\,(\Gamma)$. The premise of the second declaration typing derivations tells us that $\Gamma \vdash e_2 \Rightarrow \eta_2^\epsilon \rightsquigarrow e_2$. The goal now follows directly from Lemma 59. Note that as Lemma 59 does not hold under eager deep instantiation, the same is true for Property 8. □

B.6 Single vs. Multiple Equations

Property 10 (Single/multiple Equations is Type Preserving)**.**
If $\Gamma \vdash x\,\overline{\pi} = e \Rightarrow \Gamma, x : \sigma$ *then* $\Gamma \vdash x\,\overline{\pi} = e, x\,\overline{\pi} = e \Rightarrow \Gamma'$

The proof proceeds by case analysis on the declaration typing derivation (rule Decl-NoAnnSingle). From the derivation premise, we get $\Gamma \vdash^P$

$\boxed{\mathit{numargs}\,(\sigma) = m}$ *(Explicit Argument Counting)*

$$\textsc{Numargs-TyVar}\ \frac{}{\mathit{numargs}\,(a) = 0} \qquad \textsc{Numargs-Con}\ \frac{}{\mathit{numargs}\,(T\,\overline{\sigma}) = 0} \qquad \textsc{Numargs-Arrow}\ \frac{\mathit{numargs}\,(\sigma_2) = m}{\mathit{numargs}\,(\sigma_1 \to \sigma_2) = m + 1}$$

$$\textsc{Numargs-Forall}\ \frac{\mathit{numargs}\,(\sigma) = m}{\mathit{numargs}\,(\forall\, a.\sigma) = m} \qquad \textsc{Numargs-InfForall}\ \frac{\mathit{numargs}\,(\sigma) = m}{\mathit{numargs}\,(\forall\, \{a\}.\sigma) = m}$$

Figure B.2: Counting Explicit Arguments

$\overline{\pi} \Rightarrow \overline{\psi}; \Delta$, $\Gamma, \Delta \vdash e \Rightarrow \eta^{\epsilon}$, $\mathit{type}\,(\overline{\psi}; \eta^{\epsilon}\ \sigma_1)$ and $\sigma = \forall\,\overline{\{a\}}_1.\sigma_1$ where $\overline{a}_1 = \mathbf{fv}\,(\sigma_1) \backslash \mathbf{dom}\,(\Gamma)$. The goal to be proven thus reduces to $\Gamma, \Delta \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \rho$, $\mathit{type}\,(\overline{\psi}; \rho\ \sigma_2)$ and $\sigma = \forall\,\overline{\{a\}}_2.\sigma_2$ where $\overline{a}_2 = \mathbf{fv}\,(\sigma_2) \setminus \mathbf{dom}\,(\Gamma)$. It is clear that the property can not hold under lazy instantiation, as rule Decl-NoAnnMulti performs an additional instantiation step, thus altering the type. Under eager instantiation, η^{ϵ} is already an instantiated type by the type inference relation, making the instantiation in the goal a no-op (by definition). The goal is thus trivially true. □

B.7 η-expansion

Property 11b (η-expansion is Type Preserving)**.**

- *If* $\Gamma \vdash e \Rightarrow \eta^{\epsilon}$ *where* $\mathit{numargs}(\eta^{\epsilon}) = n$ *and* $\Gamma \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \tau$ *then* $\Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Rightarrow \eta^{\epsilon}$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $\mathit{numargs}(r) = n$ *then* $\Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Leftarrow \sigma$

A formal definition of *numargs* is shown in Figure B.2. We prove Property 11b by first introducing a slightly more general lemma:

Lemma 60 (η-expansion is Type Preserving - Generalised)**.**

- *If* $\Gamma \vdash e \Rightarrow \eta^{\epsilon}$ *where* $0 \leqslant n \leqslant \mathit{numargs}(\eta^{\epsilon})$ *and* $\Gamma \vdash \eta^{\epsilon} \xrightarrow{inst\ \delta} \tau$ *then* $\Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Rightarrow \eta^{\epsilon}$
- *If* $\Gamma \vdash e \Leftarrow \sigma$ *where* $0 \leqslant n \leqslant \mathit{numargs}(\rho)$ *then* $\Gamma \vdash \lambda\overline{x}^{n}.e\,\overline{x}^{n} \Leftarrow \sigma$

The proof proceeds by induction on the integer n.

Case $n = 0$ **:**

This case is trivial, as it follows directly from the premise.

Case $n = m + 1 \leqslant \mathit{numargs}(\eta^\epsilon)$ **:**

case synthesis mode : We know from the induction hypothesis that $\Gamma \vdash \lambda \overline{x}^m . e\,\overline{x}^m \Rightarrow \eta^\epsilon$. We perform case analysis on this result (m repeated applications of rule Tm-InfAbs) to get $\Gamma, \overline{x_i : \tau_i}^{\,i<m} \vdash e\,\overline{x}^m \Rightarrow \eta_1^\epsilon$ where $\eta^\epsilon = \overline{\tau_i}^{\,i<m} \to \eta_1^\epsilon$. Performing case analysis again on this result (rule Tm-InfApp), gives us $\Gamma, \overline{x_i : \tau_i}^{\,i<m} \vdash^H e \Rightarrow \sigma_1$, $\Gamma, \overline{x_i : \tau_i}^{\,i<m} \vdash^A \overline{x}^m \Leftarrow \sigma_1 \Rightarrow \sigma_2$ and $\Gamma, \overline{x_i : \tau_i}^{\,i<m} \vdash \sigma_2 \xrightarrow{inst\ \delta} \eta_1^\epsilon$.

The goal to be proven is $\Gamma \vdash \lambda \overline{x}^{m+1} . e\,\overline{x}^{m+1} \Rightarrow \eta^\epsilon$, which (by rule TmInfAbs) reduces to $\Gamma, \overline{x_i : \tau_i}^{\,i<m}, x : \tau \vdash e\,\overline{x}^{m+1} \Rightarrow \eta_2^\epsilon$, where $\eta^\epsilon = \overline{\tau_i}^{\,i<m} \to \tau \to \eta_2^\epsilon$.

Note that this requires proving that $\eta_1^\epsilon = \tau \to \eta_2^\epsilon$. While we know that $m < \mathit{numargs}(\eta^\epsilon)$, we can only prove this under eager deep instantiation. Under lazy instantiation, type inference does not instantiate the result type at all. Under eager shallow, it is instantiated, but only up to the first function type. From here on out, we will thus assume eager deep instantiation. Furthermore, note that as even deep instantiation does not instantiate argument types, we need the additional premise that η^ϵ instantiates into a monotype, in order to prove this goal.

This result in turn (by rule Tm-InfApp) reduces to $\Gamma, \overline{x_i : \tau_i}^{\,i<m}, x : \tau \vdash^H e \Rightarrow \sigma_1$ (follows by weakening), $\Gamma, \overline{x_i : \tau_i}^{\,i<m}, x : \tau \vdash^A x, \overline{x}^m \Leftarrow \sigma_1 \Rightarrow \sigma_3$ (follows by rule Arg-Inst, rule Arg-App and the fact that $\eta_1^\epsilon = \tau \to \eta_2^\epsilon$) and $\Gamma, \overline{x_i : \tau_i}^{\,i<m}, x : \tau \vdash \sigma_3 \xrightarrow{inst\ \delta} \eta_2^\epsilon$ (follows by the definition of instantiation).

case checking mode : We know from the induction hypothesis that $\Gamma \vdash \lambda \overline{x}^m . e\,\overline{x}^m \Leftarrow \sigma$. The proof proceeds similarly to the synthesis mode case, by case analysis on this result (rule Tm-CheckAbs). One additional step is that rule Tm-CheckInf is applied to type $e\,\overline{x}^m$. The derivation switches to synthesis mode at this point, and becomes completely identical to the previous case. □

The proof for Property 11b now follows directly by Lemma 60, by taking $n = \mathit{numargs}(\eta^\epsilon)$. □

Appendix C

Coherence Proofs

C.1 Logical Relations

In the definitions for the logical relations below, $\gamma = \gamma', \delta \mapsto (d_1, d_2)$ and $\phi = \phi', x \mapsto (e_1, e_2)$ are substitutions which map all dictionary variables $\delta \in \Gamma$ and term variables $x \in \Gamma$ onto two (possibly different) dictionaries and terms respectively. Notation-wise, we adopted the convention that γ_1 maps the dictionary variable δ to the leftmost value dv_1 and γ_2 substitutes δ for the rightmost value dv_2. Similarly for ϕ_1 and ϕ_2.

The third kind of substitution $R = R', a \mapsto (\sigma, r)$ maps all type variables $a \in \Gamma$ onto closed types σ, while also storing a relation r. This relation r is an arbitrary member of the set of all relations $Rel[\sigma]$ which offer the following property:

$$Rel[\sigma] = \{r \in \mathcal{P}(v \times v) \ \mid \ \forall (v_1, v_2) \in r : \Sigma; \Gamma_C; \bullet \vdash_{tm} v_1 : \sigma \wedge \Sigma; \Gamma_C; \bullet \vdash_{tm} v_2 : \sigma\}$$

C.1.1 Dictionary Relation

$\boxed{(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}}$ (Closed Dictionary Value Relation)

$$(\Sigma_1 : D\,\overline{\sigma}_m\,\overline{d}_{n\,1}, \Sigma_2 : D\,\overline{\sigma}_m\,\overline{d}_{n\,2}) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_m\,\overline{d}_{n\,1} : Q \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d D\,\overline{\sigma}_m\,\overline{d}_{n\,2} : Q$$

$$\forall d_{1\,i}, d_{2\,i} : \overline{(\Sigma_1 : d_{1\,i}, \Sigma_2 : d_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_m/\overline{a}_m]C_i]\!]^{\Gamma_C}}^{i<n}$$

$$\mathit{where}(D : \forall \overline{a}_m.\overline{C}_n \Rightarrow Q').m \mapsto e_1 \in \Sigma_1 \wedge Q = [\overline{\sigma}_m/\overline{a}_m]Q'$$

$\boxed{(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C]\!]^{\Gamma_C}}$ (Closed Dictionary Relation)

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C_1 \Rightarrow C_2]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : C_1 \Rightarrow C_2 \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : C_1 \Rightarrow C_2$$

$$\wedge\, \forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![C_1]\!]^{\Gamma_C} \Rightarrow (\Sigma_1 : d_1\,d_3, \Sigma_2 : d_2\,d_4) \in \mathcal{E}[\![C_2]\!]^{\Gamma_C}$$

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![\forall a.C']\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : \forall a.C' \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : \forall a.C'$$

$$\wedge\, \forall \sigma : \Gamma_C;\bullet \vdash_{ty} \sigma \Rightarrow (\Sigma_1 : d_1\,\sigma, \Sigma_2 : d_2\,\sigma) \in \mathcal{E}[\![[\sigma/a]C']\!]^{\Gamma_C}$$

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![Q]\!]^{\Gamma_C}$$

$$\triangleq \Sigma_1;\Gamma_C;\bullet \vdash_d d_1 : Q \wedge \Sigma_2;\Gamma_C;\bullet \vdash_d d_2 : Q$$

$$\wedge\, \exists dv_1, dv_2, d_1 \longrightarrow^* dv_1, d_2 \longrightarrow^* dv_2, (\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}$$

$\boxed{\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C}$ (Logical Equivalence for Open Dictionaries)

$$\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C$$

$$\triangleq \forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}, \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C},$$

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

C.1.2 Expression Relation

$\boxed{(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]^{\Gamma_C}_{R}}$ (Closed Expression Value Relation)

$$(\Sigma_1 : \mathit{True}, \Sigma_2 : \mathit{True}) \in \mathcal{V}[\![\mathit{Bool}]\!]^{\Gamma_C}_{R}$$

$$(\Sigma_1 : \mathit{False}, \Sigma_2 : \mathit{False}) \in \mathcal{V}[\![\mathit{Bool}]\!]^{\Gamma_C}_{R}$$

$$(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![a]\!]^{\Gamma_C}_{R}$$

$$\triangleq (a \mapsto (\sigma, r)) \in R \wedge \Sigma_1; \Gamma_C; \bullet \vdash_{tm} v_1 : \sigma \wedge \Sigma_2; \Gamma_C; \bullet \vdash_{tm} v_2 : \sigma$$

$$\wedge\, (v_1, v_2) \in r$$

$$(\Sigma_1 : \lambda x : \sigma_1.e_1, \Sigma_2 : \lambda x : \sigma_1.e_2) \in \mathcal{V}[\![\sigma_1 \rightarrow \sigma_2]\!]^{\Gamma_C}_{R}$$

$$\triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_1 : R(\sigma_1 \rightarrow \sigma_2)$$

$$\wedge\, \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_2 : R(\sigma_1 \rightarrow \sigma_2)$$

$$\wedge\, \forall (\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]^{\Gamma_C}_{R} :$$

$$(\Sigma_1 : (\lambda x : \sigma.e_1)\, e_3, \Sigma_2 : (\lambda x : \sigma.e_2)\, e_4) \in \mathcal{E}[\![\sigma_2]\!]^{\Gamma_C}_{R}$$

$$(\Sigma_1 : \lambda \delta : C.e_1, \Sigma_2 : \lambda \delta : C.e_2) \in \mathcal{V}[\![C \Rightarrow \sigma]\!]^{\Gamma_C}_{R}$$

$$\triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : C.e_1 : R(C \Rightarrow \sigma) \wedge \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : C.e_2 : R(C \Rightarrow \sigma)$$

$$\wedge\, \forall (\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} :$$

$$(\Sigma_1 : (\lambda \delta : C.e_1)\, d_1, \Sigma_2 : (\lambda \delta : C.e_2)\, d_2) \in \mathcal{E}[\![\sigma]\!]^{\Gamma_C}_{R}$$

$$(\Sigma_1 : \Lambda a.e_1, \Sigma_2 : \Lambda a.e_2) \in \mathcal{V}[\![\forall a.\sigma]\!]^{\Gamma_C}_{R}$$

$$\triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_1 : R(\forall a.\sigma) \wedge \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_2 : R(\forall a.\sigma)$$

$$\wedge\, \forall \sigma', \forall r \in \mathit{Rel}[\sigma'] : \Gamma_C; \bullet \vdash_{ty} \sigma' \Rightarrow$$

$$(\Sigma_1 : (\Lambda a.e_1)\, \sigma', \Sigma_2 : (\Lambda a.e_2)\, \sigma') \in \mathcal{E}[\![\sigma]\!]^{\Gamma_C}_{R, a \mapsto (\sigma', r)}$$

$\boxed{(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma_1]\!]^{\Gamma_C}_{R}}$ (Closed Expression Relation)

$$(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

$$\triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_1 : R(\sigma) \wedge \Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_2 : R(\sigma)$$

$$\wedge \, \exists v_1, v_2, \Sigma_1 \vdash e_1 \longrightarrow^* v_1, \Sigma_2 \vdash e_2 \longrightarrow^* v_2, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C}$$

$\boxed{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma}$ (Logical Equivalence for Open Expressions)

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$$

$$\triangleq \forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}, \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}, \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C},$$

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

$\boxed{\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')}$ (Logical Equivalence for Contexts)

$$\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$$

$$\triangleq \forall e_1, e_2 : \Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$$

$$\Rightarrow \Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma'$$

Definition 14 (Interpretation of type variables in type contexts)**.**

$$\frac{}{\bullet \in \mathcal{F}[\![\bullet]\!]^{\Gamma_C}} \qquad \frac{R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}}{R \in \mathcal{F}[\![\Gamma, x : \sigma]\!]^{\Gamma_C}} \qquad \frac{\begin{array}{c} R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \\ r \in \mathit{Rel}[\sigma] \\ \Gamma_C ; \bullet \vdash_{ty} \sigma \end{array}}{R, a \mapsto (\sigma, r) \in \mathcal{F}[\![\Gamma, a]\!]^{\Gamma_C}}$$

$$\frac{R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}}{R \in \mathcal{F}[\![\Gamma, \delta : C]\!]^{\Gamma_C}}$$

Definition 15 (Interpretation of term variables in type contexts)**.**

$$\frac{}{\bullet \in \mathcal{G}[\![\bullet]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}} \qquad \frac{\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}{\phi \in \mathcal{G}[\![\Gamma, a]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}$$

$$\frac{\begin{array}{c} \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C} \\ (\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \end{array}}{\phi, x \mapsto (e_1, e_2) \in \mathcal{G}[\![\Gamma, x : \sigma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}} \qquad \frac{\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}{\phi \in \mathcal{G}[\![\Gamma, \delta : C]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}$$

Definition 16 (Interpretation of dictionary variables dictionary contexts)**.**

$$\frac{}{\bullet \in \mathcal{H}[\![\bullet]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}} \qquad \frac{\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}{\gamma \in \mathcal{H}[\![\Gamma, a]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}$$

$$\frac{\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}{\gamma \in \mathcal{H}[\![\Gamma, x : \sigma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}} \qquad \frac{\begin{array}{c} \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C} \\ (\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} \end{array}}{\gamma, \delta \mapsto (d_1, d_2) \in \mathcal{H}[\![\Gamma, \delta : C]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}}$$

C.1.3 Environment Relation

$\boxed{\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2}$ *(Logical Equivalence for Environments)*

CTXLOG-EMPTY

$$\frac{}{\Gamma_C \vdash \bullet \simeq_{log} \bullet}$$

CTXLOG-CONS

$$\frac{\begin{array}{c} \Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2 \\ \Gamma_C ; \bullet \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma' \end{array}}{\Gamma_C \vdash \Sigma_1, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma).m \mapsto e_1 \simeq_{log} \Sigma_2, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma).m \mapsto e_2}$$

C.2 Strong Normalization Relations

As opposed to Section C.1, the relations and substitutions in the strong normalization relations described below, are unary. The substitutions $\gamma^{SN} = \gamma^{SN'}, \delta \mapsto d$ and $\phi^{SN} = \phi^{SN'}, x \mapsto e$ map all dictionary variables $\delta \in \Gamma$ and term variables $x \in \Gamma$ onto well-typed dictionaries d and expressions $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. The final kind of substitution $R^{SN} = R^{SN'}, a \mapsto (\sigma, r)$ maps all type variables $a \in \Gamma$ onto closed types σ, while also storing a relation r. This relation r is an arbitrary member of the set of all relations $Rel[\sigma]$ which offer the following property:

$$Rel[\sigma] = \{r \in \mathcal{P}(e) \ \mid \ \forall e \in r : \Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma\}$$

We adopted the convention that ${R^{SN}}_1(a)$ maps the type variable a onto the closed type σ and ${R^{SN}}_2(a)$ denotes the contained set of expressions r.

C.2.1 Dictionary Relation

$\boxed{d \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}}$ (Strong Normalization Relation for Dictionaries)

$$d \in \mathcal{SN}[\![TC\,\sigma]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : TC\,\sigma \wedge \exists D, \overline{\sigma}_j, \overline{d}_i : d \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i$$

$$\text{where } \Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2$$

$$\wedge\ (m : TC\,a : \sigma_m) \in \Gamma_C \wedge\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_{ty} \sigma_j}^{\,j}$$

$$\wedge\ \overline{d_i \in \mathcal{SN}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Sigma,\Gamma_C}}^{\,i} \wedge\ \sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q$$

$$\wedge\, e \in \mathcal{SN}[\![\forall \overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma_m]\!]_{\bullet}^{\Sigma_1,\Gamma_C}$$

$$d \in \mathcal{SN}[\![C_1 \Rightarrow C_2]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : C_1 \Rightarrow C_2 \wedge \exists dv : d \longrightarrow^* dv$$

$$\wedge\, \forall d' : d' \in \mathcal{SN}[\![C_1]\!]^{\Sigma,\Gamma_C} \Rightarrow d\,d' \in \mathcal{SN}[\![C_2]\!]^{\Sigma,\Gamma_C}$$

$$d \in \mathcal{SN}[\![\forall a.C]\!]^{\Sigma,\Gamma_C}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_d d : \forall a.C \wedge \exists dv : d \longrightarrow^* dv$$

$$\wedge\, \forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow d\,\sigma \in \mathcal{SN}[\![[\sigma/a]C]\!]^{\Sigma,\Gamma_C}$$

C.2.2 Expression Relation

$\boxed{e \in \mathcal{SN}[\![\sigma]\!]_{R^{SN}}^{\Sigma,\Gamma_C}}$ (Strong Normalization Relation)

$$e \in \mathcal{SN}[\![Bool]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e : Bool \wedge \exists v : \Sigma \vdash e \longrightarrow^* v$$

$$e \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e : {R^{SN}}_1(a) \wedge \exists v : \Sigma \vdash e \longrightarrow^* v$$

$$\wedge\, v \in {R^{SN}}_2(a)$$

$$e \in \mathcal{SN}[\![\sigma_1 \to \sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e : {R^{SN}}_1(\sigma_1 \to \sigma_2) \wedge \exists v : \Sigma \vdash e \longrightarrow^* v$$

$$\wedge\, \forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$e \in \mathcal{SN}[\![C \Rightarrow \sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e : {R^{SN}}_1(C \Rightarrow \sigma) \wedge \exists v : \Sigma \vdash e \longrightarrow^* v$$

$$\wedge\, \forall d : d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C} \Rightarrow e\, d \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$e \in \mathcal{SN}[\![\forall a.\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

$$\triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e : {R^{SN}}_1(\forall a.\sigma) \wedge \exists v : \Sigma \vdash e \longrightarrow^* v$$

$$\wedge\, \forall \sigma', r \in Rel[\sigma'] : e\, \sigma' \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (\sigma', r)}$$

Definition 17 (Interpretation of type variables in type contexts for strong normalization)**.**

$$\frac{}{\bullet \in \mathcal{F}^{\mathcal{SN}}[\![\bullet]\!]^{\Sigma,\Gamma_C}} \qquad \frac{R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}}{R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma, x : \sigma]\!]^{\Sigma,\Gamma_C}}$$

$$\frac{\begin{array}{c} R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C} \\ r = \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \\ \Gamma_C; \bullet \vdash_{ty} {R^{SN}}_1(\sigma) \end{array}}{R^{SN}, a \mapsto ({R^{SN}}_1(\sigma), r) \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma, a]\!]^{\Sigma,\Gamma_C}} \qquad \frac{R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}}{R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma, \delta : C]\!]^{\Sigma,\Gamma_C}}$$

Definition 18 (Interpretation of term variables in type contexts for strong normalization)**.**

$$\frac{}{\bullet \in \mathcal{G}^{\mathcal{SN}}[\![\bullet]\!]^{\Sigma,\Gamma_C}_{R^{SN}}} \qquad \frac{\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}{\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma, a]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}$$

$$\frac{\begin{array}{c}\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \\ e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}\end{array}}{\phi^{SN}, x \mapsto e \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma, x : \sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}} \qquad \frac{\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}{\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma, \delta : C]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}$$

Definition 19 (Interpretation of dictionary variables dictionary contexts for strong normalization)**.**

$$\frac{}{\bullet \in \mathcal{H}^{\mathcal{SN}}[\![\bullet]\!]^{\Sigma,\Gamma_C}_{R^{SN}}} \qquad \frac{\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}{\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma, a]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}$$

$$\frac{\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}{\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma, x : \sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}} \qquad \frac{\begin{array}{c}\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \\ d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}\end{array}}{\gamma^{SN}, \delta \mapsto d \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma, \delta : C]\!]^{\Sigma,\Gamma_C}_{R^{SN}}}$$

C.3 Equivalence Relations

C.3.1 Kleene Equivalence Relations

$\boxed{\Sigma_1 : e_1 \simeq \Sigma_2 : e_2}$ (Kleene Equivalence for $F_{\mathbf{D}}$ Expressions)

$$\Sigma_1 : e_1 \simeq \Sigma_2 : e_2 \triangleq \exists v : \Sigma_1 \vdash e_1 \longrightarrow^* v \wedge \Sigma_2 \vdash e_2 \longrightarrow^* v$$

$\boxed{e_1 \simeq e_2}$ (Kleene Equivalence for $F_{\{\}}$ Expressions)

$$e_1 \simeq e_2 \triangleq \exists v : e_1 \longrightarrow^* v \wedge e_2 \longrightarrow^* v$$

C.3.2 Contextual Equivalence Relations

$\boxed{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma}$ (Contextual Equivalence for $F_{\mathbf{D}}$ Expressions)

$$\begin{aligned}
&\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma \\
&\qquad \triangleq \forall M_1 : (\Sigma_1;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_1;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \\
&\qquad \wedge \forall M_2 : (\Sigma_2;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_2;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \\
&\qquad \wedge \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \\
&\qquad \Rightarrow \Sigma_1 : M_1[e_1] \simeq \Sigma_2 : M_2[e_2]
\end{aligned}$$

$\boxed{P;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau}$ (Contextual Equivalence for $F_{\{\}}$ Expressions)

$$\begin{aligned}
&P;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau \\
&\qquad \triangleq \forall M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1 \\
&\qquad \wedge M : (P;\Gamma_C;\Gamma \Rightarrow \tau) \mapsto (P;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2 \\
&\qquad \Rightarrow M_1[e_1] \simeq M_2[e_2]
\end{aligned}$$

$\boxed{\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma}$ (Contextual Equivalence for $F_{\{\}}$ Expressions in $F_{\mathbf{D}}$ context)

$$\begin{aligned}
&\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma \\
&\qquad \triangleq \forall M_1 : (\Sigma_1;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_1;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1 \\
&\qquad \wedge \forall M_2 : (\Sigma_2;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_2;\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2 \\
&\qquad \wedge \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Gamma_C;\bullet \Rightarrow \mathit{Bool}) \\
&\qquad \Rightarrow M_1[e_1] \simeq M_2[e_2]
\end{aligned}$$

C.4 $\lambda^{\Rightarrow}_{\mathbf{TC}}$ Theorems

C.4.1 Conjectures

We are confident that the following lemmas can be proven using well-known proof techniques.

Lemma 61 (Type Variable Substitution in $\lambda_{\mathbf{TC}}^{\Rightarrow}$ Constraint Typing)**.** *If* $\Gamma_C; \Gamma_1, a, \Gamma_2 \vdash_C^M C \rightsquigarrow C$ *and* $\Gamma_C; \Gamma_1 \vdash_{ty}^M \tau \rightsquigarrow \sigma$ *then* $\Gamma_C; \Gamma_1, [\tau/a]\Gamma_2 \vdash_C^M [\tau/a]C \rightsquigarrow [\sigma/a]C$*.*

Lemma 62 (Type Well-Formedness Environment Weakening)**.** *If* $\Gamma_{C1}; \Gamma_1 \vdash_{ty}^M \sigma \rightsquigarrow \sigma$ *and* $\vdash_{ctx}^M \bullet; \Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \rightsquigarrow \bullet; \Gamma_C; \Gamma$ *then* $\Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \vdash_{ty}^M \sigma \rightsquigarrow \sigma$*.*

Lemma 63 (Class Constraint Well-Formedness Environment Weakening)**.** *If* $\Gamma_{C1}; \Gamma_1 \vdash_Q^M Q \rightsquigarrow Q'$ *and* $\vdash_{ctx}^M \bullet; \Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \rightsquigarrow \bullet; \Gamma_C; \Gamma$ *then* $\Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \vdash_Q^M Q \rightsquigarrow Q'$*.*

Lemma 64 (Constraint Well-Formedness Environment Weakening)**.** *If* $\Gamma_{C1}; \Gamma_1 \vdash_C^M C \rightsquigarrow C'$ *and* $\vdash_{ctx}^M \bullet; \Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \rightsquigarrow \bullet; \Gamma_C; \Gamma$ *then* $\Gamma_{C1}, \Gamma_{C2}; \Gamma_1, \Gamma_2 \vdash_C^M C \rightsquigarrow C'$*.*

Lemma 65 (Context Well-Formedness Class Environment Weakening)**.** *If* $\vdash_{ctx}^M P; \Gamma_{C1}; \Gamma \rightsquigarrow \Sigma; \Gamma_{C1}; \Gamma$ *and* $\vdash_{ctx}^M \bullet; \Gamma_{C1}, \Gamma_{C2}; \Gamma \rightsquigarrow \bullet; \Gamma_{C1}, \Gamma_{C2}; \Gamma$ *then* $\vdash_{ctx}^M P; \Gamma_{C1}, \Gamma_{C2}; \Gamma \rightsquigarrow \Sigma; \Gamma_{C1}, \Gamma_{C2}; \Gamma$*.*

Lemma 66 (Context Well-Formedness Typing Environment Weakening)**.** *If* $\vdash_{ctx}^M P; \Gamma_C; \Gamma_1 \rightsquigarrow \Sigma; \Gamma_C; \Gamma_1$ *and* $\vdash_{ctx}^M \bullet; \Gamma_C; \Gamma_1, \Gamma_2 \rightsquigarrow \bullet; \Gamma_C; \Gamma_1, \Gamma_2$ *then* $\vdash_{ctx}^M P; \Gamma_C; \Gamma_1, \Gamma_2 \rightsquigarrow \Sigma; \Gamma_C; \Gamma_1, \Gamma_2$*.*

C.4.2 Lemmas

Lemma 67 (Determinism of Context Typing)**.**

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_1) \rightsquigarrow M_1$ *and* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_2$
  then $\tau_1 = \tau_2$*.*

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1) \rightsquigarrow M_1$ *and* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_2) \rightsquigarrow M_2$
  then $\tau_1 = \tau_2$.
- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_1) \rightsquigarrow M_1$ *and* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_2$
  then $\tau_1 = \tau_2$.
- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1) \rightsquigarrow M_1$ *and* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_2) \rightsquigarrow M_2$
  then $\tau_1 = \tau_2$.

Proof. By straightforward induction on the 1^{st} typing derivation, in combination with case analysis on the second derivation.

□

Lemma 68 (Class Constraint Elaboration to $F_{\mathbf{D}}$ Uniqueness)**.**
If $\Gamma_C; \Gamma \vdash_Q^M Q \rightsquigarrow Q_1$ *and* $\Gamma_C; \Gamma \vdash_Q^M Q \rightsquigarrow Q_2$*, then* $Q_1 = Q_2$.

Proof. By mutual induction on both well-formedness derivations, together with Lemma 69.

□

Lemma 69 (Type Elaboration to $F_{\mathbf{D}}$ Uniqueness)**.**
If $\Gamma_C; \Gamma \vdash_{ty}^M \sigma \rightsquigarrow \sigma_1$ *and* $\Gamma_C; \Gamma \vdash_{ty}^M \sigma \rightsquigarrow \sigma_2$*, then* $\sigma_1 = \sigma_2$.

Proof. By mutual induction on both well-formedness derivations, together with Lemma 68.

□

Lemma 70 (Constraint Elaboration to $F_{\mathbf{D}}$ Uniqueness)**.**
If $\Gamma_C; \Gamma \vdash_C^M C \rightsquigarrow C_1$ *and* $\Gamma_C; \Gamma \vdash_C^M C \rightsquigarrow C_2$*, then* $C_1 = C_2$.

Proof. By straightforward induction on both well-formedness derivations, in combination with Lemma 68.

□

Lemma 71 (Environment Elaboration to $F_{\mathbf{D}}$ Uniqueness)**.** *If* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_{C1}; \Gamma_2$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_{C2}; \Gamma_2$*, then* $\Gamma_{C1} = \Gamma_{C2}$ *and* $\Gamma_1 = \Gamma_2$*.*

Proof. By straightforward induction on both well-formedness derivations, in combination with Lemmas 68, 69 and 70.

□

Lemma 72 (Environment Well-Formedness of $\lambda^{\Rightarrow}_{\mathbf{TC}}$ Typing)**.**

- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$ *then* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$*.*
- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e$ *then* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$*.*

Proof. By straightforward induction on the typing derivation.

□

Lemma 73 (Environment Well-Formedness of $\lambda^{\Rightarrow}_{\mathbf{TC}}$ Typing through $F_{\mathbf{D}}$)**.**

- *If* $P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e$ *then* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*.*
- *If* $P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e$ *then* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*.*

Proof. By straightforward induction on the typing derivation.

□

Lemma 74 (Well-Formedness of $\lambda^{\Rightarrow}_{\mathbf{TC}}$ Typing Result)**.**

- *If* $P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e$ *then* $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$*.*
- *If* $P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e$ *then* $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$*.*

Proof. By straightforward induction on the typing derivation.

□

Lemma 75 (Preservation of Environment Term Variables from $\lambda^{\Rightarrow}_{\mathbf{TC}}$ to $F_{\mathbf{D}}$)**.**

- *If* $(x : \sigma) \in \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $(x : \sigma) \in \Gamma$ *where* $\Gamma_C; \Gamma \vdash^{M}_{ty} \sigma \rightsquigarrow \sigma$.
- *If* $x \notin \boldsymbol{dom}(\Gamma)$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $x \notin \boldsymbol{dom}(\Gamma)$.

Proof. By straightforward induction on the environment elaboration derivation. □

Lemma 76 (Preservation of Environment Type Variables from $\lambda^{\Rightarrow}_{\mathbf{TC}}$ to $F_{\mathbf{D}}$)**.**

- *If* $a \in \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $a \in \Gamma$.
- *If* $a \notin \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $a \notin \Gamma$.

Proof. By straightforward induction on the environment elaboration derivation. □

Lemma 77 (Preservation of Environment Dictionary Variables from $\lambda^{\Rightarrow}_{\mathbf{TC}}$ to $F_{\mathbf{D}}$)**.**

- *If* $(\delta : C) \in \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $(\delta : C) \in \Gamma$ *where* $\Gamma_C; \Gamma \vdash^{M}_{C} C \rightsquigarrow C$.
- *If* $\delta \notin \boldsymbol{dom}(\Gamma)$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $\delta \notin \boldsymbol{dom}(\Gamma)$.

Proof. By straightforward induction on the environment elaboration derivation. □

Lemma 78 (Preservation of Environment Classes from $\lambda^{\Rightarrow}_{\mathbf{TC}}$ to $F_{\mathbf{D}}$)**.**

- *If* $(m : \overline{C}_i \Rightarrow TC\, a : \sigma) \in \Gamma_C$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $(m : TC\, a : \sigma) \in \Gamma_C$ *where* $\Gamma_C; \Gamma \vdash^{M}_{ty} \sigma \rightsquigarrow \sigma$.
- *If* $m \notin \boldsymbol{dom}(\Gamma_C)$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $m \notin \boldsymbol{dom}(\Gamma_C)$.

Proof. By straightforward induction on the environment elaboration derivation. □

Lemma 79 (Preservation of Environment Instances from $\lambda_{\mathbf{TC}}^{\Rightarrow}$ to $F_{\mathbf{D}}$)**.**

- *If* $(D : C).m \mapsto \Gamma' : e \in P$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $(D : C).m \mapsto e \in \Sigma$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d D : C$ *where* $\Gamma_C; \Gamma \vdash_{C}^{M} C \rightsquigarrow C$.
- *If* $D \notin \boldsymbol{dom}(P)$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *then* $D \notin \boldsymbol{dom}(\Sigma)$.

Proof. By case analysis the environment elaboration derivation. This theorem is mutually proven with Theorems 21, 26, 27 and 28 (Figure C.1). Note that at the dependency from Theorem 28 to 21, and from Lemma 79 to Theorem 21, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorem 26 and 27 perform induction over a (finite) derivation. Consequently, the size of P is strictly decreasing in every possible cycle. The induction thus remains well-founded.

Since we know that $(D : C).m \mapsto \Gamma' : e \in P$, we know from rule sCtx-pgmInst that

$$\Gamma_C; \bullet \vdash_{C}^{M} C \rightsquigarrow C \tag{C.1}$$

$$(m : \overline{C}'_m \Rightarrow TC\, a : \sigma) \in \Gamma_C \tag{C.2}$$

$$P_1; \Gamma_C; \Gamma' \vdash_{tm}^{M} e \Leftarrow \tau_1 \rightsquigarrow e \tag{C.3}$$

$$\Gamma_C; \bullet, a \vdash_{ty}^{M} \sigma \rightsquigarrow \sigma \tag{C.4}$$

$$\Sigma = \Sigma_1, (D : C).m \mapsto e', \Sigma_2 \tag{C.5}$$

$$\vdash_{ctx}^{M} P_1; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma \tag{C.6}$$

where e' is obtained by abstracting over the typing environment Γ'. By rule D-con, the remaining goals to be proven are

$$(m : TC\, a : \sigma_m) \in \Gamma_C \tag{C.7}$$

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.8}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i}^{\,i} \tag{C.9}$$

$$\Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e' : [\sigma_q/a]\sigma_m \tag{C.10}$$

Note here that $C = \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma_q$ and $e = \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e'$. Goal C.7 follows from Lemma 78 and Equation C.2. Goal C.8 follows by Theorem 28 and the second hypothesis. By applying Theorem 25 to Equation C.1 we get

$$\Gamma_C ; \Gamma \vdash_C C$$

Goal C.9 follows from this result through repeated case analysis (rule iC-forall and rule iC-arrow). Finally, Goal C.10 follows by applying Theorem 21 to Equation C.3, followed by repeated case analysis (rule iTm-forallI and rule iTm-constrI).

□

Lemma 80 (Environment Well-Formedness Strengthening).
If $\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma$ *then* $\vdash_{ctx} P ; \Gamma_C ; \bullet \rightsquigarrow \bullet$.

Proof. By case analysis on the hypothesis, the last rules used to construct it must be (possibly zero) consecutive applications of rule sCtxT-pgmInst. Revert those rules, to obtain $\vdash_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma$. By further case analysis (rule sCtxT-tyEnvTm, rule sCtxT-tyEnvTy and rule sCtxT-tyEnvD), we get $\vdash_{ctx} \bullet ; \Gamma_C ; \bullet \rightsquigarrow \bullet$. The goal follows by consecutively re-applying rule rule sCtxT-pgmInst with the appropriate premises.

□

Lemma 81 (Environment Well-Formedness with $F_{\mathbf{D}}$ Elaboration Strengthening).
If $\vdash^{M}_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Sigma ; \Gamma_C ; \Gamma$ *then* $\vdash^{M}_{ctx} P ; \Gamma_C ; \bullet \rightsquigarrow \Sigma ; \Gamma_C ; \bullet$.

Proof. By case analysis on the hypothesis, the last rules used to construct it must be (possibly zero) consecutive applications of rule sCtx-pgmInst. Revert those rules, to obtain $\vdash^{M}_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \bullet ; \Gamma_C ; \Gamma$. By further case analysis (rule sCtx-tyEnvTm, rule sCtx-tyEnvTy and rule sCtx-tyEnvD), we get $\vdash^{M}_{ctx} \bullet ; \Gamma_C ; \bullet \rightsquigarrow \bullet ; \Gamma_C ; \bullet$. The goal follows by consecutively re-applying rule rule sCtx-pgmInst with the appropriate premises.

□

C.4.3 Typing Preservation

Theorem 21 (Typing Preservation - Expressions).

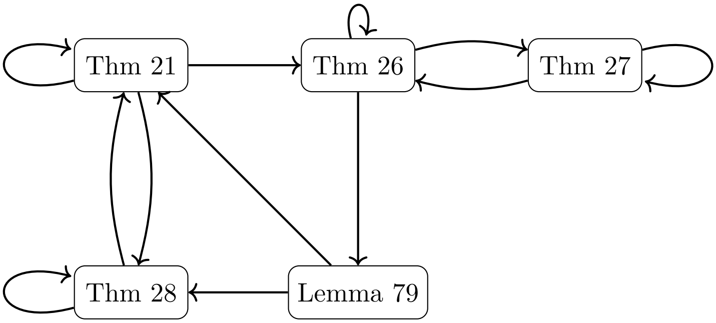

Figure C.1: Dependency graph for Typing Preservation Theorems

- *If* $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e$*, and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$*, then* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*, and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$*.*
- *If* $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e$*, and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$*, then* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*, and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$*.*

Proof. This theorem is mutually proven with Theorems 26, 27, and 28, as well as Lemma 79. This mutual dependency is illustrated in Figure C.1, where an arrow from A to B denotes A being dependent on B. Note that at the dependency from Theorem 28 to 21, and from Lemma 79 to Theorem 21, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorem 26 and 27 perform induction over a (finite) derivation. Consequently, the size of P is strictly decreasing in every possible cycle. The induction thus remains well-founded.

By applying Lemma 73 to the 1st hypothesis, we get:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \tag{C.11}$$

We continue by induction on the lexicographic order of the tuple (size of the expression, typing mode). Regarding typing mode, we define type checking to be larger than type inference. In each mutual dependency, we know that the tuple size decreases, meaning that the induction is well-founded.

Part 1

$$\boxed{\textbf{rule } \textsc{sTm-inf-true}} \qquad \frac{\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vdash^M_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}}\ \textsc{sTm-inf-true}$$

By rule sTy-bool, we know that:

$$\Gamma_C;\Gamma \vdash^M_{ty} \mathit{Bool} \rightsquigarrow \mathit{Bool}$$

The goal to be proven is the following:

$$\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}$$

From Theorem 28, we know that:

$$\vdash_{ctx} \Sigma;\Gamma_C;\Gamma$$

The goal follows from rule iTm-true.

$\boxed{\textbf{rule sTm-inf-false}}$

$$\textsc{sTm-inf-false}\quad \frac{\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma}{P;\Gamma_C;\Gamma \vdash^M_{tm} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}}$$

Similar to the rule sTm-inf-true case.

$\boxed{\textbf{rule sTm-inf-let}}$

sTm-inf-let

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \\ \mathbf{unambig}(\forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1) \\ \mathbf{closure}(\Gamma_C;\overline{C}_i) = \overline{C}_k \\ \Gamma_C;\Gamma \vdash^M_{ty} \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \\ P;\Gamma_C;\Gamma,\overline{a}_j,\overline{\delta}_k : \overline{C}_k \vdash^M_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P;\Gamma_C;\Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \vdash^M_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \\ e = \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j.\lambda \overline{\delta}_k : \overline{C}_k.e_1 \ \mathbf{in}\ \ e_2 \end{array}}{P;\Gamma_C;\Gamma \vdash^M_{tm} \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1 \ \mathbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e}$$

Given

$$\Gamma_C;\Gamma \vdash^M_{ty} \tau_2 \rightsquigarrow \sigma_2$$

The goal to be proven is the following:

$$\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j.\lambda \overline{\delta}_k : \overline{C}_k.e_1 \ \mathbf{in}\ \ e_2 : \sigma_2$$

By case analysis on Equation C.11 (rule sCtx-pgmInst), we know:

$$\vdash^M_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \bullet;\Gamma_C;\Gamma \tag{C.12}$$

From the rule premise we know that:

$$\Gamma_C;\Gamma \vdash^M_{ty} \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma \tag{C.13}$$

Applying Theorem 25 to Equations C.12 and C.13, we get that:

$$\Gamma_C;\Gamma \vdash_{ty} \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma \tag{C.14}$$

By repeated case analysis on Equation C.13 (rule sTy-scheme and rule sTy-qual), we get that:

$$\frac{\overline{a}_j \notin \Gamma}{\overline{\Gamma_C; \Gamma, \overline{a}_j \vdash^M_C C_k \rightsquigarrow C_k}^{k}}$$

Applying these results, together with Equation C.12, to rule sCtx-tyEnvTy and rule sCtx-tyEnvD, we get:

$$\vdash^M_{ctx} \bullet; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \rightsquigarrow \bullet; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \tag{C.15}$$

By weakening (Lemma 66) on Equations C.11 and C.15, we get:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \rightsquigarrow \Sigma; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \tag{C.16}$$

The rule premise also gives us that:

$$P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash^M_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \tag{C.17}$$

By applying induction hypothesis with Equations C.13 and C.17, we get that:

$$\Sigma; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash_{tm} e_1 : \sigma$$

Because of rule iTm-constrI and rule iTm-forallI, it is equivalent to say that:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1 : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma \tag{C.18}$$

Through a similar analysis, we get that:

$$\Sigma; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma \vdash_{tm} e_2 : \sigma_2 \tag{C.19}$$

By rule iTm-let, in combination with Equations C.14, C.18 and C.19, the goal has been proven.

$\boxed{\textbf{rule } \textsc{sTm-inf-ArrE}}$

sTm-inf-ArrE

$$\frac{P; \Gamma_C; \Gamma \vdash^M_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \qquad P; \Gamma_C; \Gamma \vdash^M_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2}{P; \Gamma_C; \Gamma \vdash^M_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2}$$

From the rule premise:

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \tag{C.20}$$

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \tag{C.21}$$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2$$

where $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_2 \rightsquigarrow \sigma_2$.

Because the typing result is well-formed (Lemma 74), we know:

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2$$

By applying the induction hypothesis on Equations C.20 and C.21, we know respectively:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_1$$

The goal follows from rule iTm-arrE.

rule sTm-inf-Ann

$$\textsc{sTm-inf-Ann}\quad \frac{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e}$$

Follows directly from the induction hypothesis.

Part 2

rule sTm-check-var

$$\textsc{sTm-check-var}\quad \frac{\begin{array}{c}(x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma \\ \textbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \\ \overline{P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j / \overline{a}_j] C_i] \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j / \overline{a}_j] \tau \rightsquigarrow x\, \overline{\sigma}_j\, \overline{d}_i}$$

From the rule premise:

$$(x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma$$

By repeated case analysis on Equation C.11 (rule sCtx-pgmInst), we get that:

$$\vdash_{ctx}^{M} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma \tag{C.22}$$

By case analysis on Equation C.22 (rule sCtx-tyEnvTm), we know:

$$(x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma \tag{C.23}$$

$$\Gamma_C; \Gamma_1 \vdash_{ty}^{M} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \rightsquigarrow \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma \tag{C.24}$$

where $\Gamma = \Gamma_1, x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau, \Gamma_2$.

By applying Lemma 75 to Equation C.23, we get:

$$(x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma) \in \Gamma \tag{C.25}$$

Furthermore, from the rule premise, we know that:

$$\overline{\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \tag{C.26}$$

By Typing Preservation - Types (Theorem 25), together with Equation C.22, we have:

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_j}^{j} \tag{C.27}$$

Similarly, the rule premise tells us that:

$$\overline{P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j/\overline{a}_j]C_i] \rightsquigarrow d_i}^{i} \tag{C.28}$$

By applying weakening (Lemma 62) to Equation C.24, we get:

$$\Gamma_C; \Gamma \vdash^{M}_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma \tag{C.29}$$

By repeated case analysis on Equation C.29 (rule sTy-qual), we get that:

$$\overline{\Gamma_C; \Gamma, \overline{a}_j \vdash^{M}_{C} C_i \rightsquigarrow C_i}^{i} \tag{C.30}$$

By applying Lemma 61 on Equations C.30 and C.26, we get:

$$\overline{\Gamma_C; \Gamma \vdash^{M}_{C} [\overline{\tau}_j/\overline{a}_j]C_i \rightsquigarrow [\overline{\sigma}_j/\overline{a}_j]C_i}^{i} \tag{C.31}$$

By Typing Preservation - Constraint Entailment (Theorem 26), applied to Equations C.28, C.11 and C.31, we have:

$$\overline{\Sigma; \Gamma_C; \Gamma \vdash_{d} d_i : [\overline{\sigma}_j/\overline{a}_j]C_i}^{i} \tag{C.32}$$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} x\, \overline{\sigma}_j\, \overline{d}_i : [\overline{\sigma}_j/\overline{a}_j]\sigma$$

where $\Gamma_C; \Gamma \vdash^{M}_{ty} [\overline{\tau}_j/\overline{a}_j]\tau \rightsquigarrow [\overline{\sigma}_j/\overline{a}_j]\sigma$.

From Equation C.25, by applying rule iTm-var, we get

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma \tag{C.33}$$

By Equations C.27, C.33 and rule iTm-forallE, we get:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} x\, \overline{\sigma}_j : [\overline{\sigma}_j/\overline{a}_j]\overline{C}_i \Rightarrow [\overline{\sigma}_j/\overline{a}_j]\sigma \tag{C.34}$$

By Equations C.32 and C.34, in combination with rule rule ITM-CONSTRE, we get

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} x\, \overline{\sigma}_j\, \overline{d}_i : [\overline{\sigma}_j/\overline{a}_j]\sigma \tag{C.35}$$

which is exactly the goal.

rule STM-CHECK-METH

STM-CHECK-METH

$$\frac{\begin{array}{c} (m : \overline{C}'_k \Rightarrow TC\, a : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{C}_i \Rightarrow \tau') \\ P; \Gamma_C; \Gamma \vDash^M [TC\, \tau] \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma \\ \overline{P; \Gamma_C; \Gamma \vDash^M [[\overline{\tau}_j/\overline{a}_j][\tau/a]C_i] \rightsquigarrow d_i}^{\,i} \\ \overline{\Gamma_C; \Gamma \vdash^M_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \\ \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash^M_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\, \overline{\sigma}_j\, \overline{d}_i}$$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m\, \overline{\sigma}_j\, \overline{d}_i : [\overline{\sigma}_j/\overline{a}_j][\sigma/a]\sigma' \tag{C.36}$$

where $\Gamma_C; \bullet, \overline{a}_j, a \vdash^M_{ty} \tau' \rightsquigarrow \sigma'$.

From the rule premise, we get that:

$$P; \Gamma_C; \Gamma \vDash^M [TC\, \tau] \rightsquigarrow d \tag{C.37}$$

$$\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma \tag{C.38}$$

By repeated case analysis on Equation C.11 (rule SCTX-CLSENV), together with the 1st rule premise, we get:

$$\Gamma_{C1}; \bullet, a \vdash^M_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau' \rightsquigarrow \sigma''$$

where $\Gamma_C = \Gamma_{C1}, m : \overline{C}'_k \Rightarrow TC\, a : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau', \Gamma_{C2}$.

Following rule SQ-TC, in combination with this result, Equation C.38 and the 1st rule premise, we have:

$$\Gamma_C; \Gamma \vdash^M_Q TC\, \tau \rightsquigarrow TC\, \sigma \tag{C.39}$$

Applying Typing Preservation - Constraints Proving (Theorem 26) on Equations C.37 and C.39, we get:

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\, \sigma \tag{C.40}$$

Furthermore, we know from the rule premise that:

$$(m : \overline{C}'_k \Rightarrow TC\, a : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau') \in \Gamma_C$$

Consequently, by repeated case analysis on Equation C.11 (rule sCtx-clsEnv), we know that:

$$(m : TC\, a : \sigma'') \in \Gamma_C \tag{C.41}$$

By Equations C.40 and C.41, in combination with rule rule iTm-method, we get:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma''$$

The rest of the proof is similar to case rule sTm-check-var.

rule sTm-check-ArrI

sTm-check-ArrI

$$\frac{x \notin \mathbf{dom}(\Gamma) \qquad P; \Gamma_C; \Gamma, x : \tau_1 \vdash^M_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \qquad \Gamma_C; \Gamma \vdash^M_{ty} \tau_1 \rightsquigarrow \sigma}{P; \Gamma_C; \Gamma \vdash^M_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma.e}$$

The second hypothesis is:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$$

It is easy to verify that

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma, x : \tau_1 \rightsquigarrow \Sigma; \Gamma_C; \Gamma, x : \sigma$$

The goal follows directly by applying the induction hypothesis, in combination with rule rule iTm-arrI.

rule sTm-check-Inf

sTm-check-Inf

$$\frac{P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e}$$

Follows directly from the induction hypothesis.

□

Theorem 22 (Typing Preservation - Instance)**.** *If* $P; \Gamma_C \vdash^M_{inst} inst : P'$*, and* $\vdash^M_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \Sigma; \Gamma_C; \bullet$ *then we have* $\vdash^M_{ctx} P, P'; \Gamma_C; \bullet \rightsquigarrow \Sigma, \Sigma'; \Gamma_C; \bullet$.

Proof. We restate the rule for typing instance declarations for reference:

sInst-inst

$$\dfrac{\begin{array}{c} (m : \overline{C}'_i \Rightarrow TC\,a : \forall \overline{a}_j.\overline{C}_y \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \\ \Gamma_C; \bullet, \overline{b}_k \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \\ \left.\dfrac{\left.\dfrac{\begin{array}{c}\mathbf{closure}(\Gamma_C; \overline{C}_p) = \overline{C}_q \\ \mathbf{unambig}(\forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau)\end{array}}{\Gamma_C; \bullet, \overline{b}_k \vdash^{M}_{C} C_q \rightsquigarrow C_q}\right.^{q}}{P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q \vDash^{M} [[\tau/a]C'_i] \rightsquigarrow d_i}\right.^{i} \\ P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y \vdash^{M}_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ D\ \mathbf{fresh} \\ \overline{\delta}_y\ \mathbf{fresh} \qquad \overline{\delta}_q\ \mathbf{fresh} \\ (D' : \forall \overline{b}'_m.\overline{C}'_n \Rightarrow TC\,\tau_2).m' \mapsto \Gamma' : e' \notin P\ \textit{where}\ [\overline{\tau}'_m/\overline{b}'_m]\tau_2 = [\overline{\tau}'_k/\overline{b}_k]\tau \\ P' = (D : \forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y : e \end{array}}{P; \Gamma_C \vdash^{M}_{inst} \mathbf{instance}\ \overline{C}_p \Rightarrow TC\,\tau\,\mathbf{where}\ \{m = e\} : P'}$$

By inversion of rule rule sInst-inst, we know that:

$$P' = (D : \forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y : e$$

Therefore our goal is

$$\vdash^{M}_{ctx} P, (D : \forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_k, \overline{\delta}_q : \overline{C}_q, \overline{a}_j, \overline{\delta}_y : [\tau/a]\overline{C}_y : e; \Gamma_C; \bullet \rightsquigarrow \Sigma, \Sigma'; \Gamma_C; \bullet \tag{C.42}$$

From the hypothesis, we know that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \Sigma; \Gamma_C; \bullet \tag{C.43}$$

Goal C.42 follows directly from rule sCtx-pgmInst with

$$\Sigma' = (D : \forall \overline{b}_k.\overline{C}_q \Rightarrow TC\,\sigma).m \mapsto \Lambda \overline{b}_k.\lambda \overline{\delta}_q : \overline{C}_q.\Lambda \overline{a}_j.\lambda \overline{\delta}_y : [\sigma/a]\overline{C}_y.e$$

if we can show the following:

$$\mathbf{unambig}(\forall\overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau) \tag{C.44}$$

$$\Gamma_C;\bullet \vdash^M_C \forall\overline{b}_k.\overline{C}_q \Rightarrow TC\,\tau \rightsquigarrow \forall\overline{b}_k.\overline{C}_q \Rightarrow TC\,\sigma \tag{C.45}$$

$$(m : \overline{C}'_i \Rightarrow TC\,a : \forall\overline{a}_j.\overline{C}_y \Rightarrow \tau_1) \in \Gamma_C \tag{C.46}$$

$$P;\Gamma_C;\bullet,\overline{b}_k,\overline{\delta}_q : \overline{C}_q,\overline{a}_j,\overline{\delta}_y : [\tau/a]\overline{C}_y \vdash^M_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \tag{C.47}$$

$$\Gamma_C;\bullet,a \vdash^M_{ty} \forall\overline{a}_j.\overline{C}_y \Rightarrow \tau_1 \rightsquigarrow \sigma' \tag{C.48}$$

$$D \notin \mathbf{dom}(P) \tag{C.49}$$

$$(D' : \forall\overline{b}'_k.\overline{C}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P \tag{C.50}$$

$$\mathbf{where}[\overline{\tau}_j/\overline{b}_j]\tau = [\overline{\tau}'_k/\overline{b}'_k]\tau'' \tag{C.51}$$

$$\vdash^M_{ctx} P;\Gamma_C;\bullet \rightsquigarrow \Sigma;\Gamma_C;\bullet \tag{C.52}$$

Goal C.52 is exactly Equation C.43, which we already have. Goals C.44 and C.46 follow directly from the premise of rule sINST-INST. The premise also tells us that D is freshly generated, which satisfies Goal C.49. Similarly Goals C.47, C.50 and C.51 can be proven directly from the rule premise.

From the premise, we know:

$$\Gamma_C;\bullet,\overline{b}_k \vdash^M_{ty} \tau \rightsquigarrow \sigma \tag{C.53}$$

$$\overline{\Gamma_C;\bullet,\overline{b}_k \vdash^M_C C_q \rightsquigarrow C_q}^{\,q} \tag{C.54}$$

Goal C.45 follows directly from the definition of well-formedness of constraints and types. Goals C.47 and C.48 remain to be proven.

From the rule premise, we know that

$$(m : \overline{C}'_i \Rightarrow TC\,a : \forall\overline{a}_j.\overline{C}_y \Rightarrow \tau_1) \in \Gamma_C \tag{C.55}$$

From the definition of well-formedness of the source context, we know that:

$$\Gamma_{C1};\bullet,a \vdash^M_{ty} \forall\overline{a}_j.\overline{C}_y \Rightarrow \tau_1 \rightsquigarrow \sigma'$$

$$\Gamma_C = \Gamma_{C1},\Gamma_{C2}$$

By weakening of class environment (Lemma 62), we can prove Goal C.48.

□

Theorem 23 (Typing Preservation - Classes)**.**
If $\Gamma_C \vdash^{M}_{cls} cls : {\Gamma_C}'$*, and* $\vdash^{M}_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \Sigma; \Gamma_C; \bullet$*, then we have* $\vdash^{M}_{ctx} P; \Gamma_C, {\Gamma_C}'; \bullet \rightsquigarrow \Sigma; \Gamma_C, {\Gamma_C}'; \bullet$*.*

Proof. We restate the rule for class declaration typing for reference:

sCls-cls

$$\frac{\begin{array}{c} m \notin \mathbf{dom}(\Gamma_C) \\ \mathbf{closure}(\Gamma_C; \overline{C}_m) = \overline{C}_n \\ \Gamma_C; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau \rightsquigarrow \sigma \\ \mathbf{unambig}(\forall \overline{a}_j, a . \overline{C}_n \Rightarrow \tau) \\ \overline{\Gamma_C; \bullet, a \vdash^{M}_{C} C_i \rightsquigarrow C_i}^{\,i<q} \\ \nexists TC' : (m : \overline{C}'_w \Rightarrow TC'\ b : \sigma') \in \Gamma_C \\ \nexists m' : (m' : \overline{C}'_w \Rightarrow TC\ a : \sigma') \in \Gamma_C \\ {\Gamma_C}' = m : \overline{C}_q \Rightarrow TC\,a : \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau \end{array}}{\Gamma_C \vdash^{M}_{cls} \mathbf{class}\ \overline{C}_q \Rightarrow TC\,a\,\mathbf{where}\ \{m : \forall \overline{a}_j . \overline{C}_m \Rightarrow \tau\} : {\Gamma_C}'}$$

By case analysis, we know that ${\Gamma_C}'$ is of the form

$$\bullet, m : \overline{C}_q \Rightarrow TC\,a : \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau$$

The goal to be proven is the following:

$$\vdash^{M}_{ctx} P; \Gamma_C, m : \overline{C}_q \Rightarrow TC\,a : \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau; \bullet \rightsquigarrow \Sigma; \Gamma_C, m : TC\,a : \sigma; \bullet \tag{C.56}$$

We can derive from rule sCtx-clsEnv that

$$\vdash^{M}_{ctx} \bullet; \Gamma_C, m : \overline{C}_q \Rightarrow TC\,a : \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau; \bullet \rightsquigarrow \bullet; \Gamma_C, m : TC\,a : \sigma; \bullet \tag{C.57}$$

assuming we can show that:

$$\Gamma_C; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_n \Rightarrow \tau \rightsquigarrow \sigma \tag{C.58}$$

$$\overline{a}_j, a = \mathbf{fv}(\tau) \tag{C.59}$$

$$\overline{\Gamma_C; \bullet, a \vdash^{M}_{Q} TC_i\,a \rightsquigarrow Q_i}^{\,i} \tag{C.60}$$

$$m \notin \mathbf{dom}(\Gamma_C) \tag{C.61}$$

$$TC\,b \notin \mathbf{dom}(\Gamma_C) \tag{C.62}$$

$$\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet; \Gamma_C; \bullet \tag{C.63}$$

Goals C.58 till C.62 follow directly from the premises and from the hypothesis. Goal C.63 follows by repeated inversion on the second hypothesis. Finally, Goal C.56 follows from Equation C.57 by the definition of environment well-formedness and the second hypothesis.

□

Theorem 24 (Typing Preservation - Programs)**.** *If* $P;\Gamma_C \vdash^{M}_{pgm} pgm : \tau; P';\Gamma_C{}' \rightsquigarrow e$*, and* $\vdash^{M}_{ctx} P;\Gamma_C;\bullet \rightsquigarrow \Sigma;\Gamma_C;\bullet$*, and* $\Gamma_C,\Gamma_C{}';\bullet \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$ *then we have* $\vdash^{M}_{ctx} P,P';\Gamma_C,\Gamma_C{}';\bullet \rightsquigarrow \Sigma,\Sigma';\Gamma_C,\Gamma_C{}';\bullet$*, and we have* $\Sigma,\Sigma';\Gamma_C,\Gamma_C{}';\bullet \vdash_{tm} e : \sigma$*.*

Proof. By structural induction on the typing derivation.

$$\boxed{\textbf{rule}\ \textsc{sPgmCls}}\qquad \textsc{sPgm-cls}\ \frac{\begin{array}{c}\Gamma_C \vdash^{M}_{cls} cls : \Gamma_C{}' \\ P;\Gamma_C,\Gamma_C{}' \vdash^{M}_{pgm} pgm : \tau; P';\Gamma_C{}'' \rightsquigarrow e\end{array}}{P;\Gamma_C \vdash^{M}_{pgm} cls; pgm : \tau; P';\Gamma_C{}',\Gamma_C{}'' \rightsquigarrow e}$$

We know that

$$\vdash^{M}_{ctx} P;\Gamma_C;\bullet \rightsquigarrow \Sigma;\Gamma_C;\bullet$$

By inversion it follows that:

$$\vdash^{M}_{ctx} \bullet;\Gamma_C;\bullet \rightsquigarrow \bullet;\Gamma_C;\bullet$$

By Typing Preservation - Classes (Theorem 23), we know

$$\vdash^{M}_{ctx} \bullet;\Gamma_C,\Gamma_C{}';\bullet \rightsquigarrow \bullet;\Gamma_C,\Gamma_C{}';\bullet$$

Through weakening (Lemma 65), we know that

$$\vdash^{M}_{ctx} P;\Gamma_C,\Gamma_C{}';\bullet \rightsquigarrow \Sigma;\Gamma_C,\Gamma_C{}';\bullet$$

The goal follows directly from the induction hypothesis.

$$\boxed{\textbf{rule}\ \textsc{sPgm-Inst}}\qquad \textsc{sPgm-inst}\ \frac{\begin{array}{c}P;\Gamma_C \vdash^{M}_{inst} inst : P' \\ P,P';\Gamma_C \vdash^{M}_{pgm} pgm : \tau; P'';\Gamma_C{}' \rightsquigarrow e\end{array}}{P;\Gamma_C \vdash^{M}_{pgm} inst; pgm : \tau; P',P'';\Gamma_C{}' \rightsquigarrow e}$$

We know that

$$\vdash^{M}_{ctx} P;\Gamma_C;\bullet \rightsquigarrow \Sigma;\Gamma_C;\bullet$$

By Typing Preservation - Instance (Theorem 22), we know that

$$\vdash^{M}_{ctx} P,P';\Gamma_C;\bullet \rightsquigarrow \Sigma,\Sigma';\Gamma_C;\bullet \tag{C.64}$$

The goal follows directly from the induction hypothesis.

$$\boxed{\textbf{rule } \textsc{sPgm-expr}} \quad \frac{P; \Gamma_C; \bullet \vdash^{M}_{tm} e \Rightarrow \tau \rightsquigarrow e}{P; \Gamma_C \vdash^{M}_{pgm} e : \tau; \bullet; \bullet \rightsquigarrow e}\ \textsc{sPgm-expr}$$

Follows directly from Typing Preservation - Expressions (Theorem 21). □

Theorem 25 (Typing Preservation - Types and Constraints)**.**

- *If* $\Gamma_C; \Gamma \vdash^{M}_{ty} \sigma \rightsquigarrow \sigma$*, and* $\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma$*, then* $\Gamma_C; \Gamma \vdash_{ty} \sigma$*.*
- *If* $\Gamma_C; \Gamma \vdash^{M}_{Q} Q \rightsquigarrow Q$*, and* $\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma$*, then* $\Gamma_C; \Gamma \vdash_{Q} Q$ *.*
- *If* $\Gamma_C; \Gamma \vdash^{M}_{C} C \rightsquigarrow C$*, and* $\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma$*, then* $\Gamma_C; \Gamma \vdash_{C} C$ *.*

Proof. By induction on the lexicographic order of the tuple (size of Γ_C, the derivation height of type well-formedness and the constraint well-formedness). In each mutual dependency (with the exception of going from part 3 to part 2), the size of the tuple is decreasing, so we know that the induction is well-founded.

Part 1

$$\boxed{\textbf{rule } \textsc{sTy-bool}} \quad \frac{}{\Gamma_C; \Gamma \vdash^{M}_{ty} Bool \rightsquigarrow Bool}\ \textsc{sTy-bool}$$

Follows directly by rule iTy-bool.

$$\boxed{\textbf{rule } \textsc{sTy-var}} \quad \frac{a \in \Gamma}{\Gamma_C; \Gamma \vdash^{M}_{ty} a \rightsquigarrow a}\ \textsc{sTy-var}$$

It is easy to verify that for any environment for which $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ holds, $a \in \Gamma$ implies $a \in \Gamma$. Therefore, the goal follows from rule iTy-var.

$$\boxed{\textbf{rule } \textsc{sTy-arrow}} \quad \frac{\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightsquigarrow \sigma_1 \qquad \Gamma_C; \Gamma \vdash^{M}_{ty} \tau_2 \rightsquigarrow \sigma_2}{\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightarrow \tau_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}\ \textsc{sTy-arrow}$$

By induction hypothesis, we get

$$\Gamma_C; \Gamma \vdash_{ty} \sigma_1$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma_2$$

The goal follows directly from rule iTy-Arrow.

$$\boxed{\textbf{rule } \textsc{sTy-qual}} \quad \dfrac{\begin{array}{c}\textsc{sTy-qual}\\ \Gamma_C ; \Gamma \vdash_C^M C \leadsto C \\ \Gamma_C ; \Gamma \vdash_{ty}^M \rho \leadsto \sigma\end{array}}{\Gamma_C ; \Gamma \vdash_{ty}^M C \Rightarrow \rho \leadsto C \Rightarrow \sigma}$$

By induction hypothesis, we get

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma$$

By Part 2 of this lemma, we get

$$\Gamma_C ; \Gamma \vdash_C C$$

The goal follows directly from rule iTy-Qual.

$$\boxed{\textbf{rule } \textsc{sTy-scheme}} \quad \dfrac{\begin{array}{c}\textsc{sTy-scheme}\\ a \notin \Gamma \\ \Gamma_C ; \Gamma , a \vdash_{ty}^M \sigma \leadsto \sigma\end{array}}{\Gamma_C ; \Gamma \vdash_{ty}^M \forall a . \sigma \leadsto \forall a . \sigma}$$

Given $\vdash_{ctx}^M \bullet ; \Gamma_C ; \Gamma \leadsto \bullet ; \Gamma_C ; \Gamma$, by rule sCtx-tyEnvTy, we know that $\vdash_{ctx}^M \bullet ; \Gamma_C ; \Gamma , a \leadsto \bullet ; \Gamma_C ; \Gamma , a$ By induction hypothesis, we get

$$\Gamma_C ; \Gamma , a \vdash_{ty} \sigma$$

The goal follows directly by rule iTy-scheme.

Part 2

$$\boxed{\textbf{rule } \textsc{sQ-TC}} \quad \dfrac{\begin{array}{c}\textsc{sQ-TC}\\ \Gamma_C ; \Gamma \vdash_{ty}^M \tau \leadsto \sigma \\ \Gamma_C = \Gamma_{C1} , m : \overline{C}_i \Rightarrow TC\, a : \sigma , \Gamma_{C2} \\ \Gamma_{C1} ; \bullet , a \vdash_{ty}^M \sigma \leadsto \sigma'\end{array}}{\Gamma_C ; \Gamma \vdash_Q^M TC\, \tau \leadsto TC\, \sigma}$$

By Part 1 of this lemma, we know that

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma \tag{C.65}$$

It is easy to verify that given any environment for which

$$\vdash_{ctx}^M \bullet ; \Gamma_C ; \Gamma \leadsto \bullet ; \Gamma_C ; \Gamma$$

$$\Gamma_C = \Gamma_{C1} , m : \overline{C}_i \Rightarrow TC\, a : \sigma , \Gamma_{C2} \tag{C.66}$$

then

$$\Gamma_C = \Gamma_{C1} , m : TC\, a : \sigma' , \Gamma_{C2}$$

$$\vdash_{ctx}^M \bullet ; \Gamma_{C1} ; \bullet \leadsto \bullet ; \Gamma_{C1} ; \bullet$$

$$\Gamma_{C1} ; \bullet , a \vdash_{ty}^M \sigma \leadsto \sigma'$$

By rule sCtx-TyEnvTy, we get

$$\vdash^{M}_{ctx} \bullet; \Gamma_{C1}; \bullet, a \rightsquigarrow \bullet; \Gamma_{C1}; \bullet, a$$

The size of Γ_{C1} is trivially smaller than Γ_C. So by induction hypothesis, we have

$$\Gamma_C; \bullet, a \vdash_{ty} \sigma' \tag{C.67}$$

The goal follows directly from rule sQ-TC, and Equations C.65, C.66, C.67.

Part 3

$$\boxed{\textbf{rule } \textsc{sC-forall}} \qquad \frac{\begin{array}{c}\Gamma_C; \Gamma, a \vdash^{M}_{C} C \rightsquigarrow C \\ a \notin \Gamma\end{array}}{\Gamma_C; \Gamma \vdash^{M}_{C} \forall a.C \rightsquigarrow \forall a.C}\ \textsc{sC-forall}$$

By rule sCtx-tyEnvTy, we know that $\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma, a \rightsquigarrow \bullet; \Gamma_C; \Gamma, a$. By the induction hypothesis, we then get

$$\Gamma_C; \Gamma, a \vdash_C C$$

The goal follows by rule iC-forall.

$$\boxed{\textbf{rule } \textsc{sC-arrow}} \qquad \frac{\begin{array}{c}\Gamma_C; \Gamma \vdash^{M}_{C} C_1 \rightsquigarrow C_1 \\ \Gamma_C; \Gamma \vdash^{M}_{C} C_2 \rightsquigarrow C_2\end{array}}{\Gamma_C; \Gamma \vdash^{M}_{C} C_1 \Rightarrow C_2 \rightsquigarrow C_1 \Rightarrow C_2}\ \textsc{sC-arrow}$$

By the induction hypothesis, we get

$$\Gamma_C; \Gamma \vdash_C C_1$$

$$\Gamma_C; \Gamma \vdash_C C_2$$

The goal follows by rule iC-arrow.

$$\boxed{\textbf{rule } \textsc{sC-classConstr}} \qquad \frac{\Gamma_C; \Gamma \vdash^{M}_{Q} Q \rightsquigarrow Q}{\Gamma_C; \Gamma \vdash^{M}_{C} Q \rightsquigarrow Q}\ \textsc{sC-classConstr}$$

The goal follows directly by Part 2 of this lemma.

□

Theorem 26 (Typing Preservation - Constraint Entailment)**.** *If* $P; \Gamma_C; \Gamma \vDash^{M} [C] \rightsquigarrow d$, *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$, *and* $\Gamma_C; \Gamma \vdash^{M}_{C} C \rightsquigarrow C$, *then* $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$.

Proof. By induction on the constraint resolution derivation tree. This theorem is mutually proven with Theorems 21, 27, and 28, as well as Lemma 79 (Figure C.1). Note that at the dependency from Theorem 28 to 21, and from Lemma 79 to Theorem 21, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, this theorem and Theorem 27 perform induction over a (finite) derivation. Consequently, the size of P is strictly decreasing in every possible cycle. The induction thus remains well-founded.

$$\boxed{\textbf{rule } \textsc{sEntail-arrow}} \quad \overset{\textsc{sEntail-arrow}}{\dfrac{\begin{array}{c} P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash^M [C_2] \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash_C^M C_1 \rightsquigarrow C_1 \end{array}}{P; \Gamma_C; \Gamma \vDash^M [C_1 \Rightarrow C_2] \rightsquigarrow \lambda \delta_1 : C_1 . d}}$$

Case analysis on the third hypothesis (rule sC-arrow) gives us

$$\Gamma_C; \Gamma \vdash_C^M C_1 \rightsquigarrow C_1$$

$$\Gamma_C; \Gamma \vdash_C^M C_2 \rightsquigarrow C_2$$

Applying Lemma 66 in combination with rule sCtx-tyEnvD, results in

$$\vdash_{ctx}^M P; \Gamma_C; \Gamma, \delta : C_1 \rightsquigarrow \Sigma; \Gamma_C; \Gamma, \delta : C_1$$

Using these results, we can apply the induction hypothesis to obtain

$$\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2$$

The goal now follows by rule D-dabs (the well-formedness of C_1 is derived through Theorem 25).

$$\boxed{\textbf{rule } \textsc{sEntail-forall}} \quad \overset{\textsc{sEntail-forall}}{\dfrac{P; \Gamma_C; \Gamma, a \vDash^M [C] \rightsquigarrow d}{P; \Gamma_C; \Gamma \vDash^M [\forall a . C] \rightsquigarrow \Lambda a . d}}$$

Case analysis on the third hypothesis (rule sC-forall) gives us

$$\Gamma_C; \Gamma, a \vdash_C^M C \rightsquigarrow C$$

Applying Lemma 66 in combination with rule sCtx-tyEnvTy, results in

$$\vdash_{ctx}^M P; \Gamma_C; \Gamma, a \rightsquigarrow \Sigma; \Gamma_C; \Gamma, a$$

Using these results, we can apply the induction hypothesis to obtain

$$\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C$$

The goal now follows by rule D-tyabs.

$$\boxed{\textbf{rule } \textsc{sEntail-inst}} \quad \overset{\textsc{sEntail-inst}}{\dfrac{\begin{array}{c} P = P_1, (D : C).m \mapsto \Gamma' : e, P_2 \\ \vdash_{ctx}^M P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \\ P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash D : C] \vDash^M Q \rightsquigarrow \bullet \vdash d \end{array}}{P; \Gamma_C; \Gamma \vDash^M [Q] \rightsquigarrow d}}$$

As we know that $(D : C).m \mapsto \Gamma' : e \in P$ and $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \leadsto \Sigma;\Gamma_C;\Gamma$, if follows from rule sCtx-pgmInst that

$$\Gamma_C;\bullet \vdash^{M}_{C} C \leadsto C$$

By Weakening Lemma 64:

$$\Gamma_C;\Gamma \vdash^{M}_{C} C \leadsto C$$

It follows from Lemma 79 that

$$\Sigma;\Gamma_C;\Gamma \vdash_d D : C$$

Using these results, the goal follows directly from Theorem 27.

$$\boxed{\textbf{rule } \textsc{sEntail-local}} \qquad \frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash^{M}_{ctx} P;\Gamma_C;\Gamma \leadsto \Sigma;\Gamma_C;\Gamma \\ P;\Gamma_C;\Gamma;[\bullet;\bullet \vdash \delta : C] \vDash^{M} Q \leadsto \bullet \vdash d\end{array}}{P;\Gamma_C;\Gamma \vDash^{M} [Q] \leadsto d}\ \textsc{sEntail-local}$$

As we know that $(\delta : C) \in \Gamma$ and $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \leadsto \Sigma;\Gamma_C;\Gamma$, we can easily derive from rule sCtx-tyEnvD and Weakening Lemma 64 that

$$\Gamma_C;\Gamma \vdash^{M}_{C} C \leadsto C$$

It follows from Lemma 77 that

$$\Sigma;\Gamma_C;\Gamma \vdash_d \delta : C$$

Using these results, the goal follows directly from Theorem 27. □

Theorem 27 (Typing Preservation - Constraint Matching)**.**
If $P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta} : \overline{C} \vdash d_1 : C_1] \vDash^{M} Q_2 \leadsto \overline{\tau} \vdash d_2$,
and $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \leadsto \Sigma;\Gamma_C;\Gamma$,
and $\Gamma_C;\Gamma \vdash^{M}_{Q} Q_2 \leadsto Q_2$, *and* $\Gamma_C;\Gamma,\overline{a} \vdash^{M}_{C} C_1 \leadsto C_1$,
and $\overline{\Gamma_C;\Gamma,\overline{a} \vdash^{M}_{C} C_i \leadsto C_i}^{\,i}$ *and* $\overline{\Gamma_C;\Gamma \vdash^{M}_{ty} \tau_j \leadsto \sigma_j}^{\,j}$
and $\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta} : \overline{C} \vdash_d d_1 : C_1$, *then* $\Sigma;\Gamma_C;\Gamma,\overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d d_2 : Q_2$.

Proof. By induction on the constraint matching derivation. This theorem is mutually proven with Theorems 21, 26, and 28, as well as Lemma 79 (Figure C.1). Note that at the dependency from Theorem 28 to 21, and from Lemma 79 to Theorem 21, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, this theorem and Theorem 26 perform induction over a (finite) derivation. Consequently, the size of P is strictly decreasing in every possible cycle. The induction thus remains well-founded.

$$\boxed{\textbf{rule } \textsc{sMatch-arrow}} \qquad \overset{\textsc{sMatch-arrow}}{\frac{\begin{array}{c} P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C},\delta_1 : C_1 \vdash d_0\,\delta_1 : C_2] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash d_2 \\ P;\Gamma_C;\Gamma \vDash^M [[\overline{\tau}/\overline{a}]C_1] \rightsquigarrow d_1 \end{array}}{P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C} \vdash d_0 : C_1 \Rightarrow C_2] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash [d_1/\delta_1]d_2}}$$

Case analysis on the fourth hypothesis $\Gamma_C;\Gamma,\overline{a} \vdash_C^M C_2' \Rightarrow C_2'' \rightsquigarrow C_2' \Rightarrow C_2''$ (rule sC-arrow) gives us

$$\Gamma_C;\Gamma,\overline{a} \vdash_C^M C_2' \rightsquigarrow C_2'$$

$$\Gamma_C;\Gamma,\overline{a} \vdash_C^M C_2'' \rightsquigarrow C_2''$$

Note that from the 7th hypothesis we have that

$$\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C} \vdash_d d_0 : C_2' \Rightarrow C_2''$$

By rule D-var it is easy to see that

$$\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C},\delta_1 : C_2' \vdash_d \delta_1 : C_2'$$

Combining these results (using Weakening Lemma 96) with rule D-dapp gives us

$$\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C},\delta_1 : C_2' \vdash_d d_0\,\delta_1 : C_2''$$

By the induction hypothesis we get that

$$\Sigma;\Gamma_C;\Gamma,\overline{\delta}:[\overline{\sigma}/\overline{a}]\overline{C},\delta_1 : [\overline{\sigma}/\overline{a}]C_2' \vdash_d d_2 : Q_2 \tag{C.68}$$

By Lemma 61, in combination with the 6th hypothesis

$$\Gamma_C;\Gamma \vdash_C^M [\overline{\tau}/\overline{a}]C_2' \rightsquigarrow [\overline{\sigma}/\overline{a}]C_2'$$

We can then apply Theorem 26 on the 2nd rule hypothesis:

$$\Sigma;\Gamma_C;\Gamma \vdash_d d_1 : [\overline{\sigma}/\overline{a}]C_2'$$

The goal now follows by applying Substitution Lemma 91 in Equation C.68, using this result.

$$\boxed{\textbf{rule } \textsc{sMatch-forall}} \qquad \overset{\textsc{sMatch-forall}}{\frac{P;\Gamma_C;\Gamma;[\overline{a},a;\overline{\delta}:\overline{C} \vdash d_0\,a : C] \vDash^M Q \rightsquigarrow \overline{\tau},\tau \vdash d_1}{P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C} \vdash d_0 : \forall a.C] \vDash^M Q \rightsquigarrow \overline{\tau} \vdash d_1}}$$

Case analysis on the fourth hypothesis $\Gamma_C;\Gamma,\overline{a} \vdash_C^M \forall a.C_2' \rightsquigarrow \forall a.C_2'$ (rule sC-forall) gives us

$$\Gamma_C;\Gamma,\overline{a},a \vdash_C^M C_2' \rightsquigarrow C_2'$$

By rule D-tyApp and the (weakened) 7th hypothesis, together with rule iTyVar:

$$\Sigma; \Gamma_C; \Gamma, \overline{a}, a \vdash_d d_0\, a : C'_2$$

By the induction hypothesis we get

$$\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}][\sigma/a]\overline{C} \vdash_d d_1 : Q_2$$

However, the 5th hypothesis tells us that $\overline{C}$ is well formed under $\Gamma, \overline{a}$. The goal thus follows directly from this result.

$\boxed{\textbf{rule}\ \text{sMatch-classConstr}}$

sMatch-classConstr

$$\frac{\begin{array}{c}\tau_1 = [\overline{\tau}/\overline{a}]\tau_0 \\ \overline{\Gamma_C; \Gamma \vdash^M_{ty} \tau_i \rightsquigarrow \sigma_i}^{\,i}\end{array}}{P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : TC\,\tau_0] \models^M TC\,\tau_1 \rightsquigarrow \overline{\tau} \vdash [\overline{\sigma}/\overline{a}]d_0}$$

As we know from the 7th hypothesis that

$$\Sigma; \Gamma_C; \Gamma, \overline{a}, \overline{\delta} : \overline{C} \vdash_d d_0 : TC\,\sigma_0$$

the goal

$$\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d [\overline{\sigma}/\overline{a}]d_0 : TC\,[\overline{\sigma}/\overline{a}]\sigma_0$$

follows directly from Lemma 93. □

> **Theorem 28** (Typing Preservation - Environment Well-Formedness)**.**
> *If* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$*, then* $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$*.*

Proof. By induction on the well-formedness derivation. This theorem is mutually proven with Theorems 21, 26, and 27, as well as Lemma 79 (Figure C.1). Note that at the dependency from Theorem 28 to 21, and from Lemma 79 to Theorem 21, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorem 26 and 27 perform induction over a (finite) derivation. Consequently, the size of P is strictly decreasing in every possible cycle. The induction thus remains well-founded.

sCtx-empty

$\boxed{\textbf{rule}\ \text{sCtx-empty}}$ $$\frac{}{\vdash^M_{ctx} \bullet; \bullet; \bullet \rightsquigarrow \bullet; \bullet; \bullet}$$

Follows directly by rule iCtx-empty.

sCtx-tyEnvTm

$\boxed{\textbf{rule}\ \text{sCtx-tyEnvTm}}$ $$\frac{\begin{array}{c}\Gamma_C; \Gamma \vdash^M_{ty} \sigma \rightsquigarrow \sigma \\ x \notin \mathbf{dom}(\Gamma) \\ \vdash^M_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma\end{array}}{\vdash^M_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma \rightsquigarrow \bullet; \Gamma_C; \Gamma, x : \sigma}$$

By induction hypothesis, we know

$$\vdash_{ctx} \bullet; \Gamma_C; \Gamma$$

By Typing Preservation - Types and Class Constraints (Theorem 25), we know

$$\Gamma_C; \Gamma \vdash_{ty} \sigma$$

Since $x \notin \mathbf{dom}(\Gamma)$, it is easy to verify that $x \notin \mathbf{dom}(\Gamma)$. Therefore the goal follows directly by rule $\textsc{iCtx-tyEnvTm}$.

$$\boxed{\textbf{rule}\ \textsc{sCtx-tyEnvTy}} \quad \frac{\begin{array}{c}\textsc{sCtx-tyEnvTy} \\ a \notin \Gamma \\ \vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma\end{array}}{\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma, a \rightsquigarrow \bullet; \Gamma_C; \Gamma, a}$$

By induction hypothesis, we know

$$\vdash_{ctx} \bullet; \Gamma_C; \Gamma$$

Since we know $a \notin \Gamma$, it is easy to verify that $a \notin \Gamma$. Therefore the goal follows directly by rule $\textsc{iCtx-tyEnvTy}$.

$$\boxed{\textbf{rule}\ \textsc{sCtx-tyEnvD}} \quad \frac{\begin{array}{c}\textsc{sCtx-tyEnvD} \\ \Gamma_C; \Gamma \vdash^{M}_{C} C \rightsquigarrow C \\ \delta \notin \mathbf{dom}(\Gamma) \\ \vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma\end{array}}{\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma, \delta : C \rightsquigarrow \bullet; \Gamma_C; \Gamma, \delta : C}$$

By induction hypothesis, we know

$$\vdash_{ctx} \bullet; \Gamma_C; \Gamma$$

By Typing Preservation - Types and Class Constraints (Theorem 25), we know

$$\Gamma_C; \Gamma \vdash_C C$$

Since $\delta \notin \mathbf{dom}(\Gamma)$, it is easy to verify that $\delta \notin \mathbf{dom}(\Gamma)$. Therefore the goal follows directly by rule $\textsc{iCtx-tyEnvD}$.

$\boxed{\textbf{rule}\ \textsc{sCtx-clsEnv}}$

$\textsc{sCtx-clsEnv}$

$$\frac{\begin{array}{c}\Gamma_C; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \rightsquigarrow \sigma \\ \overline{a}_j, a = \mathbf{fv}(\tau) \\ \overline{\Gamma_C; \bullet, a \vdash^{M}_{C} C_i \rightsquigarrow Q_i}^{\,i} \\ m \notin \mathbf{dom}(\Gamma_C) \\ TC\, b \notin \mathbf{dom}(\Gamma_C) \\ \vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet; \Gamma_C; \bullet\end{array}}{\vdash^{M}_{ctx} \bullet; \Gamma_C, m : \overline{C}_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau; \bullet \rightsquigarrow \bullet; \Gamma_C, m : TC\, a : \sigma; \bullet}$$

By induction hypothesis, we know

$$\vdash_{ctx} \bullet; \Gamma_C; \bullet$$

By rule sCtx-tyEnvTy, we know that

$$\vdash_{ctx}^{M} \bullet; \Gamma_C; \bullet, a \rightsquigarrow \bullet; \Gamma_C; \bullet, a$$

Then by Typing Preservation - Types and Class Constraints (Theorem 25), we know

$$\Gamma_C; \bullet, a \vdash_{ty} \sigma$$

It is easy to verify that given $m \notin \mathbf{dom}(\Gamma_C)$, $TC\, b \notin \mathbf{dom}(\Gamma_C)$, we can derive $m \notin \mathbf{dom}(\Gamma_C)$, $TC\, b \notin \mathbf{dom}(\Gamma_C)$.

The goal follows directly by rule iCtx-ClsEnv.

$\boxed{\textbf{rule } \textsc{sCtx-pgmInst}}$

sCtx-pgmInst

$$\frac{\begin{array}{c}\mathbf{unambig}(\forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\tau) \\ \Gamma_C; \bullet \vdash_{C}^{M} \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\tau \rightsquigarrow \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\sigma \\ (m : \overline{C}'_m \Rightarrow TC\,a : \forall \overline{a}_k.\overline{C}_y \Rightarrow \tau') \in \Gamma_C \\ P; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}_y \vdash_{tm}^{M} e \Leftarrow [\tau/a]\tau' \rightsquigarrow e \\ \Gamma_C; \bullet, a \vdash_{ty}^{M} \forall \overline{a}_k.\overline{C}_y \Rightarrow \tau' \rightsquigarrow \forall \overline{a}_k.\overline{C}_y \Rightarrow \sigma' \\ D \notin \mathbf{dom}(P) \\ (D' : \forall \overline{b}'_k.\overline{C}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P \\ \mathbf{where}[\overline{\tau}_j/\overline{b}_j]\tau = [\overline{\tau}'_k/\overline{b}'_k]\tau'' \\ \vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \\ \Sigma' = \Sigma, (D : \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\sigma).m \mapsto \Lambda \overline{b}_j.\lambda \overline{\delta}_i : \overline{C}_i.\Lambda \overline{a}_k.\lambda \overline{\delta}_y : [\sigma/a]\overline{C}_y.e\end{array}}{\vdash_{ctx}^{M} P, (D : \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}_y : e; \Gamma_C; \Gamma \rightsquigarrow \Sigma'; \Gamma_C; \Gamma}$$

By induction hypothesis, we know that

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.69}$$

Also, since we know that

$$\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$$

by applying inversion, we get:

$$\vdash_{ctx}^{M} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma \tag{C.70}$$

From the premise, we already know

$$\Gamma_C; \bullet \vdash_{C}^{M} \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\tau \rightsquigarrow \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\sigma \tag{C.71}$$

Then by Typing Preservation - Types and Class Constraints (Theorem 25) on Equations C.70 and C.71, we know

$$\Gamma_C; \bullet \vdash_C \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\,\sigma \tag{C.72}$$

From the 3rd rule premise, we know that

$$(m : \overline{C}'_m \Rightarrow TC\,a : \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau') \in \Gamma_C \tag{C.73}$$

By inversion on Equation C.70 (rule sCtx-clsEnv), together with Equation C.73, we get

$$(m : TC\,a : \sigma_1) \in \Gamma_C \tag{C.74}$$

$$\Gamma_{C1}; \bullet, a \vdash_{ty}^{M} \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau' \rightsquigarrow \sigma_1 \tag{C.75}$$

$$\Gamma_C = \Gamma_{C1}, \Gamma_{C2} \tag{C.76}$$

By applying weakening (Lemma 62) on Equation C.75, we have

$$\Gamma_C; \bullet, a \vdash_{ty}^{M} \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau' \rightsquigarrow \sigma_1 \tag{C.77}$$

From the 5th rule premise, we know that

$$\Gamma_C; \bullet, a \vdash_{ty}^{M} \forall \overline{a}_k . \overline{C}_y \Rightarrow \tau' \rightsquigarrow \forall \overline{a}_k . \overline{C}_y \Rightarrow \sigma' \tag{C.78}$$

Because the elaboration of types is deterministic (Lemma 69), combining Equations C.77 and C.78, we know that $\sigma_1 = \forall \overline{a}_k . \overline{C}_y \Rightarrow \sigma'$. By rewriting Equation C.74, we get

$$(m : TC\,a : \forall \overline{a}_k . \overline{C}_y \Rightarrow \sigma') \in \Gamma_C \tag{C.79}$$

By applying Theorem 21 to the 4th rule premise, we get:

$$\Sigma; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\sigma/a]\overline{C}_y \vdash_{tm} e : [\sigma/a]\sigma' \tag{C.80}$$

Lemma 113, applied to this result, gives us:

$$\vdash_{ctx} \Sigma; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\sigma/a]\overline{C}_y \tag{C.81}$$

Furthermore, by applying rule iTm-constrI and rule iTm-forallI to Equation C.80, in combination with inversion on Equation C.81, we get

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} \Lambda \overline{b}_j . \lambda \overline{\delta}_i : \overline{C}_i . \Lambda \overline{a}_k . \lambda \overline{\delta}_y : [\sigma/a]\overline{C}_y . e : \forall \overline{b}_j . \overline{C}_i \Rightarrow \forall \overline{a}_k . [\sigma/a]\overline{C}_y \Rightarrow [\sigma/a]\sigma'$$

which is equivalent to

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} \Lambda \overline{b}_j . \lambda \overline{\delta}_i : \overline{C}_i . \Lambda \overline{a}_k . \lambda \overline{\delta}_y : [\sigma/a]\overline{C}_y . e : \forall \overline{b}_j . \overline{C}_i \Rightarrow [\sigma/a](\forall \overline{a}_k . \overline{C}_y \Rightarrow \sigma') \tag{C.82}$$

Given

$$\mathbf{unambig}(\forall \bar{b}_j.\overline{C}_i \Rightarrow TC\,\tau)$$

$$D \notin \mathbf{dom}(P)$$

$$(D' : \forall \bar{b}'_k.\overline{C}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P$$

$$\mathbf{where}[\overline{\tau}_j/\bar{b}_j]\tau = [\overline{\tau}'_k/\bar{b}'_k]\tau''$$

It is easy to verify that

$$\mathbf{unambig}(\forall \bar{b}_j.\overline{C}_i \Rightarrow TC\,\sigma) \tag{C.83}$$

$$D \notin \mathbf{dom}(\Sigma) \tag{C.84}$$

$$(D' : \forall \overline{a}'_m.\overline{C}''_n \Rightarrow TC\,\sigma'').m' \mapsto e' \notin \Sigma \tag{C.85}$$

$$\mathbf{where}[\overline{\sigma}_j/\overline{a}_j]\sigma = [\overline{\sigma}'_m/\overline{a}'_m]\sigma'' \tag{C.86}$$

The goal follows by combining Equations C.69, C.72, C.79, C.82, C.83, C.84, C.85, C.86, and the rule rule iCtx-MEnv. □

C.5 F_{D} Theorems

C.5.1 Conjectures

We are confident that the following lemmas can be proven using well-known proof techniques.

Lemma 82 (Type Variable Substitution in Types).
If $\Gamma_C; \Gamma_1, a, \Gamma_2 \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1$ *and* $\Gamma_C; \Gamma_1 \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2$ *then* $\Gamma_C; \Gamma_1, [\sigma_2/a]\Gamma_2 \vdash_{ty} [\sigma_2/a]\sigma_1 \rightsquigarrow [\sigma_2/a]\sigma_1$.

Lemma 83 (Type Variable Substitution in Class Constraints).
If $\Gamma_C; \Gamma_1, a, \Gamma_2 \vdash_Q Q \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma_1 \vdash_{ty} \sigma' \rightsquigarrow \sigma'$ *then* $\Gamma_C; \Gamma_1, [\sigma'/a]\Gamma_2 \vdash_Q [\sigma'/a]Q \rightsquigarrow [\sigma'/a]\sigma$.

Lemma 84 (Type Variable Substitution in Constraints).
If $\Gamma_C; \Gamma_1, a, \Gamma_2 \vdash_C C \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma_1 \vdash_{ty} \sigma' \rightsquigarrow \sigma'$ *then* $\Gamma_C; \Gamma_1, [\sigma'/a]\Gamma_2 \vdash_C [\sigma'/a]C \rightsquigarrow [\sigma'/a]\sigma$.

Lemma 85 (Variable Substitution).
If $\Sigma; \Gamma_C; \Gamma_1, x : \sigma_2, \Gamma_2 \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1$ *and* $\Sigma; \Gamma_C; \Gamma_1, \Gamma_2 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2$ *then* $\Sigma; \Gamma_C; \Gamma_1, \Gamma_2 \vdash_{tm} [e_2/x]e_1 : \sigma_1 \rightsquigarrow [e_2/x]e_1$.

Lemma 86 (Reverse Variable Substitution).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} [e_2/x]e_1 : \sigma_1$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_2$ *then* $\Sigma; \Gamma_C; \Gamma, x : \sigma_2 \vdash_{tm} e_1 : \sigma_1$.

Lemma 87 (Dictionary Variable Substitution).
If $\Sigma; \Gamma_C; \Gamma_1, \delta : C, \Gamma_2 \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Sigma; \Gamma_C; \Gamma_1, \Gamma_2 \vdash_d d : C \rightsquigarrow e'$ *then* $\Sigma; \Gamma_C; \Gamma_1, \Gamma_2 \vdash_{tm} [d/\delta]e : \sigma \rightsquigarrow [e'/\delta]e$.

Lemma 88 (Reverse Dictionary Variable Substitution).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} [d/\delta]e : \sigma$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$ *then* $\Sigma; \Gamma_C; \Gamma, \delta : C \vdash_{tm} e : \sigma$.

Lemma 89 (Type Variable Substitution)**.**
If $\Sigma;\Gamma_C;\Gamma_1,a,\Gamma_2 \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Gamma_C;\Gamma_1 \vdash_{ty} \sigma' \rightsquigarrow \sigma'$ *then* $\Sigma;\Gamma_C;\Gamma_1,[\sigma'/a]\Gamma_2 \vdash_{tm} [\sigma'/a]e : [\sigma'/a]\sigma \rightsquigarrow [\sigma'/a]e$*.*

Lemma 90 (Reverse Type Variable Substitution)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_{tm} [\sigma'/a]e : [\sigma'/a]\sigma$ *then* $\Sigma;\Gamma_C;\Gamma,a \vdash_{tm} e : \sigma$*.*

Lemma 91 (Dictionary Variable Substitution in Dictionaries)**.**
If $\Sigma;\Gamma_C;\Gamma_1,\delta : C',\Gamma_2 \vdash_d d : C$ *and* $\Sigma;\Gamma_C;\Gamma_1,\Gamma_2 \vdash_d d : C'$ *then* $\Sigma;\Gamma_C;\Gamma_1,\Gamma_2 \vdash_d [d/\delta]d : C$*.*

Lemma 92 (Reverse Dictionary Variable Substitution in Dictionaries)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_d [d/\delta]d : C$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_d d : C'$ *then* $\Sigma;\Gamma_C;\Gamma,\delta : C' \vdash_d d : C$*.*

Lemma 93 (Type Variable Substitution in Dictionaries)**.**
If $\Sigma;\Gamma_C;\Gamma_1,a,\Gamma_2 \vdash_d d : C$ *and* $\Gamma_C;\Gamma_1 \vdash_{ty} \sigma$ *then* $\Sigma;\Gamma_C;\Gamma_1,[\sigma/a]\Gamma_2 \vdash_d [\sigma/a]d : [\sigma/a]C$*.*

Lemma 94 (Reverse Type Variable Substitution in Dictionaries)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_d [\sigma/a]d : [\sigma/a]C$ *then* $\Sigma;\Gamma_C;\Gamma,a \vdash_d d : C$*.*

Lemma 95 (Expression Well-Typed Method Environment Weakening)**.**
If $\Sigma_1;\Gamma_C;\Gamma \vdash_{tm} e : \sigma$ *and* $\vdash_{ctx} \Sigma_1,\Sigma_2;\Gamma_C;\Gamma$ *then* $\Sigma_1,\Sigma_2;\Gamma_C;\Gamma \vdash_{tm} e : \sigma$*.*

Lemma 96 (Dictionary Well-Formedness Weakening)**.**
If $\Sigma;\Gamma_C;\Gamma_1 \vdash_d d : C$ *and* $\vdash_{ctx} \Sigma;\Gamma_C;\Gamma_1,\Gamma_2$ *then* $\Sigma;\Gamma_C;\Gamma_1,\Gamma_2 \vdash_d d : C$*.*

Lemma 97 (Type Well-Formedness Dictionary Environment Weakening)**.**
If $\vdash_{ctx} \Sigma;\Gamma_{C1};\Gamma_1$ *and* $\Gamma_{C1};\Gamma_1 \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\vdash_{ctx} \Sigma;\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2$*, then* $\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2 \vdash_{ty} \sigma \rightsquigarrow \sigma$*.*

Lemma 98 (Class Constraint Well-Formedness Environment Weakening)**.**
If $\Gamma_{C1};\Gamma_1 \vdash_Q Q \rightsquigarrow \sigma$ *and* $\vdash_{ctx} \Sigma;\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2$ *then* $\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2 \vdash_Q Q \rightsquigarrow \sigma$*.*

Lemma 99 (Constraint Well-Formedness Environment Weakening).
If $\Gamma_{C1};\Gamma_1 \vdash_C C \rightsquigarrow \sigma$ *and* $\vdash_{ctx} \Sigma;\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2$ *then* $\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2 \vdash_C C \rightsquigarrow \sigma$.

Lemma 100 (Logical Equivalence Environment Weakening).
If $\Gamma_{C1};\Gamma_1 \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *and* $\vdash_{ctx} \Sigma_1,\Sigma_1';\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2$ *and* $\vdash_{ctx} \Sigma_2,\Sigma_2';\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2$,
then $\Gamma_{C1},\Gamma_{C2};\Gamma_1,\Gamma_2 \vdash \Sigma_1,\Sigma_1' : e_1 \simeq_{log} \Sigma_2,\Sigma_2' : e_2 : \sigma$.

Lemma 101 (Strong Normalization Relation Method Environment Weakening).
If $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma_1,\Gamma_C}_{R^{SN}}$ *and* $\vdash_{ctx} \Sigma_1,\Sigma_2;\Gamma_C;\Gamma$ *then* $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_{R^{SN}}$.

Lemma 102 (Strong Normalization Relation Substitution Weakening).
If $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}{}_1}$ *and* $\overline{r_j \in Rel[\sigma_j]}^{\,j}$ *then* $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}{}_{1,\overline{a}_j \mapsto (\overline{\sigma}_j,\overline{r}_j)}}$.

C.5.2 Lemmas

Lemma 103 (Environment Well-Formedness Strengthening).
If $\vdash_{ctx} \Sigma;\Gamma_C;\Gamma$ *then* $\vdash_{ctx} \Sigma;\Gamma_C;\bullet$.

Proof. By case analysis on the hypothesis, the last rules used to construct it must be (possibly zero) consecutive applications of rule iCtx-MEnv. Revert those rules, to obtain $\vdash_{ctx} \bullet;\Gamma_C;\Gamma$. By further case analysis (with rule iCtx-tyEnvTm, rule iCtx-tyEnvTy and rule iCtx-tyEnvD), we get $\vdash_{ctx} \bullet;\Gamma_C;\bullet$. The goal follows by consecutively re-applying rule rule iCtx-MEnv with the appropriate premises.

□

Lemma 104 (Variable Strengthening in Dictionaries).
If $\Sigma;\Gamma_C;\Gamma_1,x:\sigma,\Gamma_2 \vdash_d d : C$ *then* $\Sigma;\Gamma_C;\Gamma_1,\Gamma_2 \vdash_d d : C$.

Proof. By straightforward induction on the well-formedness derivation.

□

Lemma 105 (Variable Strengthening in Types)**.**
If $\Gamma_C ; \Gamma_1, x : \sigma, \Gamma_2 \vdash_{ty} \sigma'$ *then* $\Gamma_C ; \Gamma_1, \Gamma_2 \vdash_{ty} \sigma'$*.*

Proof. By straightforward induction on the well-formedness derivation. □

Lemma 106 (Dictionary Variable Strengthening in Constraints)**.**
If $\Gamma_C ; \Gamma_1, \delta : C_1, \Gamma_2 \vdash_C C_2$ *then* $\Gamma_C ; \Gamma_1, \Gamma_2 \vdash_C C_2$ *.*

Proof. By straightforward induction on the well-formedness derivation. □

Lemma 107 (Method Type Well-Formedness)**.**
If $\vdash_{ctx} \Sigma ; \Gamma_C ; \Gamma$ *and* $\Gamma_C = \Gamma_{C1}, m : TC\, a : \sigma, \Gamma_{C2}$ *then there is a* σ *such, that* $\Gamma_{C1} ; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma$*.*

Proof. By straightforward induction on the environment well-formedness derivation. □

Lemma 108 (Method Environment Well-Formedness)**.**
If $\vdash_{ctx} \Sigma ; \Gamma_C ; \Gamma$ *and* $(D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\, \sigma).m \mapsto e \in \Sigma$*, where* $i = 1 \ldots n$*, then there are unique* a*,* σ_m*,* σ_m *and* $\overline{\sigma}_i$*, such that* $(m : TC\, a : \sigma_m) \in \Gamma_C$ *and* $\Gamma_C ; \bullet, a \vdash_{ty} \sigma_m \rightsquigarrow \sigma_m$ *and* $\Sigma ; \Gamma_C ; \bullet \vdash_{tm} e : \forall \overline{a}_j . \overline{C}_i \Rightarrow [\sigma / a] \sigma_m \rightsquigarrow e$ *and* $\overline{\Gamma_C ; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma_i}^{\,i}$*. and* $\Gamma_C ; \bullet, \overline{a}_j \vdash_Q TC\, \sigma \rightsquigarrow [\sigma / a]\{m : \sigma_m\}$ *and* $\Gamma_C ; \bullet, \overline{a}_j \vdash_{ty} \sigma \rightsquigarrow \sigma$*.*

Proof. By straightforward induction on the environment well-formedness derivation. □

Lemma 109 (Determinism of Dictionary Evaluation)**.**
If $d \longrightarrow d_1$ *and* $d \longrightarrow d_2$ *then* $d_1 = d_2$*.*

Proof. By straightforward induction on the evaluation derivation. □

Lemma 110 (Determinism of Evaluation)**.**
If $\Sigma \vdash e \longrightarrow e_1$ *and* $\Sigma \vdash e \longrightarrow e_2$ *then* $e_1 = e_2$*.*

Proof. By straightforward induction on the evaluation derivation. □

Lemma 111 (Preservation of Environment Type Variables from $F_{\mathbf{D}}$ to $F_{\{\}}$)**.**

- *If* $a \in \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $a \in \Gamma$*.*
- *If* $a \notin \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $a \notin \Gamma$*.*

Proof. By straightforward induction on the environment elaboration derivation. □

Lemma 112 (Well-Formedness of $F_{\mathbf{D}}$ Typing Result)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$ *then* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$*.*

Proof. By straightforward induction on the typing derivation. □

Lemma 113 (Context Well-Formedness of $F_{\mathbf{D}}$ Typing)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ *then* $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$*.*

Proof. By straightforward induction on the typing derivation. □

Lemma 114 (Context Well-Formedness of Dictionary Typing)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$ *then* $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$*.*

Proof. By straightforward induction on the dictionary typing derivation. □

C.5.3 Type Safety

Theorem 29 (Dictionary Preservation)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_d d : C$*, and* $d \longrightarrow d'$*, then* $\Sigma;\Gamma_C;\Gamma \vdash_d d' : C$*.*

Proof. By induction on the typing derivation.

$$\boxed{\textbf{rule}\ \text{D-VAR}} \qquad \frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d \delta : C}\ \text{D-VAR}$$

δ cannot be reduced, so impossible case.

$$\boxed{\textbf{rule}\ \text{D-CON}} \qquad \frac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma \\ \overline{\Gamma_C;\bullet,\overline{a}_j \vdash_C C_i}^{\,i} \\ \Sigma_1;\Gamma_C;\bullet,\overline{a}_j,\overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d D : \forall\overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q}\ \text{D-CON}$$

D is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \text{D-DABS}} \qquad \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma,\delta : C_1 \vdash_d d : C_2 \\ \Gamma_C;\Gamma \vdash_C C_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d \lambda\delta : C_1.d : C_1 \Rightarrow C_2}\ \text{D-DABS}$$

$\lambda\delta : C.d$ is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \text{D-DAPP}} \qquad \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \\ \Sigma;\Gamma_C;\Gamma \vdash_d d_2 : C_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d d_1\,d_2 : C_2}\ \text{D-DAPP}$$

By case analysis on the evaluation derivation, two options arise:

- Case rule iDictEval-app.

  By applying the induction hypothesis to the 1st rule premise, we get

  $$\Sigma;\Gamma_C;\Gamma \vdash_d d_1' : C_1 \Rightarrow C_2$$

  The goal then follows directly from rule D-dapp.

- Case rule iDictEval-appAbs.

  We know from the 1st hypothesis that

  $$\Sigma;\Gamma_C;\Gamma \vdash_d \lambda\delta : C_1.d_1' : C_1 \Rightarrow C_2$$

By case analysis (rule D-dAbs) we know

$$\Sigma;\Gamma_C;\Gamma,\delta : C_1 \vdash_d d_1' : C_2$$

The goal follows from Lemma 91.

$$\boxed{\textbf{rule } \text{D-tyAbs}} \qquad \begin{array}{c}\text{D-tyAbs}\\ \dfrac{\Sigma;\Gamma_C;\Gamma,a \vdash_d d : C}{\Sigma;\Gamma_C;\Gamma \vdash_d \Lambda a.d : \forall a.C}\end{array}$$

$\Lambda a.d$ is already a value, so impossible case.

$$\boxed{\textbf{rule } \text{D-tyApp}} \qquad \begin{array}{c}\text{D-tyApp}\\ \dfrac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d : \forall a.C\\ \Gamma_C;\Gamma \vdash_{ty} \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d d\,\sigma : [\sigma/a]C}\end{array}$$

By case analysis on the evaluation derivation, two options arise:

- Case rule iDictEval-tyApp.

  We apply the induction hypothesis to the 1st rule premise to get

  $$\Sigma;\Gamma_C;\Gamma \vdash_d d' : \forall a.C$$

  The goal follows by rule D-tyApp.

- Case rule iDictEval-tyAppAbs.

  We know from the 1st hypothesis that

  $$\Sigma;\Gamma_C;\Gamma \vdash_d \Lambda a.d : \forall a.C$$

  By case analysis (rule D-tyAbs) we know

  $$\Sigma;\Gamma_C;\Gamma,a \vdash_d d : C$$

  The goal follows from Lemma 93.

□

Theorem 30 (Preservation)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma$, *and* $\Sigma \vdash e \longrightarrow e'$, *then* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e' : \sigma$.

Proof. By induction on the typing derivation.

$$\boxed{\textbf{rule } \text{iTm-true}} \qquad \begin{array}{c}\text{iTm-true}\\ \dfrac{\vdash_{ctx} \Sigma;\Gamma_C;\Gamma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}}\end{array}$$

True is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-false}} \qquad \textsc{iTm-false}\ \frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool}}$$

False is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-var}} \qquad \textsc{iTm-var}\ \frac{(x : \sigma) \in \Gamma \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma}$$

x cannot be reduced, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-let}} \qquad \textsc{iTm-let}\ \frac{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \qquad \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \qquad \Gamma_C; \Gamma \vdash_{ty} \sigma_1}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \textbf{let}\ \ x : \sigma_1 = e_1\ \textbf{in}\ \ e_2 : \sigma_2}$$

By inversion on the evaluation (rule iEval-let), we get that

$$e' = [e_1/x]e_2$$

Given

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1$$

$$\Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2$$

The goal follows directly from Lemma 85.

$$\boxed{\textbf{rule}\ \textsc{iTm-method}} \qquad \textsc{iTm-method}\ \frac{\Sigma; \Gamma_C; \Gamma \vdash_d d : \mathit{TC}\,\sigma \qquad (m : \mathit{TC}\,a : \sigma') \in \Gamma_C}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma'}$$

By inversion on the evaluation derivation, two options arise:

- rule iEval-method.

  By applying Theorem 29 to the 1st rule premise, we get

  $$\Sigma; \Gamma_C; \Gamma \vdash_d d' : \mathit{TC}\,\sigma$$

  The goal thus follows directly from rule iTm-method.

- rule iEval-methodVal.

  The goal to be proven thus becomes:

  $$\Sigma; \Gamma_C; \bullet \vdash_{tm} e\,\overline{\sigma}_m\,\overline{d}_n : [\sigma/a]\sigma' \tag{C.87}$$

We know from the 1st rule premise that

$$\Sigma; \Gamma_C; \Gamma \vdash_d D\,\overline{\sigma}_m\,\overline{d}_n : TC\,\sigma \tag{C.88}$$

By inversion on Equation C.88 (rule D-con), we know that

$$(D : \forall \overline{a}_m . \overline{C}_n \Rightarrow TC\,\sigma_1).m \mapsto \Lambda \overline{a}_m . \lambda \overline{\delta}_n : \overline{C}_n . e \in \Sigma \tag{C.89}$$

$$\sigma = [\overline{\sigma}_m / \overline{a}_m]\sigma_1 \tag{C.90}$$

$$\overline{\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_m / \overline{a}_m] C_i}^{\,i<n} \tag{C.91}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_i}^{\,i<m} \tag{C.92}$$

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.93}$$

By combining this result with Equation C.89, through inversion (rule iCtx-MEnv), we know that

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \Lambda \overline{a}_m . \lambda \overline{\delta}_n : \overline{C}_n . e : \forall \overline{a}_m . \overline{C}_n \Rightarrow [\sigma_1 / a]\sigma' \tag{C.94}$$

where $\Sigma = \Sigma_1, (D : \forall \overline{a}_m . \overline{C}_n \Rightarrow TC\,\sigma_1).m \mapsto \Lambda \overline{a}_m . \lambda \overline{\delta}_n : \overline{C}_n . e, \Sigma_2$. By rule iTm-forallE and Equations C.94 and C.92, we have

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e\,\overline{\sigma}_m : [\overline{\sigma}_m / \overline{a}_m]\overline{C}_n \Rightarrow [\overline{\sigma}_m / \overline{a}_m][\sigma_1 / a]\sigma' \tag{C.95}$$

By rule iTm-constrE and Equations C.95 and C.91, we have

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e\,\overline{\sigma}_m\,\overline{d}_n : [\overline{\sigma}_m / \overline{a}_m][\sigma_1 / a]\sigma' \tag{C.96}$$

From the 2nd rule premise we know that

$$(m : TC\,a : \sigma') \in \Gamma_C \tag{C.97}$$

Combining this result with Equation C.93, by inversion (rule iCtx-clsEnv), we know

$$\Gamma_{C1}; \bullet, a \vdash_{ty} \sigma' \tag{C.98}$$

$$\Gamma_C = \Gamma_{C1}, \Gamma_{C2} \tag{C.99}$$

Therefore, $\overline{\sigma}_m$ are not free variables in σ'. Equation C.95 thus simplifies to

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e\,\overline{\sigma}_m\,\overline{d}_n : [[\overline{\sigma}_m / \overline{a}_m]\sigma_1 / a]\sigma' \tag{C.100}$$

By applying Equation C.100 to Lemma 95, in combination with Equation C.93, we get

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e\, \overline{\sigma}_m\, \overline{d}_n : [[\overline{\sigma}_m/\overline{a}_m]\sigma_1/a]\sigma'$$

Goal C.87 follows by combining this result with Equation C.90.

$$\boxed{\textbf{rule}\ \textsc{iTm-arrI}} \qquad \frac{\begin{array}{c}\textsc{iTm-arrI}\\ \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2}$$

$\lambda x : \sigma_1.e$ is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-arrE}} \qquad \frac{\begin{array}{c}\textsc{iTm-arrE}\\ \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \\ \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2}$$

By inversion on the evaluation, we have two possible cases:

- Case rule iEval-app.

  By induction hypothesis, we get

  $$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1' : \sigma_1 \to \sigma_2$$

  By rule iTm-arrE we get

  $$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1'\, e_2 : \sigma_2$$

- Case rule iEval-appAbs.

  From premise, we know

  $$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma.e_1 : \sigma \to \sigma_2$$

  By inversion (rule iTm-arrI)

  $$\Sigma; \Gamma_C; \Gamma, x : \sigma \vdash_{tm} e_1 : \sigma_2$$

  By substitution (Lemma 85) we get

  $$\Sigma; \Gamma_C; \Gamma \vdash_{tm} [e_2/x]e_1 : \sigma_2$$

$$\boxed{\textbf{rule}\ \textsc{iTm-constrI}} \qquad \frac{\begin{array}{c}\textsc{iTm-constrI}\\ \Sigma; \Gamma_C; \Gamma, \delta : C \vdash_{tm} e : \sigma \\ \Gamma_C; \Gamma \vdash_{C} C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda \delta : C.e : C \Rightarrow \sigma}$$

$\lambda \delta : Q.e$ is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-constrE}} \quad \textsc{iTm-constrE}\ \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : C \Rightarrow \sigma \\ \Sigma;\Gamma_C;\Gamma \vdash_d d : C\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\, d : \sigma}$$

Similar to case rule iTm-arrE. The only difference lies in applying Lemma 87.

$$\boxed{\textbf{rule}\ \textsc{iTm-forallI}} \quad \textsc{iTm-forallI}\ \frac{\Sigma;\Gamma_C;\Gamma, a \vdash_{tm} e : \sigma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma}$$

$\Lambda a.e$ is already a value, so impossible case.

$$\boxed{\textbf{rule}\ \textsc{iTm-forallE}} \quad \textsc{iTm-forallE}\ \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \forall a.\sigma' \\ \Gamma_C;\Gamma \vdash_{ty} \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\, \sigma : [\sigma/a]\sigma'}$$

Similar to case rule iTm-arrE. The only difference lies in applying Lemma 89.

□

> **Theorem 31** (Dictionary Progress)**.**
> *If $\Sigma;\Gamma_C;\bullet \vdash_d d : C$, then either d is a dictionary value, or there exists d' such that $d \longrightarrow d'$.*

Proof. By structural induction on the typing derivation.

$$\boxed{\textbf{rule}\ \textsc{D-var}} \quad \textsc{D-var}\ \frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d \delta : C}$$

with $\Gamma = \bullet$.

δ cannot be well-typed in an empty typing context. Impossible case.

$$\boxed{\textbf{rule}\ \textsc{D-con}} \quad \textsc{D-con}\ \frac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\, \sigma_q).m \mapsto \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e, \Sigma_2 \\ (m : TC\, a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma;\Gamma_C;\Gamma \\ \overline{\Gamma_C;\bullet,\overline{a}_j \vdash_C C_i}^{\,i} \\ \Sigma_1;\Gamma_C;\bullet,\overline{a}_j,\overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\, \sigma_q}$$

with $\Gamma = \bullet$.

D is a value.

$$\boxed{\textbf{rule}\ \textsc{D-dabs}} \quad \textsc{D-dabs}\ \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma,\delta : C_1 \vdash_d d : C_2 \\ \Gamma_C;\Gamma \vdash_C C_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d \lambda\delta : C_1.d : C_1 \Rightarrow C_2}$$

with $\Gamma = \bullet$.

$\lambda \delta : C.d$ is a value.

rule D-DAPP

$$\textsc{D-dApp}\quad \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \\ \Sigma;\Gamma_C;\Gamma \vdash_d d_2 : C_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d d_1\, d_2 : C_2}$$

with $\Gamma = \bullet$.

By applying the induction hypothesis on the 1st rule premise, we get either

- d_1 is a value. Case analysis tells us that $d_1 = \lambda\delta : C_1.d_1'$. By rule IDICTEVAL-APPABS, we thus know that
$$(\lambda\delta : C_1.d_1')\, d_2 \longrightarrow [d_2/\delta]d_1'$$
- There exists an d_1', such that $d_1 \longrightarrow d_1'$. By rule IDICTEVAL-APP, we thus get that
$$d_1\, d_2 \longrightarrow d_1'\, d_2$$

rule D-TYABS

$$\textsc{D-tyAbs}\quad \frac{\Sigma;\Gamma_C;\Gamma, a \vdash_d d : C}{\Sigma;\Gamma_C;\Gamma \vdash_d \Lambda a.d : \forall a.C}$$

with $\Gamma = \bullet$.

$\Lambda a.d$ is a value.

rule D-TYAPP

$$\textsc{D-tyApp}\quad \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d : \forall a.C \\ \Gamma_C;\Gamma \vdash_{ty} \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d d\,\sigma : [\sigma/a]C}$$

with $\Gamma = \bullet$.

By applying the induction hypothesis on the 1st rule premise, we get either

- d is a value. Through case analysis, we can see that $d = \Lambda a.d'$. By rule IDICTEVAL-TYAPPABS, we thus get
$$(\Lambda a.d')\,\sigma \longrightarrow [\sigma/a]d'$$
- There exists an d', such that $d \longrightarrow d'$. By rule IDICTEVAL-TYAPP, we thus get
$$d\,\sigma \longrightarrow d'\,\sigma$$

□

Theorem 32 (Progress).
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$*, then either* e *is a value, or there exists* e' *such that* $\Sigma \vdash e \longrightarrow e'$*.*

Proof. By structural induction on the typing derivation.

$$\boxed{\textbf{rule } \textsc{iTm-True}} \qquad \textsc{iTm-True}\ \frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}}$$

with $\Gamma = \bullet$.

True is a value.

$$\boxed{\textbf{rule } \textsc{iTm-False}} \qquad \textsc{iTm-False}\ \frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool}}$$

with $\Gamma = \bullet$.

False is a value.

$$\boxed{\textbf{rule } \textsc{iTm-Var}} \qquad \textsc{iTm-Var}\ \frac{(x : \sigma) \in \Gamma \quad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma}$$

with $\Gamma = \bullet$.

x cannot be well-typed in an empty context. Impossible case.

$$\boxed{\textbf{rule } \textsc{iTm-Let}} \qquad \textsc{iTm-Let}\ \frac{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \quad \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \quad \Gamma_C; \Gamma \vdash_{ty} \sigma_1}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \textbf{let}\ \ x : \sigma_1 = e_1\ \textbf{in}\ \ e_2 : \sigma_2}$$

with $\Gamma = \bullet$.

By rule $\textsc{iEval-Let}$:

$$e' = [e_1/x]e_2$$

$$\boxed{\textbf{rule } \textsc{iTm-Method}} \qquad \textsc{iTm-Method}\ \frac{\Sigma; \Gamma_C; \Gamma \vdash_d d : \mathit{TC}\,\sigma \quad (m : \mathit{TC}\, a : \sigma') \in \Gamma_C}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma'}$$

with $\Gamma = \bullet$.

Applying Theorem 31 to the 1st rule premise tells us that either:

- d is a value. As it has a class constraint type, we know that $d = D\,\overline{\sigma}_m\,\overline{d}_n$. Repeated inversion on the 1st rule premise (rule D-con) teaches us that

  $$(D : \forall \overline{a}_m . \overline{C}_n \Rightarrow TC\,\sigma_1).m \mapsto e_1 \in \Sigma$$

  By rule iEval-methodVal:

  $$e' = e_1\,\overline{\sigma}_m\,\overline{d}_n$$

- There exists a d' such that $d \longrightarrow d'$. By rule iEval-method:

  $$e' = d'.m$$

$$\boxed{\textbf{rule } \textsc{iTm-arrI}} \qquad \textsc{iTm-arrI}\ \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \\ \Gamma_C;\Gamma \vdash_{ty} \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2}$$

with $\Gamma = \bullet$.

$\lambda x : \sigma_1.e$ is a value.

$$\boxed{\textbf{rule } \textsc{iTm-arrE}} \qquad \textsc{iTm-arrE}\ \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \\ \Sigma;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1\,e_2 : \sigma_2}$$

with $\Gamma = \bullet$.

From the 1st rule premise:

$$\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \tag{C.101}$$

By applying the induction hypothesis on Equation C.101, we know that either:

- e_1 is a value. Because it has an arrow type, we know:

  $$e_1 = (\lambda x : \sigma.e_1)\,e_2$$

  By rule iEval-appAbs:

  $$e' = [e_2/x]e_1$$

- There exists an e_1' where $\Sigma \vdash e_1 \longrightarrow e_1'$. By rule iEval-app:

  $$e' = e_1'\,e_2$$

$\boxed{\textbf{rule } \textsc{iTm-constrI}}$ with $\Gamma = \bullet$.

$$\textsc{iTm-constrI}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : C \vdash_{tm} e : \sigma \\ \Gamma_C; \Gamma \vdash_C C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma}$$

$\lambda\delta : C.e$ is a value.

$\boxed{\textbf{rule } \textsc{iTm-constrE}}$ with $\Gamma = \bullet$.

$$\textsc{iTm-constrE}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : C \Rightarrow \sigma \\ \Sigma; \Gamma_C; \Gamma \vdash_d d : C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, d : \sigma}$$

From the 1st rule premise:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : C \Rightarrow \sigma \tag{C.102}$$

By applying the induction hypothesis on Equation C.102, we know that either:

- e_1 is a value. By case analysis, we know:

  $$e_1 = (\lambda\delta : C.e)$$

  By rule iEval-DAppAbs:

  $$e' = [d/\delta]e$$

- There exists an e' where $\Sigma \vdash e \longrightarrow e'$ By rule iEval-DApp:

  $$\Sigma \vdash e\, d \longrightarrow e'\, d$$

$\boxed{\textbf{rule } \textsc{iTm-forallI}}$ with $\Gamma = \bullet$.

$$\textsc{iTm-forallI}\quad \frac{\Sigma; \Gamma_C; \Gamma, a \vdash_{tm} e : \sigma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma}$$

$\Lambda a.e$ is a value.

$\boxed{\textbf{rule } \textsc{iTm-forallE}}$ with $\Gamma = \bullet$.

$$\textsc{iTm-forallE}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \\ \Gamma_C; \Gamma \vdash_{ty} \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, \sigma : [\sigma/a]\sigma'}$$

From the 1st rule premise:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \tag{C.103}$$

By applying the induction hypothesis on Equation C.103, we know that either:

- e_1 is a value. By case analysis, we know:

  $$e_1 = (\Lambda a.e)$$

  By rule iEval-tyAppAbs:

  $$e' = [\sigma/a]e$$

- There exists an e_1' where $\Sigma \vdash e \longrightarrow e'$. By rule iEval-tyApp:

  $$e' = e_1'\,\sigma$$

□

C.5.4 Strong Normalization

Theorem 33 (Strong Normalization)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$ then all possible evaluation derivations for e terminate : $\exists v : \Sigma \vdash e \longrightarrow^* v$.

Proof. By Theorem 36 and 38, with $R^{SN} = \bullet$, $\phi^{SN} = \bullet$, $\gamma^{SN} = \bullet$, since $\Gamma = \bullet$, it follows that:

$$\exists v : \Sigma \vdash e \longrightarrow^* v$$

Furthermore, since evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), there is exactly 1 possible evaluation derivation. Consequently, all derivations terminate. □

Theorem 34 (Strong Normalization - Dictionaries)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_d d : C$ then all possible evaluation derivations for d terminate : $\exists dv : d \longrightarrow^* dv$.

Proof. By Theorem 35 and 37, with $R^{SN} = \bullet$, $\gamma^{SN} = \bullet$, since $\Gamma = \bullet$, it follows that:

$$\exists dv : d \longrightarrow^* dv$$

Furthermore, since evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), there is exactly 1 possible evaluation derivation. Consequently, all derivations terminate. □

Lemma 115 (Well Typedness from Strong Normalization)**.**
$e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, then $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : {R^{SN}}_1(\sigma)$

Proof. The goal is baked into the relation. It follows by simple induction on σ. □

Lemma 116 (Strong Normalization for dictionaries preserved by forward/backward reduction)**.**
Suppose $\Sigma; \Gamma_C; \bullet \vdash_d d_1 : {R^{SN}}_1(C)$, and $d_1 \longrightarrow d_2$, then

- *If $d_1 \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}$, then $d_2 \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}$.*
- *If $d_2 \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}$, then $d_1 \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}$.*

Proof. **Part 1** By induction on C.

$\boxed{C = TC\,\sigma}$

$$d_1 \in \mathcal{SN}[\![TC\,\sigma]\!]^{\Sigma,\Gamma_C} \triangleq \Sigma;\Gamma_C;\bullet \vdash_d d_1 : TC\,\sigma$$

$$\wedge\, \exists D, \overline{\sigma}_j, \overline{d}_i : d_1 \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i$$

$$\text{where } \Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2$$

$$\text{and } (m : TC\,a : \sigma_m) \in \Gamma_C$$

$$\text{and } \overline{\Gamma_C;\bullet,\overline{a}_j \vdash_{ty} \sigma_j}^{\,j}$$

$$\text{and } \overline{d_i \in \mathcal{SN}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Sigma,\Gamma_C}}^{\,i}$$

$$\text{and } \sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q$$

$$\wedge\, e \in \mathcal{SN}[\![\forall\overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma_m]\!]_{\bullet}^{\Sigma_1,\Gamma_C}$$

We know from the rule premise that $d_1 \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i$ for some D, $\overline{\sigma}_j$ and $\overline{d}_i$. By inversion on this fact, we know that either

- $d_1 = D\,\overline{\sigma}_j\,\overline{d}_i$: Impossible case as d_1 is now a value, which contradicts the hypothesis that $d_1 \longrightarrow d_2$.
- $d_1 \longrightarrow d_2$ and $d_2 \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i$: As dictionary evaluation is deterministic (Lemma 109), we thus need to prove that

$$\Sigma;\Gamma_C;\bullet \vdash_d d_2 : TC\,\sigma \tag{C.104}$$

$$d_2 \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i \tag{C.105}$$

$$\Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2 \tag{C.106}$$

$$(m : TC\,a : \sigma_m) \in \Gamma_C \tag{C.107}$$

$$\overline{\Gamma_C;\bullet,\overline{a}_j \vdash_{ty} \sigma_j}^{\,j} \tag{C.108}$$

$$\overline{d_i \in \mathcal{SN}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Sigma,\Gamma_C}}^{\,i} \tag{C.109}$$

$$\sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q \tag{C.110}$$

$$e \in \mathcal{SN}[\![\forall\overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma_m]\!]_{\bullet}^{\Sigma_1,\Gamma_C} \tag{C.111}$$

Goal C.104 follows by preservation Theorem 29. Goals C.105 till C.111 are given by the rule premise.

$\boxed{C = C_1 \Rightarrow C_2}$

$$\begin{aligned} d_1 \in \mathcal{SN}[\![C_1 \Rightarrow C_2]\!]^{\Sigma,\Gamma_C} \triangleq\ & \Sigma;\Gamma_C;\bullet \vdash_d d_1 : C_1 \Rightarrow C_2 \\ & \wedge \exists dv : d_1 \longrightarrow^* dv \\ & \wedge \forall d' : d' \in \mathcal{SN}[\![C_1]\!]^{\Sigma,\Gamma_C} \Rightarrow d_1\, d' \in \mathcal{SN}[\![C_2]\!]^{\Sigma,\Gamma_C} \end{aligned}$$

Applying Preservation Theorem 29 to both hypotheses gives us that $\Sigma;\Gamma_C;\bullet \vdash_d d_2 : {R^{SN}}_1(C_1 \Rightarrow C_2)$. As dictionary evaluation is deterministic (Lemma 109) and we $d_1 \longrightarrow^* dv$, we know that $d_2 \longrightarrow^* dv$. Given any d' where $d' \in \mathcal{SN}[\![C_1]\!]^{\Sigma,\Gamma_C}$, we know that $d_1\, d' \longrightarrow d_2\, d'$, as $d_1 \longrightarrow d_2$. Applying the induction hypothesis thus gives that $d_2\, d' \in \mathcal{SN}[\![C_2]\!]^{\Sigma,\Gamma_C}$.

$\boxed{C = \forall a.C'}$

$$\begin{aligned} d_1 \in \mathcal{SN}[\![\forall a.C']\!]^{\Sigma,\Gamma_C} \triangleq\ & \Sigma;\Gamma_C;\bullet \vdash_d d_1 : \forall a.C' \\ & \wedge \exists dv : d_1 \longrightarrow^* dv \\ & \wedge \forall \sigma : \Gamma_C;\bullet \vdash_{ty} \sigma \Rightarrow d_1\, \sigma \in \mathcal{SN}[\![[\sigma/a]C']\!]^{\Sigma,\Gamma_C} \end{aligned}$$

Applying Preservation Theorem 29 to both hypotheses gives us that $\Sigma;\Gamma_C;\bullet \vdash_d d_2 : {R^{SN}}_1(\forall a.C')$. As dictionary evaluation is deterministic (Lemma 109) and we $d_1 \longrightarrow^* dv$, we know that $d_2 \longrightarrow^* dv$. Given any σ where $\Gamma_C;\bullet \vdash_{ty} \sigma$, we know that $d_1\, \sigma \longrightarrow d_2\, \sigma$, as $d_1 \longrightarrow d_2$. Applying the induction hypothesis thus gives that $d_2\, \sigma \in \mathcal{SN}[\![[\sigma/a]C']\!]^{\Sigma,\Gamma_C}$.

Part 2 Similar to Part 1.

□

Lemma 117 (Strong Normalization for expressions preserved by forward/backward reduction)**.**
Suppose $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : {R^{SN}}_1(\sigma)$, *and* $\Sigma \vdash e_1 \longrightarrow e_2$, *then*

- *If* $e_1 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $e_2 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.
- *If* $e_2 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $e_1 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

Proof. **Part 1** By induction on σ.

$\boxed{\sigma = Bool}$

$$e_1 \in \mathcal{SN}[\![Bool]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : Bool$$
$$\wedge\ \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$

By Preservation (Theorem 30), we know that $\Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : Bool$. Because the evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), given $\Sigma \vdash e_1 \longrightarrow^* v$, we have $\Sigma \vdash e_2 \longrightarrow^* v$.

$\boxed{\sigma = a}$

$$e_1 \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : {R^{SN}}_1(a)$$
$$\wedge\ \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$
$$\wedge\ v \in {R^{SN}}_2(a)$$

Similar to Bool case.

$\boxed{\sigma = \sigma_1 \rightarrow \sigma_2}$

$$e_1 \in \mathcal{SN}[\![\sigma_1 \rightarrow \sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : {R^{SN}}_1(\sigma_1 \rightarrow \sigma_2)$$
$$\wedge\ \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$
$$\wedge\ \forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e_1\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Preservation (Theorem 30), we know that $\Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : {R^{SN}}_1(\sigma_1 \rightarrow \sigma_2)$. Because the evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), given $\Sigma \vdash e_1 \longrightarrow^* v$, we have $\Sigma \vdash e_2 \longrightarrow^* v$. Given any $e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, we know that $\Sigma \vdash e_1 \longrightarrow e_2$, so $\Sigma \vdash e_1\, e' \longrightarrow e_2\, e'$. By induction hypothesis, we get $e_2\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

$\boxed{\sigma = C \Rightarrow \sigma'}$

$$e_1 \in \mathcal{SN}[\![C \Rightarrow \sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : {R^{SN}}_1(C \Rightarrow \sigma')$$
$$\wedge\ \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$
$$\wedge\ \forall d : d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C} \Rightarrow e_1\, d \in \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Preservation (Theorem 30), we know that $\Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : {R^{SN}}_1(C \Rightarrow \sigma')$. Because the evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), given $\Sigma \vdash e_1 \longrightarrow^* v$, we have $\Sigma \vdash e_2 \longrightarrow^* v$. Given any $d : d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}$, we know that $\Sigma \vdash e_1 \longrightarrow e_2$, so $\Sigma \vdash e_1\, d \longrightarrow e_2\, d$. By induction hypothesis, we get $e_2\, d \in \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

$\boxed{\sigma = \forall a.\sigma_1}$

$$e_1 \in \mathcal{SN}[\![\forall a.\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : {R^{SN}}_1(\forall a.\sigma_1)$$

$$\land\, \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$

$$\land\, \forall \sigma_2, r \in \mathit{Rel}[\sigma_2] : e_1\, \sigma_2 \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (\sigma_2, r)}$$

By Preservation (Theorem 30), we know that $\Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : {R^{SN}}_1(\forall a.\sigma_1)$. Because the evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), given $\Sigma \vdash e_1 \longrightarrow^* v$, we have $\Sigma \vdash e_2 \longrightarrow^* v$. Given any σ_2 and $r \in \mathit{Rel}[\sigma_2]$, we know that $\Sigma \vdash e_1 \longrightarrow e_2$, so $\Sigma \vdash e_1\, \sigma_2 \longrightarrow e_2\, \sigma_2$. By induction hypothesis, we get $e_2\, \sigma_2 \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (\sigma_2, r)}$.

Part 2 Similar to Part 1.

□

Lemma 118 (Substitution for Context Interpretation)**.**

- *If* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, $\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$ *and* $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *we have* $\Sigma; \Gamma_C; \bullet \vdash_{tm} \gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))) : {R^{SN}}_1(\sigma)$.
- *If* $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$ *and* $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *we have* $\Sigma; \Gamma_C; \bullet \vdash_d \gamma^{SN}({R^{SN}}_1(d)) : {R^{SN}}_1(C)$.
- *If* $\Gamma_C; \Gamma \vdash_{ty} \sigma$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, *we have* $\Gamma_C; \bullet \vdash_{ty} {R^{SN}}_1(\sigma)$.
- *If* $\Gamma_C; \Gamma \vdash_Q Q$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, *we have* $\Gamma_C; \bullet \vdash_Q {R^{SN}}_1(Q)$.
- *If* $\Gamma_C; \Gamma \vdash_C C$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, *we have* $\Gamma_C; \bullet \vdash_C {R^{SN}}_1(C)$.

Proof. By induction on e, d σ, Q and C respectively. The goal follows from Definitions 14, 15 and 16.

□

Lemma 119 (Compositionality for Strong Normalization)**.** *Let* $r = \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$*, then* $e \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a\mapsto(R^{SN}{}_1(\sigma_2), r)}$ *if and only if* $e \in \mathcal{SN}[\![[\sigma_2/a]\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$*.*

Proof. By induction on σ_1.

Bool $\sigma_1 = Bool$

Since $[\sigma_2/a]Bool = Bool$, the goal follows directly.

Type variable $\sigma_1 = b$

Depending on whether b is the same variable as a, we have two cases:

- If $b = a$, then $[\sigma_2/a]a = \sigma_2$.

  Part 1: From left to right. If $e \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a\mapsto(R^{SN}{}_1(\sigma_2), r)}$, it means that:

  $$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.112}$$

  $$v \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.113}$$

  Combining with Strong Normalization preserved by forward/backward reduction (Lemma 117), the goal is proven by Equations C.112 and C.113:

  $$e \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.114}$$

  Part 2: From right to left. From the hypothesis, we know that:

  $$e \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.115}$$

  We want to prove that $e \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a\mapsto(R^{SN}{}_1(\sigma_2), r)}$. By the definition, this goal is equivalent to:

  $$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : (R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r))(a) \tag{C.116}$$

  $$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.117}$$

  $$v \in (R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r))_2\ (a) \tag{C.118}$$

  Equation C.116 simplifies to:

  $$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : R^{SN}{}_1(\sigma_2) \tag{C.119}$$

By Well-Typedness from Strong Normalization (Lemma 115), Equation C.115 proves C.119. By Strong Normalization - Part B (Theorem 38), Equation C.115 proves C.117.

We already know that $(R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r))_2\ (a) = r = \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, so we simplify Equation C.118 to get

$$v \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.120}$$

From Equation C.115 and Strong Normalization preserved by forward/backward reduction (Lemma 117), we can prove Goal C.120.

- If $b \neq a$, since $[\sigma_2/a]b = b$, the goal follows directly.

Function $\sigma_1 = \sigma_{11} \rightarrow \sigma_{12}$

Part 1: From left to right. From the hypothesis, we get:

$$e \in \mathcal{SN}[\![\sigma_{11} \rightarrow \sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(R^{SN}{}_1(\sigma_2),r)}$$

We thus know that:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : (R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r))(\sigma_{11} \rightarrow \sigma_{12}) \tag{C.121}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.122}$$

$$\forall e' : e' \in \mathcal{SN}[\![\sigma_{11}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(R^{SN}{}_1(\sigma_2),r)} \Rightarrow e\,e' \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(R^{SN}{}_1(\sigma_2),r)} \tag{C.123}$$

Our goal is to prove that $e \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{11} \rightarrow [\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. This is equivalent to proving:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : R^{SN}{}_1([\sigma_2/a]\sigma_{11} \rightarrow [\sigma_2/a]\sigma_{12}) \tag{C.124}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.125}$$

$$\forall e' : e' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{11}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e\,e' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.126}$$

Goal C.124 and C.125 are proven directly by Equations C.121 and C.122. Only Goal C.126 remains to be proven.

Given $e' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{11}]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, the induction hypothesis tells us that $e' \in \mathcal{SN}[\![\sigma_{11}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(R^{SN}{}_1(\sigma_2),r)}$. In combination with equation Equation C.123, we get

$$e\,e' \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(R^{SN}{}_1(\sigma_2),r)} \tag{C.127}$$

By induction hypothesis, we get:

$$e\, e' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.128}$$

The goal has been proven.

Part 2: From right to left. Similar to Part 1.

$\boxed{\textbf{Function over constraints}}$ $\sigma_1 = C \Rightarrow \sigma_{12}$

Part 1: From left to right. We know from the hypothesis that:

$$e \in \mathcal{SN}[\![C \Rightarrow \sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto({R^{SN}}_1(\sigma_2),r)}$$

It follows that:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} e : (R^{SN}, a \mapsto ({R^{SN}}_1(\sigma_2), r))(C \Rightarrow \sigma_{12}) \tag{C.129}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.130}$$

$$\forall d : \Sigma;\Gamma_C;\bullet \vdash_d d : (R^{SN}, a \mapsto ({R^{SN}}_1(\sigma_2), r))(C) \tag{C.131}$$

$$\Rightarrow e\, d \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto({R^{SN}}_1(\sigma_2),r)}$$

Our goal is to prove that $e \in \mathcal{SN}[\![[\sigma_2/a]C \Rightarrow [\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. This is equivalent to proving:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} e : {R^{SN}}_1([\sigma_2/a]C \Rightarrow [\sigma_2/a]\sigma_{12}) \tag{C.132}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.133}$$

$$\forall d' : \Sigma;\Gamma_C;\bullet \vdash_d d' : {R^{SN}}_1([\sigma_2/a]C) \Rightarrow e\, d' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.134}$$

Goals C.132 and C.133 are proven directly by Equations C.129 and C.130. Only Goal C.134 remains to be proven. Given $\Sigma;\Gamma_C;\bullet \vdash_d d' : {R^{SN}}_1([\sigma_2/a]C)$, in combination with Equation C.131, we get that:

$$e\, d' \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto({R^{SN}}_1(\sigma_2),r)} \tag{C.135}$$

By induction hypothesis, we get:

$$e\, d' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.136}$$

The goal has been proven.

Part 2: From right to left. Similar to Part 1.

Polymorphic types $\sigma_1 = \forall b.\sigma_{12}$

Part 1: From left to right. We know from the hypothesis that:

$$e \in \mathcal{SN}[\![\forall b.\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r)}$$

This implies that:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : (R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r))(\forall b.\sigma_{12}) \tag{C.137}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.138}$$

$$\forall \sigma', r = \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} : e\,\sigma' \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r)} \tag{C.139}$$

Our goal is to prove that $e \in \mathcal{SN}[\![\forall b.[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. This is equivalent to proving:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e : R^{SN}{}_1(\forall b.[\sigma_2/a]\sigma_{12}) \tag{C.140}$$

$$\exists v : \Sigma \vdash e \longrightarrow^* v \tag{C.141}$$

$$\forall \sigma', r = \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e\,\sigma' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.142}$$

Goals C.140 and C.141 are directly proven by Equations C.137 and C.138. Only Goal C.142 remains to be proven. Given σ' and $r = \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, by feeding it to Equation C.139, we get that:

$$e\,\sigma' \in \mathcal{SN}[\![\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}, a \mapsto (R^{SN}{}_1(\sigma_2), r)} \tag{C.143}$$

By induction hypothesis, we get:

$$e\,\sigma' \in \mathcal{SN}[\![[\sigma_2/a]\sigma_{12}]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.144}$$

The goal has been proven.

Part 2: From right to left. Similar to Part 1.

□

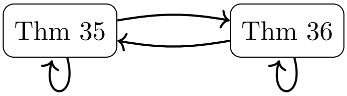

Figure C.2: Dependency graph for Strong Normalization Theorems

Corollary 1 (Compositionality for Strong Normalization (Context Interpretation))**.** *Suppose* $R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$*, then* $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$ *if and only if* $e \in \mathcal{SN}[\![{R^{SN}}_1(\sigma)]\!]^{\Sigma,\Gamma_C}_{\bullet}$.

Proof. The choices from $R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$ always satisfy the precondition of Compositionality for Strong Normalization (Lemma 119). Therefore, we can do induction on Γ and apply Compositionality for Strong Normalization (Lemma 119), in combination with the induction hypothesis, to prove the goal.

□

Theorem 35 (Strong Normalization - Dictionaries - Part A)**.** *If* $\Sigma; \Gamma_C; \Gamma \vdash_d d : C$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *it holds that* $\gamma^{SN}({R^{SN}}_1(d)) \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}$.

Proof. By induction on the first hypothesis of the theorem. This theorem is proven by mutual induction with Theorem 36, as illustrated in Figure C.2. Note that at the dependency from Theorem 35 to 36, the size of Σ is strictly decreasing, while Σ remains contant in the other direction. The induction thus remains well-founded.

$$\boxed{\textbf{rule } \text{D-VAR}} \qquad \begin{array}{c} \text{D-VAR} \\ \dfrac{(\delta : C) \in \Gamma \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C} \end{array}$$

The 3rd theorem hypothesis tells us that $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. Furthermore, we know from the rule premise that $(\delta : C) \in \Gamma$. We can thus conclude by the definition of $\mathcal{H}^{\mathcal{SN}}$ that $\delta \mapsto d \in \gamma^{SN}$ for some d with $d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}$.

By definition of the $\mathcal{SN}$ relation, we know that d does not contain any free variables in Γ. We thus know that $\gamma^{SN}({R^{SN}}_1(d)) = d$, and as a result that $\gamma^{SN}({R^{SN}}_1(\delta)) = d$. The goal to be proven thus becomes $d \in \mathcal{SN}[\![{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}$, which we have shown previously.

$$\dfrac{\begin{array}{c}\text{D-CON}\\ \Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i}^{\,i} \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q / a]\sigma_m \end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q}$$

$\boxed{\textbf{rule } \text{D-CON}}$

By simplifying the substitution, using that $\gamma^{SN}({R^{SN}}_1(D)) = D$ and ${R^{SN}}_1(\forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q) = \forall \overline{a}_j . {R^{SN}}_1(\overline{C}_i) \Rightarrow TC\,{R^{SN}}_1(\sigma_q)$, the goal becomes

$$D \in \mathcal{SN}\llbracket \forall \overline{a}_j . {R^{SN}}_1(\overline{C}_i) \Rightarrow TC\,{R^{SN}}_1(\sigma_q) \rrbracket^{\Sigma, \Gamma_C}$$

Repeatedly unfolding the definition of the $\mathcal{SN}$ relation in this goal reduces it to

$$\forall k \in 0 \ldots j : \forall \sigma_k : \overline{\Gamma_C; \bullet \vdash_{ty} \sigma_k}^{\,k<j} \tag{C.145}$$

$$\Rightarrow \Sigma; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_k : [\overline{\sigma}_k / \overline{a}_k](\forall \overline{a}_n . {R^{SN}}_1(\overline{C}_i) \Rightarrow TC\,{R^{SN}}_1(\sigma_q)) \tag{C.146}$$

where $n = j - k$

$$\wedge\ \exists dv : D\,\overline{\sigma}_k \longrightarrow^* dv \tag{C.147}$$

$$\forall k \in 0 \ldots i : \forall d_k : \overline{d_k \in \mathcal{SN}\llbracket [\overline{\sigma}_j / \overline{a}_j] {R^{SN}}_1(C_k) \rrbracket^{\Sigma, \Gamma_C}}^{\,k<i} \tag{C.148}$$

$$\Rightarrow \Sigma; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{d}_k : [\overline{\sigma}_j / \overline{a}_k]({R^{SN}}_1(\overline{C}_n) \Rightarrow TC\,{R^{SN}}_1(\sigma_q)) \tag{C.149}$$

where $n = i - k$

$$\wedge\ \exists dv : D\,\overline{\sigma}_j\,\overline{d}_k \longrightarrow^* dv \tag{C.150}$$

along with

$$\Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2 \tag{C.151}$$

$$(m : TC\,a : \sigma_m) \in \Gamma_C \tag{C.152}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_{ty} \sigma_j}^{\,j} \tag{C.153}$$

$$\overline{d_i \in \mathcal{SN}\llbracket [\overline{\sigma}_j / \overline{a}_j] C_i \rrbracket^{\Sigma, \Gamma_C}}^{\,i} \tag{C.154}$$

$$e \in \mathcal{SN}\llbracket \forall \overline{a}_j . \overline{C}_i \Rightarrow [\sigma_q / a]\sigma_m \rrbracket^{\Sigma, \Gamma_C}_{\bullet} \tag{C.155}$$

Goal C.146 follows by repeated application of rule D-TYAPP, together with the 1st hypothesis. Goal C.147 is trivial as $D\,\overline{\sigma}_k$ is already a dictionary value.

Goal C.149 holds by (repeatedly) applying rule D-DAPP to Equation C.146. Note that the well-typedness of d_k follows by the definition of the $\mathcal{SN}$ relation. Goal C.150 trivially holds as $D\,\overline{\sigma}_j\,\overline{d}_k$ is already a value. Goals C.151 and C.152 correspond to the 1st and 2nd rule premise, respectively. Goals C.153 and C.154 follow directly by Equations C.146 and C.149, respectively. And finally, Goal C.155 follows by applying Theorem 36 to the 5th rule premise.

$$\boxed{\textbf{rule } \textsc{D-dabs}} \qquad \textsc{D-dabs}\ \frac{\Sigma;\Gamma_C;\Gamma,\delta : C_1 \vdash_d d : C_2 \qquad \Gamma_C;\Gamma \vdash_C C_1}{\Sigma;\Gamma_C;\Gamma \vdash_d \lambda\delta : C_1.d : C_1 \Rightarrow C_2}$$

By simplifying the substitution, using that $\gamma^{SN}({R^{SN}}_1((\lambda\delta : C_1.d))) = \lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d)))$ and ${R^{SN}}_1(C_1 \Rightarrow C_2) = {R^{SN}}_1(C_1) \Rightarrow {R^{SN}}_1(C_2)$, the goal becomes

$$\lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))) \in \mathcal{SN}[\![{R^{SN}}_1(C_1) \Rightarrow {R^{SN}}_1(C_2)]\!]^{\Sigma,\Gamma_C}$$

Unfolding the definition of the $\mathcal{SN}$ relation reduces the goal further to

$$\Sigma;\Gamma_C;\bullet \vdash_d \lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))) : {R^{SN}}_1(C_1) \Rightarrow {R^{SN}}_1(C_2) \tag{C.156}$$

$$\exists dv : \lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))) \longrightarrow^* dv \tag{C.157}$$

$$\forall d' : d' \in \mathcal{SN}[\![{R^{SN}}_1(C_1)]\!]^{\Sigma,\Gamma_C}$$

$$\Rightarrow (\lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))))\,d' \in \mathcal{SN}[\![{R^{SN}}_1(C_2)]\!]^{\Sigma,\Gamma_C} \tag{C.158}$$

Goal C.156 follows by applying Lemma 118 to the 1st hypothesis. Goal C.157 holds trivially as $\lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d)))$ is already a dictionary value. We thus focus on proving Goal C.158. Given

$$\forall d' : d' \in \mathcal{SN}[\![{R^{SN}}_1(C_1)]\!]^{\Sigma,\Gamma_C} \tag{C.159}$$

We need to prove

$$(\lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))))\,d' \in \mathcal{SN}[\![{R^{SN}}_1(C_2)]\!]^{\Sigma,\Gamma_C} \tag{C.160}$$

Applying the induction hypothesis to the 1st rule premise gives us

$$\gamma^{SN'}({R^{SN'}}_1(d)) \in \mathcal{SN}[\![{R^{SN'}}_1(C_1)]\!]^{\Sigma,\Gamma_C} \tag{C.161}$$

for any $R^{SN'} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma,\delta : C_1]\!]^{\Sigma,\Gamma_C}$ and $\gamma^{SN'} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma,\delta : C_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. Following their respective definitions, we choose $R^{SN'} = R^{SN}$ and $\gamma^{SN'} = \gamma^{SN}, \delta \mapsto d'$.

By rule IDICTEVAL-APPABS, we have

$$(\lambda\delta : {R^{SN}}_1(C_1).(\gamma^{SN}({R^{SN}}_1(d))))\,d' \longrightarrow [d'/\delta](\gamma^{SN}({R^{SN}}_1(d)))$$

Furthermore, as d' does not depend on any dictionary variables (by definition):

$$\gamma^{SN}, \delta \mapsto d'({R^{SN}}_1(d)) = [d'/\delta](\gamma^{SN}({R^{SN}}_1(d)))$$

These results thus prove Goal C.160, by Lemma 116.

$$\boxed{\textbf{rule}\ \textsc{D-dapp}} \qquad \overset{\textsc{D-dapp}}{\frac{\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \qquad \Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1}{\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2}}$$

By simplifying the substitutions, using that

$$\gamma^{SN}({R^{SN}}_1((d_1\, d_2))) = (\gamma^{SN}({R^{SN}}_1(d_1)))\, (\gamma^{SN}({R^{SN}}_1(d_2)))$$

the goal becomes

$$(\gamma^{SN}({R^{SN}}_1(d_1)))\, (\gamma^{SN}({R^{SN}}_1(d_2))) \in \mathcal{SN}[\![{R^{SN}}_1(C_2)]\!]^{\Sigma,\Gamma_C}$$

Applying the induction hypothesis twice, to both rule premises, gives us

$$\gamma^{SN}({R^{SN}}_1(d_1)) \in \mathcal{SN}[\![{R^{SN}}_1(C_1 \Rightarrow C_2)]\!]^{\Sigma,\Gamma_C} \tag{C.162}$$

$$\gamma^{SN}({R^{SN}}_1(d_2)) \in \mathcal{SN}[\![{R^{SN}}_1(C_1)]\!]^{\Sigma,\Gamma_C} \tag{C.163}$$

Noting that ${R^{SN}}_1(C_1 \Rightarrow C_2) = {R^{SN}}_1(C_1) \Rightarrow {R^{SN}}_1(C_2)$, the definition of the $\mathcal{SN}$ relation in Equation C.162 tells us that

$$\forall d' : d' \in \mathcal{SN}[\![{R^{SN}}_1(C_1)]\!]^{\Sigma,\Gamma_C} \Rightarrow (\gamma^{SN}({R^{SN}}_1(d_1)))\, d' \in \mathcal{SN}[\![{R^{SN}}_1(C_2)]\!]^{\Sigma,\Gamma_C}$$

The goal thus follows directly from this result by taking $d' = \gamma^{SN}({R^{SN}}_1(d_2))$ and applying Equation C.163.

$$\boxed{\textbf{rule}\ \textsc{D-tyabs}} \qquad \overset{\textsc{D-tyabs}}{\frac{\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C}{\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C}}$$

By simplifying the substitution, using that $\gamma^{SN}({R^{SN}}_1((\Lambda a.d))) = \Lambda a.(\gamma^{SN}({R^{SN}}_1(d)))$ and ${R^{SN}}_1(\forall a.C) = \forall a.{R^{SN}}_1(C)$, the goal becomes

$$\Lambda a.(\gamma^{SN}({R^{SN}}_1(d))) \in \mathcal{SN}[\![\forall a.{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C}$$

Unfolding the definition of the $\mathcal{SN}$ relation reduces the goal further to

$$\Sigma; \Gamma_C; \bullet \vdash_d \Lambda a.(\gamma^{SN}({R^{SN}}_1(d))) : \forall a.{R^{SN}}_1(C) \tag{C.164}$$

$$\exists dv : \Lambda a.(\gamma^{SN}({R^{SN}}_1(d))) \longrightarrow^{*} dv \tag{C.165}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow (\Lambda a.(\gamma^{SN}({R^{SN}}_1(d))))\, \sigma \in \mathcal{SN}[\![[\sigma/a]{R^{SN}}_1(C)]\!]^{\Sigma,\Gamma_C} \tag{C.166}$$

Goal C.164 follows by applying Lemma 118 to the 1st hypothesis. Goal C.165 holds trivially as $\Lambda a.(\gamma^{SN}(R^{SN}{}_1(d)))$ is already a dictionary value. We thus focus on proving Goal C.166. Given

$$\forall \sigma : \Gamma_C ; \bullet \vdash_{ty} \sigma \tag{C.167}$$

We need to prove

$$(\Lambda a.(\gamma^{SN}(R^{SN}{}_1(d))))\,\sigma \in \mathcal{SN}[\![[\sigma/a]R^{SN}{}_1(C)]\!]^{\Sigma,\Gamma_C} \tag{C.168}$$

Applying the induction hypothesis to the 1st rule premise gives us

$$\gamma^{SN'}(R^{SN'}{}_1(d)) \in \mathcal{SN}[\![R^{SN'}{}_1(C)]\!]^{\Sigma,\Gamma_C} \tag{C.169}$$

for any $R^{SN'} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma, a]\!]^{\Sigma,\Gamma_C}$ and $\gamma^{SN'} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma, a]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. Following their respective definitions, we choose $\gamma^{SN'} = \gamma^{SN}$ and $R^{SN'} = R^{SN}, a \mapsto (\sigma, r)$, for some r.

By rule iDictEval-tyAppAbs, we have

$$(\Lambda a.(\gamma^{SN}(R^{SN}{}_1(d))))\,\sigma \longrightarrow [\sigma/a](\gamma^{SN}(R^{SN}{}_1(d)))$$

Furthermore, as σ does not depend on any variables (by definition):

$$\gamma^{SN}(R^{SN}, a \mapsto (\sigma, r)(d)) = [\sigma/a](\gamma^{SN}(R^{SN}{}_1(d)))$$

These results thus prove Goal C.168, by Lemma 116.

$$\boxed{\textbf{rule}\ \text{D-TYAPP}} \qquad \begin{array}{c} \text{D-TYAPP} \\ \dfrac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_d d : \forall a.C \\ \Gamma_C;\Gamma \vdash_{ty} \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_d d\,\sigma : [\sigma/a]C} \end{array}$$

By simplifying the substitutions, using that

$$\gamma^{SN}(R^{SN}{}_1((d\,\sigma))) = (\gamma^{SN}(R^{SN}{}_1(d)))\,(R^{SN}{}_1(\sigma))$$

the goal becomes

$$(\gamma^{SN}(R^{SN}{}_1(d)))\,(R^{SN}{}_1(\sigma)) \in \mathcal{SN}[\![R^{SN}{}_1([\sigma/a]C)]\!]^{\Sigma,\Gamma_C}$$

Applying the induction hypothesis, to the 1st rule premise, gives us

$$\gamma^{SN}(R^{SN}{}_1(d)) \in \mathcal{SN}[\![R^{SN}{}_1(\forall a.C)]\!]^{\Sigma,\Gamma_C} \tag{C.170}$$

Noting that $R^{SN}{}_1(\forall a.C) = \forall a.R^{SN}{}_1(C)$, the definition of the $\mathcal{SN}$ relation in Equation C.170 tells us that

$$\forall \sigma' : (\gamma^{SN}(R^{SN}{}_1(d)))\,\sigma' \in \mathcal{SN}[\![R^{SN}{}_1([\sigma'/a]C)]\!]^{\Sigma,\Gamma_C}$$

The goal thus follows directly from this result by taking $\sigma' = R^{SN}{}_1(\sigma)$. □

Theorem 36 (Strong Normalization - Expressions - Part A)**.**
If $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma$ *then* $\forall R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, $\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$ *and* $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$,
it holds that $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))) \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

Proof. By induction on the first hypothesis of the theorem. This theorem is proven by mutual induction with Theorem 35, as illustrated in Figure C.2. Note that at the dependency from Theorem 35 to 36, the size of Σ is strictly decreasing, while Σ remains contant in the other direction. The induction thus remains well-founded.

$$\boxed{\textbf{rule}\ \textsc{iTm-true}} \qquad \textsc{iTm-true}\ \dfrac{\vdash_{ctx} \Sigma;\Gamma_C;\Gamma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}}$$

We know $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(\mathit{True}))) = \mathit{True}$. So the goal is $\mathit{True} \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. The goal follows directly since $\Sigma;\Gamma_C;\bullet \vdash_{tm} \mathit{True} : \mathit{Bool}$ and $\Sigma \vdash \mathit{True} \longrightarrow^{*} \mathit{True}$.

$$\boxed{\textbf{rule}\ \textsc{iTm-false}} \qquad \textsc{iTm-false}\ \dfrac{\vdash_{ctx} \Sigma;\Gamma_C;\Gamma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \mathit{False} : \mathit{Bool}}$$

Similar to the rule iTm-true case.

$$\boxed{\textbf{rule}\ \textsc{iTm-var}} \qquad \textsc{iTm-var}\ \dfrac{(x : \sigma) \in \Gamma \qquad \vdash_{ctx} \Sigma;\Gamma_C;\Gamma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} x : \sigma}$$

We know that $\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$ and $(x : \sigma) \in \Gamma$, so we know that $x \mapsto e \in \phi^{SN}$ for some e with $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. Since $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, we know from the definition of the relation that e does not contain any free variables in Γ. Consequently, $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))) = e$. Therefore, $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(x))) = e$. Now our goal becomes $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, which we already know.

$$\boxed{\textbf{rule}\ \textsc{iTm-let}} \qquad \textsc{iTm-let}\ \dfrac{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \qquad \Sigma;\Gamma_C;\Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \qquad \Gamma_C;\Gamma \vdash_{ty} \sigma_1}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \textbf{let}\ x : \sigma_1 = e_1\ \textbf{in}\ e_2 : \sigma_2}$$

By induction hypothesis, we know

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1))) \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.171}$$

$$\gamma^{SN}(\phi^{SN}{}_2(R^{SN}{}_1(e_2))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.172}$$

Given Equation C.171, we can choose $\phi^{SN}{}_2 = \phi^{SN}, x \mapsto \gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))$. Equation C.172 thus reduces to:

$$\gamma^{SN}(\phi^{SN}, x \mapsto \gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))(R^{SN}{}_1(e_2))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.173}$$

Simplifying Equation C.173 results in:

$$[\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))/x](\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2)))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.174}$$

Our goal is $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(\textbf{let }\ x : \sigma_1 = e_1 \textbf{ in }\ e_2))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

We know that

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(\textbf{let }\ x : \sigma_1 = e_1 \textbf{ in }\ e_2)))$$

$$= \textbf{let }\ x : \sigma_1 = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))) \textbf{ in }\ (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2)))) \tag{C.175}$$

$$\Sigma \vdash \textbf{let }\ x : \sigma_1 = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))) \textbf{ in }\ (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2))))$$

$$\longrightarrow [\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))/x](\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2)))) \tag{C.176}$$

Consequently, by Substitution for Context Interpretation (Lemma 118), we know that

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e_3 : R^{SN}{}_1(\sigma_2) \tag{C.177}$$

where $e_3 = \textbf{let }\ x : \sigma_1 = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))) \textbf{ in }\ (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2))))$. By Equations C.174, C.176 and C.177, in combination with Strong Normalization preserved by forward/backward reduction (Lemma 117), we get that

$$\textbf{let }\ x : \sigma_1 = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_1)))) \textbf{ in }\ (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e_2)))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.178}$$

The goal follows from Equations C.175 and C.178.

$$\boxed{\textbf{rule } \textsc{iTm-method}} \qquad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\, \sigma \\ (m : TC\, a : \sigma') \in \Gamma_C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma'}\ \textsc{iTm-method}$$

The goal to be proven, adapted to this case, is

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(d.m))) \in \mathcal{SN}[\![[\sigma/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

for all $R^{SN} \in \mathcal{F}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}$, $\phi^{SN} \in \mathcal{G}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, $\gamma^{SN} \in \mathcal{H}^{\mathcal{SN}}[\![\Gamma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. Simplifying the substitutions reduces this to

$$(\gamma^{SN}(R^{SN}{}_1(d))).m \in \mathcal{SN}[\![[\sigma/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.179}$$

Applying Theorem 35 to the 1st rule premise, gives us

$$\gamma^{SN}({R^{SN}}_1(d)) \in \mathcal{SN}[\![{R^{SN}}_1(TC\,\sigma)]\!]^{\Sigma,\Gamma_C} \tag{C.180}$$

by choosing the same R^{SN} and γ^{SN} as above. Unfolding the definition of the $\mathcal{SN}$ relation in Equation C.180, and using the uniqueness of Γ_C along with the 2nd rule premise, gives us

$$\gamma^{SN}({R^{SN}}_1(d)) \longrightarrow^* D\,\overline{\sigma}_j\,\overline{d}_i \tag{C.181}$$

$$\Sigma = \Sigma_1, (D : \forall\overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e, \Sigma_2 \tag{C.182}$$

$$e \in \mathcal{SN}[\![\forall\overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma']\!]^{\Sigma_1,\Gamma_C}_{\bullet} \tag{C.183}$$

for some D, $\overline{\sigma}_j$ and $\overline{d}_i$, where

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_{ty} \sigma_j}^{\,j} \tag{C.184}$$

$$\overline{d_i \in \mathcal{SN}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Sigma,\Gamma_C}}^{\,i} \tag{C.185}$$

$$\sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q \tag{C.186}$$

Repeatedly applying rule iEval-method, in combination with Equation C.181 tells us that

$$\Sigma \vdash (\gamma^{SN}({R^{SN}}_1(d))).m \longrightarrow^* (D\,\overline{\sigma}_j\,\overline{d}_i).m$$

Applying rule iEval-methodVal to this result, together with Equation C.182 gives us

$$\Sigma \vdash (\gamma^{SN}({R^{SN}}_1(d))).m \longrightarrow^* e\,\overline{\sigma}_j\,\overline{d}_i$$

Repeatedly applying Lemma 117 thus reduces Goal C.179 to

$$e\,\overline{\sigma}_j\,\overline{d}_i \in \mathcal{SN}[\![[([\overline{\sigma}_j/\overline{a}_j]\sigma_q)/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Lemma 119, as we know that σ' only depends on a, this goal is equivalent to

$$e\,\overline{\sigma}_j\,\overline{d}_i \in \mathcal{SN}[\![[(\sigma_q)/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN},\overline{a}_j \mapsto (\overline{\sigma}_j,\overline{r}_j)} \tag{C.187}$$

for any $\overline{r_j \in \mathit{Rel}[\sigma_j]}^{\,j}$. Applying Lemmas 101 and 102 to Equation C.183 gives us

$$e \in \mathcal{SN}[\![\forall\overline{a}_j.\overline{C}_i \Rightarrow [\sigma_q/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.188}$$

By the definition of the $\mathcal{SN}$ relation, Goal C.187 follows from Equation C.188, in combination with Equations C.184 and C.185.

$$\boxed{\textbf{rule}\ \textsc{iTm-arrI}} \quad \frac{\begin{array}{c}\textsc{iTm-arrI}\\ \Sigma;\Gamma_C;\Gamma, x:\sigma_1 \vdash_{tm} e:\sigma_2 \\ \Gamma_C;\Gamma \vdash_{ty} \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \lambda x:\sigma_1.e : \sigma_1 \to \sigma_2}$$

Because $\gamma^{SN}(\phi^{SN}({R^{SN}}_1((\lambda x:\sigma_1.e)))) = \lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$, our goal is to show that:

$$\lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) \in \mathcal{SN}[\![\sigma_1 \to \sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.189}$$

By definition, we need to prove the following goals:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} \lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) : {R^{SN}}_1(\sigma_1 \to \sigma_2) \tag{C.190}$$

$$\exists v : \Sigma \vdash \lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) \longrightarrow^* v \tag{C.191}$$

$$\forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.192}$$

$$\Rightarrow (\lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Substitution for Context Interpretation (Lemma 118), we can easily prove Equation C.190.

Furthermore, $\lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$ is already a value, which proves Equation C.191. Now given

$$\forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.193}$$

We need to show

$$(\lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.194}$$

Let $\phi^{SN'} = \phi^{SN}, x \mapsto e'$. By induction hypothesis, we have

$$\gamma^{SN}(\phi^{SN}, x \mapsto e'({R^{SN}}_1(e))) \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.195}$$

We know that:

$$(\lambda x:{R^{SN}}_1(\sigma_1).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\, e'$$

$$\longrightarrow [e'/x](\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$$

$$= \gamma^{SN}(\phi^{SN}, x \mapsto e'({R^{SN}}_1(e)))$$

Consequently, by Strong Normalization preserved by forward/backward reduction(Lemma 117), Equation C.195 proves C.194.

$$\boxed{\textbf{rule}\ \textsc{iTm-arrE}}\qquad \dfrac{\begin{array}{c}\textsc{iTm-arrE}\\ \Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \\ \Sigma;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma_1\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e_1\, e_2 : \sigma_2}$$

By induction hypothesis, we have:

$$\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_1))) \in \mathcal{SN}\llbracket \sigma_1 \to \sigma_2 \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.196}$$

$$\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_2))) \in \mathcal{SN}\llbracket \sigma_1 \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.197}$$

By the definition of Equation C.196, applying Equation C.197 results in:

$$(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_1))))\,(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_2)))) \in \mathcal{SN}\llbracket \sigma_2 \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}}$$

which is exactly our goal since

$$\gamma^{SN}(\phi^{SN}({R^{SN}}_1((e_1\, e_2)))) = (\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_1))))\,(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e_2))))$$

$$\boxed{\textbf{rule}\ \textsc{iTm-constrI}}\qquad \dfrac{\begin{array}{c}\textsc{iTm-constrI}\\ \Sigma;\Gamma_C;\Gamma,\delta : C \vdash_{tm} e : \sigma \\ \Gamma_C;\Gamma \vdash_C C\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma}$$

Because $\gamma^{SN}(\phi^{SN}({R^{SN}}_1((\lambda\delta : C.e)))) = \lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$, our goal is to show that:

$$\lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) \in \mathcal{SN}\llbracket C \Rightarrow \sigma \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.198}$$

By definition, we need to prove the following goals:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} \lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) : {R^{SN}}_1(C \Rightarrow \sigma) \tag{C.199}$$

$$\exists v : \Sigma \vdash \lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) \longrightarrow^{*} v \tag{C.200}$$

$$\forall d : d \in \mathcal{SN}\llbracket {R^{SN}}_1(C) \rrbracket^{\Sigma,\Gamma_C} \tag{C.201}$$

$$\Rightarrow (\lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\, d \in \mathcal{SN}\llbracket \sigma \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Substitution for Context Interpretation (Lemma 118), we can easily prove Equation C.199.

Furthermore, $\lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$ is already a value, which proves Equation C.200. Now given

$$\forall d : d \in \mathcal{SN}\llbracket {R^{SN}}_1(C) \rrbracket^{\Sigma,\Gamma_C} \tag{C.202}$$

We need to show

$$(\lambda\delta : {R^{SN}}_1(C).(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\, d \in \mathcal{SN}\llbracket \sigma \rrbracket^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.203}$$

Let $\gamma^{SN'} = \gamma^{SN}, \delta \mapsto d$. By induction hypothesis, we have

$$\gamma^{SN}, \delta \mapsto d(\phi^{SN}(R^{SN}{}_1(e))) \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.204}$$

We know that:

$$(\lambda\delta : R^{SN}{}_1(C).(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e)))))\, d$$

$$\longrightarrow [d/\delta](\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))))$$

$$= \gamma^{SN}, \delta \mapsto d(\phi^{SN}(R^{SN}{}_1(e)))$$

Consequently, by Strong Normalization preserved by forward/backward reduction (Lemma 117), Equation C.204 proves C.203.

$$\boxed{\textbf{rule } \textsc{iTm-constrE}} \qquad \begin{array}{l}\textsc{iTm-constrE}\\ \dfrac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : C \Rightarrow \sigma\\ \Sigma;\Gamma_C;\Gamma \vdash_{d} d : C\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\, d : \sigma}\end{array}$$

Applying the induction hypothesis to the 1st rule premise gives us

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))) \in \mathcal{SN}[\![C \Rightarrow \sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.205}$$

Applying Theorem 35 to the 2nd rule premise gives us

$$\gamma^{SN}(R^{SN}{}_1(d)) \in \mathcal{SN}[\![R^{SN}{}_1(C)]\!]^{\Sigma,\Gamma_C} \tag{C.206}$$

By the definition of Equation C.205, applying Equation C.206 results in:

$$(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))))\,(\gamma^{SN}(R^{SN}{}_1(d))) \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

which is exactly our goal since

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1((e\, d)))) = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))))\,(\gamma^{SN}(R^{SN}{}_1(d)))$$

$$\boxed{\textbf{rule } \textsc{iTm-forallI}} \qquad \begin{array}{l}\textsc{iTm-forallI}\\ \dfrac{\Sigma;\Gamma_C;\Gamma, a \vdash_{tm} e : \sigma}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma}\end{array}$$

Because $\gamma^{SN}(\phi^{SN}(R^{SN}{}_1((\Lambda a.e)))) = \Lambda a.(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))))$, our goal is to show that:

$$\Lambda a.(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e)))) \in \mathcal{SN}[\![\forall a.\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.207}$$

By definition, we need to prove the following goals:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} \Lambda a.(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e)))) : R^{SN}{}_1(\forall a.\sigma) \tag{C.208}$$

$$\exists v : \Sigma \vdash \Lambda a.(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e)))) \longrightarrow^* v \tag{C.209}$$

$$\forall \sigma', r = \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow (\Lambda a.(\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e)))))\,\sigma' \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(\sigma',r)} \tag{C.210}$$

By Substitution for Context Interpretation (Lemma 118), we can easily prove Equation C.208.

Furthermore, $\Sigma \vdash \Lambda a.(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))) \longrightarrow^* v$ is already a value, which proves Equation C.209. Now given

$$\forall \sigma', r = \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.211}$$

We need to show

$$(\Lambda a.(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\,\sigma' \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(\sigma',r)} \tag{C.212}$$

Let ${R^{SN}}' = R^{SN}, a \mapsto (\sigma', r)$. By induction hypothesis, we have:

$$\gamma^{SN}(\phi^{SN}(R^{SN}, a \mapsto (\sigma', r)_1(e))) \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(\sigma',r)} \tag{C.213}$$

We know that:

$$(\Lambda a.(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e)))))\,\sigma'$$

$$\longrightarrow [\sigma'/a](\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))$$

$$= \gamma^{SN}(\phi^{SN}(R^{SN}, a \mapsto (\sigma', r)_1(e)))$$

Consequently, by Strong Normalization preserved by forward/backward reduction(Lemma 117), Equation C.213 proves C.212.

$$\boxed{\textbf{rule }\textsc{iTm-forallE}} \qquad \frac{\begin{array}{c}\textsc{iTm-forallE}\\ \Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \forall a.\sigma' \\ \Gamma_C;\Gamma \vdash_{ty} \sigma\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\,\sigma : [\sigma/a]\sigma'}$$

By induction hypothesis, we have:

$$\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))) \in \mathcal{SN}[\![\forall a.\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN}} \tag{C.214}$$

By Substitution for Context Interpretation (Lemma 118), we know:

$$\Gamma_C; \bullet \vdash_{ty} {R^{SN}}_1(\sigma) \tag{C.215}$$

Choose $r = \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$. By the definition of Equation C.214, applying Equation C.215 results in:

$$(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))\,({R^{SN}}_1(\sigma)) \in \mathcal{SN}[\![\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto({R^{SN}}_1(\sigma),r)}$$

By Compositionality for Strong Normalization (Lemma 119), we get:

$$(\gamma^{SN}(\phi^{SN}({R^{SN}}_1(e))))\,({R^{SN}}_1(\sigma)) \in \mathcal{SN}[\![[\sigma/a]\sigma']\!]^{\Sigma,\Gamma_C}_{R^{SN},a\mapsto(\sigma,r)}$$

which is exactly our goal since

$$\gamma^{SN}(\phi^{SN}(R^{SN}{}_1((e\,\sigma)))) = (\gamma^{SN}(\phi^{SN}(R^{SN}{}_1(e))))\,(R^{SN}{}_1(\sigma))$$

□

Theorem 37 (Strong Normalization - Dictionaries - Part B)**.**
If $d \in \mathcal{SN}[\![C]\!]^{\Sigma,\Gamma_C}$ *then* $\exists dv : d \longrightarrow^* dv$*.*

Proof. This goal is baked into the relation. It follows by straightforward case analysis on the hypothesis.

□

Theorem 38 (Strong Normalization - Expressions - Part B)**.**
If $e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$ *then* $\exists v : \Sigma \vdash e \longrightarrow^* v$*.*

Proof. This goal is baked into the relation. It follows by straightforward case analysis on the hypothesis.

□

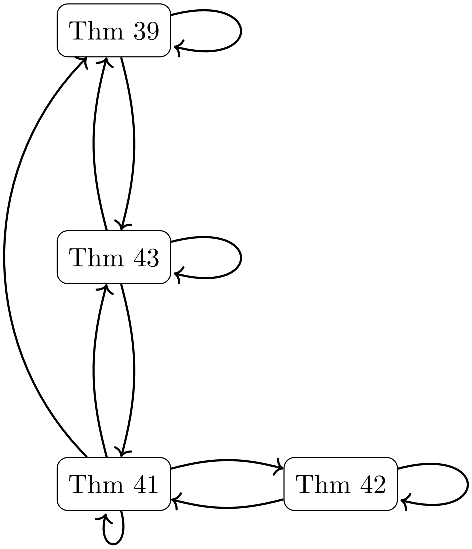

Figure C.3: Dependency graph for Equivalence Theorems

C.6 Elaboration Equivalence Theorems

> **Theorem 39** (Equivalence - Environments). *If* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Proof. By induction on the environment well-formedness relation. This theorem is mutually proven with Theorems 41, 42 and 43 (Figure C.3). Note that at the dependencies between Theorem 39 and 43 and between Theorem 41 and 43, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorems 41 and 42 are proven by induction on a finite derivation. Consequenty, the size of P is strictly decreasing at every cycle and the induction remains well-founded.

$\boxed{\textbf{rule } \text{sCtxT-empty}}$ $\vdash_{ctx} \bullet; \bullet; \bullet \rightsquigarrow \bullet$

The goal follows directly from rule sCtx-empty and rule Ctx-Empty.

$\boxed{\textbf{rule } \text{sCtxT-clsEnv}}$ $\vdash_{ctx} \bullet; \Gamma_C, m : \overline{C}_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau; \bullet \rightsquigarrow \bullet$

The goal to be proven is the following:

$$\vdash^{M}_{ctx} \bullet; \Gamma_C, m : \overline{C}_i \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau; \bullet \rightsquigarrow \Sigma; \Gamma_C; \Gamma \qquad (C.216)$$

$$\Gamma_C; \Gamma \rightsquigarrow \bullet \qquad (C.217)$$

From the rule premise we get that:

$$\Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \rightsquigarrow \sigma \tag{C.218}$$

$$\overline{\Gamma_C; \bullet, a \vdash_C C_i \rightsquigarrow \sigma_i}^{\,i} \tag{C.219}$$

$$\vdash_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet \tag{C.220}$$

By applying the induction hypothesis to Equation C.220, we get:

$$\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet; {\Gamma_C}'; \bullet \tag{C.221}$$

$${\Gamma_C}'; \bullet \rightsquigarrow \bullet \tag{C.222}$$

From rule sCtxT-tyEnvTy, rule sCtx-tyEnvTy and rule Ctx-TVar, in combination with Equation C.220, C.221 and C.222, respectively, we get:

$$\vdash_{ctx} \bullet; \Gamma_C; \bullet, a \rightsquigarrow \bullet, a$$

$$\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet, a \rightsquigarrow \bullet; {\Gamma_C}'; \bullet, a$$

$${\Gamma_C}'; \bullet, a \rightsquigarrow \bullet, a$$

Applying type and constraint equivalence (Theorem 40) to Equations C.218 and C.219, together with these results, gives us:

$$\Gamma_C; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau \rightsquigarrow \sigma \tag{C.223}$$

$${\Gamma_C}'; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.224}$$

$$\overline{\Gamma_C; \bullet, a \vdash^{M}_{C} C_i \rightsquigarrow C_i}^{\,i} \tag{C.225}$$

$$\overline{{\Gamma_C}'; \bullet, a \vdash_C C_i \rightsquigarrow \sigma_i}^{\,i} \tag{C.226}$$

Goal C.216 follows from rule sCtx-clsEnv, in combination with Equations C.221, C.223 and C.225, with $\Sigma = \bullet$, $\Gamma_C = {\Gamma_C}', m : TC\, a : \sigma$ and $\Gamma = \bullet$. Consequently, Goal C.217 follows from rule Ctx-Empty.

rule sCtxT-tyEnvTm $\quad \vdash_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma \rightsquigarrow \Gamma, x : \sigma$

The goal to be proven is the following:

$$\vdash^{M}_{ctx} \bullet; \Gamma_C; \Gamma, x : \sigma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \tag{C.227}$$

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma, x : \sigma \tag{C.228}$$

From the rule premise we get that:

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.229}$$

$$x \notin \mathbf{dom}(\Gamma) \tag{C.230}$$

$$\vdash_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.231}$$

By applying the induction hypothesis to Equation C.231, we get:

$$\vdash^M_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \bullet; \Gamma_C; \Gamma' \tag{C.232}$$

$$\Gamma_C; \Gamma' \rightsquigarrow \Gamma' \tag{C.233}$$

We know from type equivalence (Theorem 40), in combination with Equations C.229, C.232 and C.233, that:

$$\Gamma_C; \Gamma \vdash^M_{ty} \sigma \rightsquigarrow \sigma \tag{C.234}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.235}$$

Goal C.227 follows from rule sCtx-tyEnvTm, in combination with Equations C.230, C.232 and C.234, with $\Sigma = \bullet$ and $\Gamma = \Gamma', x : \sigma$. Consequently, Goal C.228 follows from rule Ctx-Var, in combination with Equations C.233 and C.235, with $\Gamma = \Gamma'$.

$\boxed{\textbf{rule } \text{sCtxT-tyEnvTy}}$ $\vdash_{ctx} \bullet; \Gamma_C; \Gamma, a \rightsquigarrow \Gamma, a$

Similar to the rule sCtxT-tyEnvTm case.

$\boxed{\textbf{rule } \text{sCtxT-tyEnvD}}$ $\vdash_{ctx} \bullet; \Gamma_C; \Gamma, \delta : C \rightsquigarrow \Gamma, \delta : \sigma$

Similar to the rule sCtxT-tyEnvTm case.

$\boxed{\textbf{rule } \text{sCtxT-pgmInst}}$

$$\vdash_{ctx} P, (D : \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}'_y : e; \Gamma_C; \Gamma \rightsquigarrow \Gamma$$

The goal to be proven is the following:

$$\vdash^M_{ctx} P, (D : \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\,\tau).m \mapsto \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}'_y : e; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \tag{C.236}$$

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.237}$$

From the rule premise we get that:

$$\textbf{unambig}(\forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau) \tag{C.238}$$

$$\Gamma_C ; \bullet \vdash_C \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau \rightsquigarrow \forall \overline{b}_j . \overline{\sigma}_i \rightarrow [\sigma / a]\{m : \forall \overline{a}_k . \overline{\sigma}'_y \rightarrow \sigma'\} \tag{C.239}$$

$$(m : \overline{C}'_m \Rightarrow TC\, a : \forall \overline{a}_k . \overline{C}'_y \Rightarrow \tau') \in \Gamma_C \tag{C.240}$$

$$\Gamma_C ; \bullet, a \vdash_{ty} \forall \overline{a}_k . \overline{C}'_y \Rightarrow \tau' \rightsquigarrow \forall \overline{a}_k . \overline{\sigma}'_y \rightarrow \sigma' \tag{C.241}$$

$$\Gamma_C ; \bullet, \overline{b}_j \vdash_{ty} \tau \rightsquigarrow \sigma \tag{C.242}$$

$$P ; \Gamma_C ; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau / a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau / a]\tau' \rightsquigarrow e \tag{C.243}$$

$$D \notin \textbf{dom}(P) \tag{C.244}$$

$$(D' : \forall \overline{b}'_k . \overline{C}''_y \Rightarrow TC\, \tau'').m' \mapsto \Gamma' : e' \notin P \textbf{where} [\overline{\tau}_j / \overline{b}_j]\tau = [\overline{\tau}'_k / \overline{b}'_k]\tau'' \tag{C.245}$$

$$\vdash_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma \tag{C.246}$$

By applying the induction hypothesis to Equation C.246, we get:

$$\vdash^{M}_{ctx} P ; \Gamma_C ; \Gamma \rightsquigarrow \Sigma' ; \Gamma_C ; \Gamma \tag{C.247}$$

$$\Gamma_C ; \Gamma \rightsquigarrow \Gamma \tag{C.248}$$

Goal C.237 follows directly from Equation C.248. From type and constraint equivalence (Theorem 40, the required assumptions follow straightforwardly from rule sCtxT-tyEnvTy, rule sCtx-tyEnvTy and rule Ctx-TVar, in combination with Equations C.246, C.247 and C.248), together with Equations C.239, C.241 and C.242, we know:

$$\Gamma_C ; \bullet \vdash^{M}_{C} \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \tau \rightsquigarrow \forall \overline{b}_j . \overline{C}_i \Rightarrow TC\, \sigma \tag{C.249}$$

$$\Gamma_C ; \bullet, a \vdash^{M}_{ty} \forall \overline{a}_k . \overline{C}'_y \Rightarrow \tau' \rightsquigarrow \forall \overline{a}_k . \overline{C}'_y \Rightarrow \sigma' \tag{C.250}$$

$$\Gamma_C ; \bullet, a \vdash_{ty} \forall \overline{a}_k . \overline{C}'_y \Rightarrow \sigma' \rightsquigarrow \forall \overline{a}_k . \overline{\sigma}'_y \rightarrow \sigma' \tag{C.251}$$

$$\Gamma_C ; \bullet, \overline{b}_j \vdash^{M}_{ty} \tau \rightsquigarrow \sigma \tag{C.252}$$

$$\Gamma_C ; \bullet, \overline{b}_j \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.253}$$

Similarly, from expression equivalence (Theorem 43, the environment well-formedness assumption is constructed straightforwardly), together with

Equation C.243, we get:

$$P; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}'_y \vdash^M_{tm} e \Leftarrow [\tau/a]\tau' \rightsquigarrow e \tag{C.254}$$

Goal C.236 follows from rule sCtx-pgmInst, in combination with Equations C.238, C.249, C.240, C.254, C.250, C.244, C.245 and C.247, and with $\Sigma = \Sigma', (D : \forall \overline{b}_j.\overline{C}_i \Rightarrow TC\,\sigma').m \mapsto \Lambda\overline{b}_j.\lambda\overline{\delta}_i : \overline{C}_i.\Lambda\overline{a}_k.\lambda\overline{\delta}_y : [\sigma/a]\overline{C}'_y.e$. □

> **Theorem 40** (Equivalence - Types and Constraints)**.**
>
> - *If* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma_C; \Gamma \vdash^M_{ty} \sigma \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$*.*
>
> - *If* $\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$ *and* $\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma$*.*
>
> - *If* $\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma_C; \Gamma \vdash^M_C C \rightsquigarrow C$ *and* $\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma$*.*

Proof. By induction on the size of the type σ, class constraint Q or constraint C.

Part 1 By case analysis on the type well-formedness derivation.

$\boxed{\textbf{rule } \textsc{sTyT-bool}}$ $\Gamma_C; \Gamma \vdash_{ty} Bool \rightsquigarrow Bool$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash^M_{ty} Bool \rightsquigarrow \sigma \tag{C.255}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow Bool \tag{C.256}$$

Goals C.255 and C.256 follow directly from rule sTy-bool and rule iTy-bool respectively, with $\sigma = Bool$.

$\boxed{\textbf{rule } \textsc{sTyT-var}}$ $\Gamma_C; \Gamma \vdash_{ty} a \rightsquigarrow a$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash^M_{ty} a \rightsquigarrow \sigma \tag{C.257}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow a \tag{C.258}$$

From the rule premise we get that:

$$a \in \Gamma \tag{C.259}$$

By applying Lemmas 76 and 111 to Equation C.259, we get:

$$a \in \Gamma \tag{C.260}$$

$$a \in \Gamma \tag{C.261}$$

Goal C.257 and C.258 follow directly from rule sTy-var and rule iTy-var respectively, in combination with Equations C.260 and C.261, with $\sigma = a$.

$\boxed{\textbf{rule } \text{sTyT-arrow}}$ $\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2$

The goal to be proven is the following:

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} \tau_1 \to \tau_2 \rightsquigarrow \sigma \tag{C.262}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma_1 \to \sigma_2 \tag{C.263}$$

From the rule premise we get that:

$$\Gamma_C ; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma_1 \tag{C.264}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.265}$$

By applying the induction hypothesis on Equations C.264 and C.265, we get:

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} \tau_1 \rightsquigarrow \sigma_1 \tag{C.266}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.267}$$

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} \tau_2 \rightsquigarrow \sigma_2 \tag{C.268}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2 \tag{C.269}$$

Goals C.262 and C.263 follow directly from rule sTy-arrow and rule iTy-arrow respectively, in combination with Equations C.266, C.267, C.268 and C.269, with $\sigma = \sigma_1 \to \sigma_2$.

$\boxed{\textbf{rule } \text{sTyT-qual}}$ $\Gamma_C ; \Gamma \vdash_{ty} C \Rightarrow \rho \rightsquigarrow \sigma_1 \to \sigma_2$

The goal to be proven is the following:

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} C \Rightarrow \rho \rightsquigarrow \sigma \tag{C.270}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma_1 \to \sigma_2 \tag{C.271}$$

From the rule premise we get that:

$$\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_1 \tag{C.272}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \rho \rightsquigarrow \sigma_2 \tag{C.273}$$

By applying the induction hypothesis on Equation C.273, we get:

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} \rho \rightsquigarrow \sigma_2 \tag{C.274}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2 \tag{C.275}$$

By applying Part 3 of this theorem on Equation C.272, we get:

$$\Gamma_C ; \Gamma \vdash_C^{M} C \rightsquigarrow C \tag{C.276}$$

$$\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_1 \tag{C.277}$$

Goals C.270 and C.271 follow directly from rule sTy-qual and rule iTy-qual respectively, in combination with Equations C.274, C.275, C.276 and C.277, with $\sigma = C \Rightarrow \sigma_2$.

$\boxed{\textbf{rule } \text{sTyT-scheme}}$ $\quad \Gamma_C ; \Gamma \vdash_{ty} \forall a . \sigma \rightsquigarrow \forall a . \sigma$

The goal to be proven is the following:

$$\Gamma_C ; \Gamma \vdash_{ty}^{M} \forall a . \sigma \rightsquigarrow \sigma \tag{C.278}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \forall a . \sigma \tag{C.279}$$

From the rule premise we get that:

$$a \notin \Gamma \tag{C.280}$$

$$\Gamma_C ; \Gamma , a \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.281}$$

By repeated case analysis on the 2nd hypothesis (rule sCtxT-pgmInst), we get:

$$\vdash_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma \tag{C.282}$$

From rule sCtxT-tyEnvTy, in combination with Equations C.282 and C.280, we know that:

$$\vdash_{ctx} \bullet ; \Gamma_C ; \Gamma , a \rightsquigarrow \Gamma , a \tag{C.283}$$

Similarly, we get from rule sCtx-tyEnvTy and rule ctx-TVar that:

$$\vdash_{ctx}^{M} \bullet ; \Gamma_C ; \Gamma , a \rightsquigarrow \bullet ; \Gamma_C ; \Gamma , a \tag{C.284}$$

$$\Gamma_C ; \Gamma , a \rightsquigarrow \Gamma , a \tag{C.285}$$

By applying the induction hypothesis on Equation C.281, together with Equations C.283, C.284 and C.285, we get:

$$\Gamma_C ; \Gamma, a \vdash^{M}_{ty} \sigma \rightsquigarrow \sigma' \tag{C.286}$$

$$\Gamma_C ; \Gamma, a \vdash_{ty} \sigma' \rightsquigarrow \sigma \tag{C.287}$$

Goals C.278 and C.279 follow directly from rule sTy-scheme and rule iTy-scheme respectively, in combination with Equations C.280, C.286 and C.287, with $\sigma = \forall a.\sigma'$.

Part 2 By case analysis on the class constraint well-formedness derivation.

$\boxed{\textbf{rule } \text{sQT-TC}}$ $\quad \Gamma_C ; \Gamma \vdash_Q TC\,\tau \rightsquigarrow [\sigma'/a]\{m : \sigma\}$

The goal to be proven is the following:

$$\Gamma_C ; \Gamma \vdash^{M}_{Q} TC\,\tau \rightsquigarrow Q \tag{C.288}$$

$$\Gamma_C ; \Gamma \vdash_Q Q \rightsquigarrow [\sigma'/a]\{m : \sigma\} \tag{C.289}$$

From the rule premise we get that:

$$\Gamma_C ; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma' \tag{C.290}$$

$$\Gamma_C = \Gamma_{C1}, m : \overline{C}_i \Rightarrow TC\,a : \sigma, \Gamma_{C2} \tag{C.291}$$

$$\Gamma_{C1} ; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.292}$$

By repeated case analysis on the 2nd hypothesis (rule sCtxT-pgmInst), we get:

$$\vdash_{ctx} \bullet ; \Gamma_C ; \Gamma \rightsquigarrow \Gamma \tag{C.293}$$

From rule sCtxT-tyEnvTy, together with Equation C.293 and the fact that $a \notin \bullet$, we know that:

$$\vdash_{ctx} \bullet ; \Gamma_{C1} ; \bullet, a \rightsquigarrow \bullet, a \tag{C.294}$$

Similarly, we get from rule sCtx-tyEnvTy and rule ctx-TVar that:

$$\vdash^{M}_{ctx} \bullet ; \Gamma_{C1} ; \bullet, a \rightsquigarrow \bullet ; \Gamma_{C1} ; \bullet, a \tag{C.295}$$

$$\Gamma_{C1} ; \bullet, a \rightsquigarrow \Gamma, a \tag{C.296}$$

By applying Part 1 of this theorem to Equations C.290 and C.292, together with Equations C.294, C.295 and C.296, we get:

$$\Gamma_C ; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma' \tag{C.297}$$

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma' \rightsquigarrow \sigma' \tag{C.298}$$

$$\Gamma_{C1} ; \bullet, a \vdash^M_{ty} \sigma \rightsquigarrow \sigma \tag{C.299}$$

$$\Gamma_{C1} ; \bullet, a \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.300}$$

Goal C.288 follows from rule sQ-TC, together with Equations C.297, C.291 and C.299, with $Q = TC\,\sigma'$. Consequently, Goal C.289 follows from rule iQ-TC, together with Equations C.298, C.291 and C.300.

Part 3 By case analysis on the constraint well-formedness derivation.

$\boxed{\textbf{rule } \text{sCT-forall}}$ $\quad \Gamma_C ; \Gamma \vdash_C \forall a.C \rightsquigarrow \forall a.\sigma$

The goal to be proven is thus the following

$$\Gamma_C ; \Gamma \vdash^M_C \forall a.C \rightsquigarrow C \tag{C.301}$$

$$\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \forall a.\sigma \tag{C.302}$$

We know from the rule premise

$$\Gamma_C ; \Gamma, a \vdash_C C \rightsquigarrow \sigma$$

Using Lemma 66 and rule sCtx-tyEnvTy, rule sCtxT-tyEnvTy and rule Ctx-TVar we can derive that

$$\vdash_{ctx} P ; \Gamma_C ; \Gamma, a \rightsquigarrow \Gamma, a$$

$$\vdash^M_{ctx} P ; \Gamma_C ; \Gamma, a \rightsquigarrow \Sigma ; \Gamma_C ; \Gamma, a$$

$$\Gamma_C ; \Gamma, a \rightsquigarrow \Gamma, a$$

Using this result, we derive from the induction hypothesis that

$$\Gamma_C ; \Gamma, a \vdash^M_C C \rightsquigarrow C'$$

$$\Gamma_C ; \Gamma, a \vdash_C C' \rightsquigarrow \sigma$$

Goals C.301 and C.302 follow by rule sC-forall and rule iC-forall respectively.

$\boxed{\textbf{rule } \text{sCT-ARROW}}$ $\quad \Gamma_C; \Gamma \vdash_C C_1 \Rightarrow C_2 \rightsquigarrow \sigma_1 \to \sigma_2$

The goal to be proven is thus the following

$$\Gamma_C; \Gamma \vdash^M_C C_1 \Rightarrow C_2 \rightsquigarrow C \tag{C.303}$$

$$\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma_1 \to \sigma_2 \tag{C.304}$$

We know from the rule premise

$$\Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1$$

$$\Gamma_C; \Gamma \vdash_C C_2 \rightsquigarrow \sigma_2$$

Applying the induction hypothesis gives us

$$\Gamma_C; \Gamma \vdash^M_C C_1 \rightsquigarrow C_1$$

$$\Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1$$

$$\Gamma_C; \Gamma \vdash^M_C C_2 \rightsquigarrow C_2$$

$$\Gamma_C; \Gamma \vdash_C C_2 \rightsquigarrow \sigma_2$$

Goals C.303 and C.304 follow by rule sC-arrow and rule iC-arrow respectively.

$\boxed{\textbf{rule } \text{sCT-classConstr}}$ $\quad \Gamma_C; \Gamma \vdash_C Q \rightsquigarrow \sigma$

The goal to be proven is thus the following

$$\Gamma_C; \Gamma \vdash^M_C Q \rightsquigarrow C$$

$$\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma$$

As we know from the rule premise

$$\Gamma_C; \Gamma \vdash_Q Q \rightsquigarrow \sigma$$

the goal follows directly from Part 2 of this theorem.

□

Theorem 41 (Equivalence - Constraint Entailment)**.**
If $P; \Gamma_C; \Gamma \vDash [C] \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma \vDash^M [C] \rightsquigarrow d$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d d : C \rightsquigarrow e$
where $\Gamma_C; \Gamma \vdash^M_C C \rightsquigarrow C$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Proof. By induction on the entailment derivation, and mutually proven with Theorems 39, 42 and 43 (Figure C.3). Note that at the dependencies between Theorem 39 and 43 and between Theorem 41 and 43, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorems 41 and 42 are proven by induction on a finite derivation. Consequenty, the size of P is strictly decreasing at every cycle and the induction remains well-founded.

From environment equivalence (Theorem 39), in combination with the 2nd hypothesis, we derive that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \tag{C.305}$$

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.306}$$

$\boxed{\textbf{rule } \textsc{sEntailT-arrow}}$ $\quad P; \Gamma_C; \Gamma \vDash [C_1 \Rightarrow C_2] \rightsquigarrow \lambda\delta_1 : \sigma_1.e$
The goal to be proven becomes

$$P; \Gamma_C; \Gamma \vDash^{M} [C_1 \Rightarrow C_2] \rightsquigarrow d \tag{C.307}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda\delta_1 : \sigma_1.e \tag{C.308}$$

$$\Gamma_C; \Gamma \vdash^{M}_{C} C_1 \Rightarrow C_2 \rightsquigarrow C_1 \Rightarrow C_2 \tag{C.309}$$

We know from the rule premise that

$$P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash [C_2] \rightsquigarrow e$$

$$\Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1$$

It follows from rule sCtxT-tyEnvD that

$$\vdash_{ctx} P; \Gamma_C; \Gamma, \delta_1 : C_1 \rightsquigarrow \Gamma, \delta_1 : \sigma_1$$

We then get from the induction hypothesis

$$P; \Gamma_C; \Gamma, \delta_1 : C_1 \vDash^{M} [C_2] \rightsquigarrow d'$$

$$\Sigma; \Gamma_C; \Gamma, \delta_1 : C_1 \vdash_d d' : C_2 \rightsquigarrow e$$

$$\Gamma_C; \Gamma, \delta_1 : C_1 \vdash^{M}_{C} C_2 \rightsquigarrow C_2$$

Goal C.307 follows by rule sEntail-arrow with $d = \lambda\delta_1 : C_1.d'$. Goal C.308 can then be proven using rule D-dabs. Goal C.309 follows from rule sC-arrow (note that we can applying strengthening, as constraint well-formedness is never impacted by dictionary variables in the context).

$\boxed{\textbf{rule } \textsc{sEntailT-forall}}$ $P; \Gamma_C; \Gamma \vDash [\forall a.C] \rightsquigarrow \Lambda a.e$

The goal to be proven is

$$P; \Gamma_C; \Gamma \vDash^M [\forall a.C] \rightsquigarrow d \tag{C.310}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow \Lambda a.e \tag{C.311}$$

$$\Gamma_C; \Gamma \vdash_C^M \forall a.C \rightsquigarrow \forall a.C \tag{C.312}$$

We know from the rule premise that

$$P; \Gamma_C; \Gamma, a \vDash [C] \rightsquigarrow e$$

It follows from rule $\textsc{sCtxT-tyEnvTy}$ that

$$\vdash_{ctx} P; \Gamma_C; \Gamma, a \rightsquigarrow \Gamma, a$$

We then get from the induction hypothesis

$$P; \Gamma_C; \Gamma, a \vDash^M [C] \rightsquigarrow d'$$

$$\Sigma; \Gamma_C; \Gamma, a \vdash_d d' : C \rightsquigarrow e$$

$$\Gamma_C; \Gamma, a \vdash_C^M C \rightsquigarrow C$$

Goal C.310 follows from rule $\textsc{sEntail-forall}$ with $d = \Lambda a.d'$. Goal C.311 then follows by rule $\textsc{D-tyAbs}$. Goal C.312 follows from rule $\textsc{sC-forall}$.

$\boxed{\textbf{rule } \textsc{sEntailT-inst}}$ $P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e$

We know from the premise that

$$P = P_1, (D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q').m \mapsto \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y : e, P_2 \tag{C.313}$$

$$P_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_0 \tag{C.314}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C'_i \rightsquigarrow \sigma'_i}^{\,i} \tag{C.315}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_C C_y \rightsquigarrow \sigma''_y}^{\,y} \tag{C.316}$$

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \Lambda \overline{a}_j . \lambda \overline{\delta'_i : \sigma'_i}^{\,i} . \{m = \Lambda \overline{b}_k . \lambda \overline{\delta_y : \sigma''_y}^{\,y} . e_0\} : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q'] \vDash Q \rightsquigarrow \overline{\tau} \vdash e \tag{C.317}$$

By inversion on Equation C.305 and C.313, we can conclude that

$$\Sigma = \Sigma_1, (D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q').m \mapsto \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}'_i . \Lambda \overline{b}_k . \lambda \overline{\delta}_y : \overline{C}_y . e, \Sigma_2 \tag{C.318}$$

From Preservation Thoerem 28 and Equation C.305 we get

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.319}$$

By inversion on Equation C.319 (rule iCtx-MEnv), in combination with Equation C.318, we know that

$$(m : TC\, a : \sigma') \in \Gamma_C \tag{C.320}$$

Applying Theorem 40 on Equations C.315 and C.316 gives us

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C^M C'_i \rightsquigarrow C'_i}^{\,i} \tag{C.321}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C'_i \rightsquigarrow \sigma'_i}^{\,i} \tag{C.322}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_C^M C_y \rightsquigarrow C''_y}^{\,y} \tag{C.323}$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_C C''_y \rightsquigarrow \sigma''_y}^{\,y} \tag{C.324}$$

From Expression Equivalence Theorem 43 and Equation C.314 we get

$$P_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y \vdash_{tm}^M e \Rightarrow \tau \rightsquigarrow e$$

$$\Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}''_y \vdash_{tm} e : \sigma \rightsquigarrow e$$

where $\Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i, \overline{b}_k, \overline{\delta}_y : \overline{C}_y \;\; \vdash_{ty}^M \;\; \tau \rightsquigarrow \sigma$. Using rule iTm-constrI and rule iTm-forallI, we can easily derive from this result that

$$\Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}'_i \vdash_{tm} \Lambda \overline{b}_k . \lambda \overline{\delta}_y : \overline{C}''_y . e : \forall \overline{b}_k . \overline{C}_y \Rightarrow \sigma \rightsquigarrow \Lambda \overline{b}_k . \lambda \overline{\delta_y : \sigma''_y}^{\,y} . e \tag{C.325}$$

It now follows by rule D-con, in combination with Equations C.318, C.320, C.319, C.321 and C.325 that

$$\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q' \rightsquigarrow \Lambda \overline{a}_j . \lambda \overline{\delta'_i : \sigma'_i}^{\,i} . \{m = \Lambda \overline{b}_k . \lambda \overline{\delta_y : \sigma''_y}^{\,y} . e_0\}$$

We then apply Theorem 42 to this result, together with Equation C.317. This gives us

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash D : \forall \overline{a}_j . \overline{C}'_i \Rightarrow Q'] \vDash^M Q \rightsquigarrow \bullet \vdash d$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e$$

$$\Gamma_C; \Gamma \vdash_Q^M Q \rightsquigarrow Q$$

The goal thus follows by this result, in combination with rule sENTAIL-INST.

$\boxed{\textbf{rule } \textsc{sEntailT-local}}$ $P; \Gamma_C; \Gamma \vDash [Q] \rightsquigarrow e$

We know from the rule premise that

$$(\delta : C) \in \Gamma \tag{C.326}$$

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta : C] \vDash Q \rightsquigarrow \bullet \vdash e \tag{C.327}$$

From Preservation Thoerem 28 and Equation C.305 we get

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.328}$$

By Lemma 77 and Equation C.326 we know that $(\delta : C) \in \Gamma$ where $\Gamma_C; \Gamma \vdash_C^M C \rightsquigarrow C$. Applying rule D-VAR to this result and Equation C.328 gives us

$$\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C \rightsquigarrow \delta$$

We can then apply Theorem 42 to this result, together with Equation C.326 to get

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta : C] \vDash^M Q \rightsquigarrow \bullet \vdash d$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e$$

$$\Gamma_C; \Gamma \vdash_Q^M Q \rightsquigarrow Q$$

The goal thus follows by this result, in combination with rule sENTAIL-LOCAL.

□

Theorem 42 (Equivalence - Constraint Matching).
If $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : C_0] \vDash Q_1 \rightsquigarrow \overline{\tau} \vdash e_1$
and $\Sigma; \Gamma_C; \Gamma, \overline{a}, \overline{\delta} : \overline{C} \vdash_d d_0 : C_0 \rightsquigarrow e_0$
where $\Gamma_C; \Gamma, \overline{a} \vdash_C^M C_0 \rightsquigarrow C_0$ *and* $\overline{\Gamma_C; \Gamma, \overline{a} \vdash_C^M C_i \rightsquigarrow C_i}^{\,i}$
and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx}^M P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : C_0] \vDash^M Q_1 \rightsquigarrow \overline{\tau} \vdash d_1$
and $\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d d_1 : Q_1 \rightsquigarrow e_1$ *where* $\Gamma_C; \Gamma \vdash_Q^M Q_1 \rightsquigarrow Q_1$.

Proof. By induction on the constraint matching derivation. This theorem is mutually proven with Theorems 39, 41 and 43 (Figure C.3). Note that at the dependencies between Theorem 39 and 43 and between Theorem 41 and 43, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorems 41 and 42 are proven by induction on a finite derivation. Consequenty, the size of P is strictly decreasing at every cycle and the induction remains well-founded.

$\boxed{\textbf{rule}\ \textsc{sMatchT-arrow}}$ $\quad P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C} \vdash e_0 : C_1 \Rightarrow C_2] \vDash Q_1 \rightsquigarrow \overline{\tau} \vdash [e_1/\delta_1]e_2$
We know from the rule premise that

$$P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C},\delta_1 : C_1 \vdash e_0\ \delta_1 : C_2] \vDash Q \rightsquigarrow \overline{\tau} \vdash e_2 \tag{C.329}$$

$$P;\Gamma_C;\Gamma \vDash [[\overline{\tau}/\overline{a}]C_1] \rightsquigarrow e_1 \tag{C.330}$$

We know from the 2nd premise that $\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C} \vdash_d d_0 : C_1 \Rightarrow C_2 \rightsquigarrow e_0$. By rule D-var we can easily derive that $\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C},\delta_1 : C_1 \vdash_d \delta_1 : C_1 \rightsquigarrow \delta_1$. By rule D-dapp (in combination with Weakening Lemma 96) we get that

$$\Sigma;\Gamma_C;\Gamma,\overline{a},\overline{\delta}:\overline{C},\delta_1 : C_1 \vdash_d d_0\,\delta_1 : C_2 \rightsquigarrow e_0\,\delta_1$$

Furthermore, as we know from the 3rd premise that $\Gamma_C;\Gamma,\overline{a} \vdash_C^M C_1 \Rightarrow C_2 \rightsquigarrow C_1 \Rightarrow C_2$, we can derive by inversion (rule sC-arrow) that $\Gamma_C;\Gamma,\overline{a} \vdash_C^M C_2 \rightsquigarrow C_2$. Using these results, we can apply the induction hypothesis to Equation C.330 to obtain

$$P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C},\delta_1 : C_1 \vdash d_0\,\delta_1 : C_2] \vDash^M Q_1 \rightsquigarrow \overline{\tau} \vdash d_2 \tag{C.331}$$

$$\Sigma;\Gamma_C;\Gamma,\overline{\delta}:[\overline{\sigma}/\overline{a}]\overline{C},\delta_1 : [\overline{\sigma}/\overline{a}]C_1 \vdash_d d_2 : Q_1 \rightsquigarrow e_2 \tag{C.332}$$

$$\Gamma_C;\Gamma \vdash_Q^M Q_1 \rightsquigarrow Q_1 \tag{C.333}$$

By Theorem 41 in combination with the 2nd rule premise we obtain

$$P;\Gamma_C;\Gamma \vDash^M [[\overline{\tau}/\overline{a}]C_1] \rightsquigarrow d_1 \tag{C.334}$$

$$\Sigma;\Gamma_C;\Gamma \vdash_d d_1 : [\overline{\sigma}/\overline{a}]C_1 \rightsquigarrow e_1 \tag{C.335}$$

$$\Gamma_C;\Gamma \vdash_C^M [\overline{\tau}/\overline{a}]C_1 \rightsquigarrow [\overline{\sigma}/\overline{a}]C_1 \tag{C.336}$$

Using rule sMatch-arrow, together with Equations C.330 and C.331, we derive

$$P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C} \vdash d_0 : C_1 \Rightarrow C_2] \vDash^M Q_1 \rightsquigarrow \overline{\tau} \vdash [d_1/\delta_1]d_2$$

Applying Weakening Lemma 96 and Substitution Lemma 91 on Equations C.332 and C.335, gives us

$$\Sigma;\Gamma_C;\Gamma,\overline{\delta}:[\overline{\sigma}/\overline{a}]\overline{C} \vdash_d [d_1/\delta_1]d_2 : Q_1 \rightsquigarrow [e_1/\delta_1]e_2$$

The goal follows directly from these results.
$\boxed{\textbf{rule}\ \textsc{sMatchT-forall}}$ $\quad P;\Gamma_C;\Gamma;[\overline{a};\overline{\delta}:\overline{C} \vdash e_0 : \forall a.C] \vDash Q_1 \rightsquigarrow \overline{\tau} \vdash e_1$
We know from the rule premise that

$$P;\Gamma_C;\Gamma;[\overline{a},a;\overline{\delta}:\overline{C} \vdash e_0\ a : C] \vDash Q_1 \rightsquigarrow \overline{\tau},\tau \vdash e_1$$

As we know from the 2nd premise that $\Sigma; \Gamma_C; \Gamma, \overline{a} \vdash_d d_0 : \forall a. C \rightsquigarrow e_0$, we can show through rule D-TYAPP that

$$\Sigma; \Gamma_C; \Gamma, \overline{a}, a \vdash_d d_0\, a : C \rightsquigarrow e_0\, a$$

By applying the induction hypothesis to these results, we get

$$P; \Gamma_C; \Gamma; [\overline{a}, a; \overline{\delta} : \overline{C} \vdash d_0\, a : C] \models^M Q_1 \rightsquigarrow \overline{\tau}, \tau \vdash d_1$$

$$\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d d_1 : Q_1 \rightsquigarrow e_1$$

$$\Gamma_C; \Gamma \vdash^M_Q Q_1 \rightsquigarrow Q_1$$

The goal follows directly from this result using rule sMatch-forall.

rule sMatchT-classConstr $P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash e_0 : TC\,\tau_0] \models TC\,\tau_1 \rightsquigarrow \overline{\tau} \vdash e_1$

The goal to be proven becomes

$$P; \Gamma_C; \Gamma; [\overline{a}; \overline{\delta} : \overline{C} \vdash d_0 : TC\,\tau_0] \models^M TC\,\tau_1 \rightsquigarrow \overline{\tau} \vdash d_1 \tag{C.337}$$

$$\Sigma; \Gamma_C; \Gamma, \overline{\delta} : [\overline{\sigma}/\overline{a}]\overline{C} \vdash_d d_1 : TC\,\sigma_1 \rightsquigarrow e_1 \tag{C.338}$$

$$\Gamma_C; \Gamma \vdash^M_Q TC\,\tau_1 \rightsquigarrow TC\,\sigma_1 \tag{C.339}$$

We know from the rule premise that

$$\tau_1 = [\overline{\tau}/\overline{a}]\tau_0 \tag{C.340}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \tau_i \rightsquigarrow \sigma_i}^{\,i} \tag{C.341}$$

$$e_1 = [\overline{\sigma}/\overline{a}]e_0 \tag{C.342}$$

Goal C.337 follows by rule sMatch-classConstr, in combination with Theorem 40 and Equation C.341, with $d_1 = [\overline{\sigma}/\overline{a}]d_0$. Goal C.338 now follows from the 2nd hypothesis ($\Sigma; \Gamma_C; \Gamma, \overline{a}, \overline{\delta} : \overline{C} \vdash_d d_0 : TC\,\sigma_0 \rightsquigarrow e_0$) in combination with Lemma 93. As we know from the 3rd hypothesis that $\Gamma_C; \Gamma, \overline{a} \vdash^M_C TC\,\tau_0 \rightsquigarrow TC\,\sigma_0$, Goal C.339 follows by Lemma 61 and rule sC-classConstr.

□

Theorem 43 (Equivalence - Expressions)**.**

- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$ *where* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$*.*

- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$
where $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$.

Proof. By induction on the lexicographic order of the tuple (size of the expression e, typing mode). Regarding typing mode, we define type checking to be larger than type inference. In each mutual dependency, we know that the tuple size decreases, meaning that the induction is well-founded.

Furthermore, this theorem is mutually proven with Theorems 39, 41 and 42 (Figure C.3). Note that at the dependencies between Theorem 39 and 43 and between Theorem 41 and 43, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Furthermore, Theorems 41 and 42 are proven by induction on a finite derivation. Consequenty, the size of P is strictly decreasing at every cycle and the induction remains well-founded.

From environment equivalence (Theorem 39), in combination with the 2nd hypothesis, we derive that:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \tag{C.343}$$

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.344}$$

Consequently, by Theorem 28 we derive that

$$\vdash_{ctx} \Sigma; \Gamma_C; \Gamma \tag{C.345}$$

Part 1 By case analysis on the typing derivation.

$\boxed{\textbf{rule } \textsc{sTm-infT-true}}$ $P; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}$

The goal follows by rule sTm-inf-true, rule iTm-true (in combination with Equation C.345) and rule sTy-bool, with $e = \mathit{True}$.

$\boxed{\textbf{rule } \textsc{sTm-infT-false}}$ $P; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{False}$

Similar to the sTm-infT-true case.

$\boxed{\textbf{rule } \textsc{sTm-infT-let}}$

$$P; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1 \textbf{ in } e_2 \Rightarrow \tau_2 \rightsquigarrow e_3$$

where $e_3 = \textbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma = \Lambda \overline{a}_j.\lambda \overline{\delta_k : \sigma_k}^k .e_1\, \textbf{in}\ \ e_2$. The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash^M_{tm} \textbf{let}\ \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1\, \textbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e \tag{C.346}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma_2 \rightsquigarrow \textbf{let}\ \ x : \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma = \Lambda \overline{a}_j.\lambda \overline{\delta_k : \sigma_k}^k .e_1\, \textbf{in}\ \ e_2 \tag{C.347}$$

$$\Gamma_C; \Gamma \vdash^M_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.348}$$

From the rule premise we know that:

$$x \notin \textbf{dom}(\Gamma) \tag{C.349}$$

$$\textbf{unambig}(\forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1) \tag{C.350}$$

$$\textbf{closure}(\Gamma_C; \overline{C}_i) = \overline{C}_k \tag{C.351}$$

$$\overline{\Gamma_C; \Gamma \vdash_C C_k \rightsquigarrow \sigma_k}^k \tag{C.352}$$

$$\Gamma_C; \Gamma \vdash_{ty} \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma \tag{C.353}$$

$$\overline{\delta}_k\ \textbf{fresh} \tag{C.354}$$

$$P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \tag{C.355}$$

$$P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \tag{C.356}$$

By applying Lemma 74 to Equation C.356, we get that:

$$\Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \vdash^M_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.357}$$

It is straightforward to see from the definition of type well-formedness, that Goal C.348 follows from Equation C.357, since term variables in the environment are not relevant for type well-formedness.

We know from the hypothesis that $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$. By repeated case analysis on this result (rule sCtxT-pgmInst), we get that $\vdash_{ctx} \bullet; \Gamma_C; \Gamma \rightsquigarrow \Gamma$. From rule sCtxT-tyEnvTm, rule sCtxT-tyEnvTy and rule sCtxT-tyEnvD, in combination with Equations C.349, C.352, C.353 and C.354, we know that:

$$\vdash_{ctx} P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \rightsquigarrow \Gamma, \overline{a}_j, \overline{\delta_k : \sigma_k}^k \tag{C.358}$$

$$\vdash_{ctx} P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \Gamma, x : \forall \overline{a}_j.\overline{\sigma}_k \rightarrow \sigma \tag{C.359}$$

Applying the induction hypothesis on Equations C.355 and C.356, in combination with Equations C.358 and C.359, results in:

$$P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash^M_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \tag{C.360}$$

$$\Sigma; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \tag{C.361}$$

$$P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \vdash^M_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \tag{C.362}$$

$$\Sigma; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \tag{C.363}$$

From constraint equivalence (Theorem 40), in combination with Equation C.352, we get:

$$\overline{\Gamma_C; \Gamma \vdash_C C_k \rightsquigarrow \sigma_k}^{k} \tag{C.364}$$

By applying rule ITM-FORALLI and rule ITM-CONSTRI to Equation C.361, together with Equation C.364, we get:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1 : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma_1 \rightsquigarrow \Lambda \overline{a}_j . \lambda\, \overline{\delta_k : \sigma_k}^{k} . e_1 \tag{C.365}$$

From Lemma 112, in combination with Equation C.365, we know that:

$$\Gamma_C; \Gamma \vdash_{ty} \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma_1 \rightsquigarrow \forall \overline{a}_j . \overline{\sigma}_k \rightarrow \sigma \tag{C.366}$$

Goals C.346 and C.347 follow from rule STM-INF-LET and rule ITM-LET respectively, with

$$e = \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma_1 = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1\ \textbf{in}\ \ e_2$$

$\boxed{\textbf{rule } \text{STM-INFT-ARRE}}$ $\quad P; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e \tag{C.367}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma_2 \rightsquigarrow e_1\, e_2 \tag{C.368}$$

$$\Gamma_C; \Gamma \vdash^M_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.369}$$

From the rule premise we know that:

$$P; \Gamma_C; \Gamma \vdash_{tm} e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1 \tag{C.370}$$

$$P; \Gamma_C; \Gamma \vdash_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \tag{C.371}$$

By applying the induction hypothesis to Equations C.370 and C.371, we get:

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \tag{C.372}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1 \tag{C.373}$$

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \tag{C.374}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2 \tag{C.375}$$

Goals C.367 and C.368 follow from rule sTm-inf-ArrE and rule iTm-arrE respectively, in combination with Equations C.372, C.373, C.374 and C.375. Goal C.369 follows by applying Lemma 74 to Equation C.367.

$\boxed{\textbf{rule } \text{sTm-infT-Ann}}$ $\quad P; \Gamma_C; \Gamma \vdash_{tm} e :: \tau \Rightarrow \tau \rightsquigarrow e$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e \tag{C.376}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e \tag{C.377}$$

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \tag{C.378}$$

From the rule premise we know that:

$$P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e \tag{C.379}$$

Goals C.376, C.377 and C.378 follow by applying Part 2 of this theorem to Equation C.379.

Part 2 By case analysis on the typing derivation.

$\boxed{\textbf{rule } \text{sTm-checkT-var}}$ $\quad P; \Gamma_C; \Gamma \vdash_{tm} x \Leftarrow [\overline{\tau}_j / \overline{a}_j]\tau \rightsquigarrow x\, \overline{\sigma}_j\, \overline{e}_i$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j / \overline{a}_j]\tau \rightsquigarrow e \tag{C.380}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma' \rightsquigarrow x\, \overline{\sigma}_j\, \overline{e}_i \tag{C.381}$$

$$\Gamma_C; \Gamma \vdash_{ty}^{M} [\overline{\tau}_j / \overline{a}_j]\tau \rightsquigarrow \sigma' \tag{C.382}$$

From the rule premise we know that:

$$(x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \in \Gamma \tag{C.383}$$

$$\textbf{unambig}(\forall \overline{a}_j . \overline{C}_i \Rightarrow \tau) \tag{C.384}$$

$$\overline{P; \Gamma_C; \Gamma \models [[\overline{\tau}_j / \overline{a}_j] C_i] \rightsquigarrow e_i}^{\,i} \tag{C.385}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \tag{C.386}$$

$$\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.387}$$

We know from Lemma 75, in combination with Equations C.383 and C.343, that:

$$(x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma) \in \Gamma \tag{C.388}$$

By applying type equivalence (Theorem 40) to Equation C.386, we get:

$$\overline{\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma_j}^{\,j} \tag{C.389}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j}^{\,j} \tag{C.390}$$

By applying dictionary equivalence (Theorem 41) to Equation C.385, we get:

$$\overline{P; \Gamma_C; \Gamma \models^{M} [[\overline{\tau}_j / \overline{a}_j] C_i] \rightsquigarrow d_i}^{\,i} \tag{C.391}$$

$$\overline{\Sigma; \Gamma_C; \Gamma \vdash_{d} d_i : [\overline{\sigma}_j / \overline{a}_j] C_i \rightsquigarrow e_i}^{\,i} \tag{C.392}$$

$$\overline{\Gamma_C; \Gamma \vdash^{M}_{C} [\overline{\tau}_j / \overline{a}_j] C_i \rightsquigarrow [\overline{\sigma}_j / \overline{a}_j] C_i}^{\,i} \tag{C.393}$$

Goal C.380 follows from rule sTm-check-var, in combination with Equations C.383, C.384, C.391, C.389 and C.343, with $e = x\, \overline{\sigma}_j\, \overline{d}_i$. Goal C.381 follows from rule iTm-var, rule iTm-forallE and rule iTm-constrE, in combination with Equations C.345, C.388, C.390 and C.392, with $\sigma' = [\overline{\sigma}_j / \overline{a}_j]\sigma$. Goal C.382 follows by applying Lemma 74 to Equation C.380.

$\boxed{\textbf{rule}\ \textsc{sTm-checkT-meth}}$ $P; \Gamma_C; \Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j / \overline{a}_j][\tau / a]\tau' \rightsquigarrow e.m\, \overline{\sigma}_j\, \overline{e}_i$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} m \Leftarrow [\overline{\tau}_j / \overline{a}_j][\tau / a]\tau' \rightsquigarrow e \tag{C.394}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma_0 \rightsquigarrow e.m\, \overline{\sigma}_j\, \overline{e}_i \tag{C.395}$$

$$\Gamma_C; \Gamma \vdash^{M}_{ty} [\overline{\tau}_j / \overline{a}_j][\tau / a]\tau' \rightsquigarrow \sigma_0 \tag{C.396}$$

From the rule premise we know that:

$$(m : \overline{C}'_k \Rightarrow TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau') \in \Gamma_C \tag{C.397}$$

$$\mathbf{unambig}(\forall \overline{a}_j, a . \overline{C}_i \Rightarrow \tau') \tag{C.398}$$

$$P; \Gamma_C; \Gamma \vDash [TC\, \tau] \rightsquigarrow e \tag{C.399}$$

$$\Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \tag{C.400}$$

$$\overline{P; \Gamma_C; \Gamma \vDash [[\overline{\tau}_j / \overline{a}_j][\tau / a] C_i] \rightsquigarrow e_i}^{i} \tag{C.401}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \tag{C.402}$$

$$\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.403}$$

By repeated case analysis on Equation C.343 (rule sCtx-clsEnv), together with Equation C.397, we know that:

$$(m : TC\, a : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma') \in \Gamma_C \tag{C.404}$$

By applying type equivalence (Theorem 40) to Equations C.400 and C.402, we get:

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \tag{C.405}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.406}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_j \rightsquigarrow \sigma_j}^{j} \tag{C.407}$$

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j}^{j} \tag{C.408}$$

By applying dictionary equivalence (Theorem 41) to Equations C.399 and C.401, we get:

$$P; \Gamma_C; \Gamma \vDash^{M} [TC\, \tau] \rightsquigarrow d \tag{C.409}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\, \sigma \rightsquigarrow e \tag{C.410}$$

$$\overline{P; \Gamma_C; \Gamma \vDash^{M} [[\overline{\tau}_j / \overline{a}_j][\tau / a] C_i] \rightsquigarrow d_i}^{i} \tag{C.411}$$

$$\overline{\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_j / \overline{a}_j][\sigma / a] C_i \rightsquigarrow e_i}^{i} \tag{C.412}$$

Goal C.394 follows from rule sTm-check-meth, in combination with Equations C.397, C.398, C.409, C.405, C.411, C.407 and C.343, with

$e = d.m\,\overline{\sigma}_j\,\overline{d}_i$. Consequently, Goal C.395 follows from rule iTm-method, rule iTm-forallE and rule iTm-constrE, in combination with Equations C.410, C.404, C.408 and C.412, with $\sigma_0 = [\overline{\sigma}_j/\overline{a}_j][\sigma/a]\sigma'$. Goal C.396 follows by applying Lemma 74 to Equation C.394.

$\boxed{\textbf{rule } \text{sTm-checkT-ArrI}}$ $\quad P; \Gamma_C; \Gamma \vdash_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma.e$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow e \tag{C.413}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma_0 \rightsquigarrow \lambda x : \sigma.e \tag{C.414}$$

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_1 \rightarrow \tau_2 \rightsquigarrow \sigma_0 \tag{C.415}$$

From the rule premise we know that:

$$x \notin \textbf{dom}(\Gamma) \tag{C.416}$$

$$P; \Gamma_C; \Gamma, x : \tau_1 \vdash_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \tag{C.417}$$

$$\Gamma_C; \Gamma \vdash_{ty} \tau_1 \rightsquigarrow \sigma \tag{C.418}$$

By applying type equivalence (Theorem 40) to Equation C.418, we get:

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau_1 \rightsquigarrow \sigma \tag{C.419}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.420}$$

From rule sCtxT-tyEnvTm, together with the 2nd hypothesis and Equation C.418, we know that:

$$\vdash_{ctx} P; \Gamma_C; \Gamma, x : \tau_1 \rightsquigarrow \Gamma, x : \sigma \tag{C.421}$$

By applying the induction hypothesis on Equation C.417, together with Equation C.421, we get:

$$P; \Gamma_C; \Gamma, x : \tau_1 \vdash_{tm}^{M} e \Leftarrow \tau_2 \rightsquigarrow e' \tag{C.422}$$

$$\Sigma; \Gamma_C; \Gamma, x : \sigma \vdash_{tm} e' : \sigma_2 \rightsquigarrow e \tag{C.423}$$

$$\Gamma_C; \Gamma, x : \tau_1 \vdash_{ty}^{M} \tau_2 \rightsquigarrow \sigma_2 \tag{C.424}$$

Goal C.413 follows from rule sTm-check-ArrI, together with Equations C.416, C.422 and C.419, with $e = \lambda x : \sigma.e'$. Consequently, Goal C.414 follows from rule iTm-arrI, together with Equations C.423 and C.420, with $\sigma_0 = \sigma \rightarrow \sigma_2$. Goal C.415 follows by applying Lemma 74 to Equation C.413.

$\boxed{\textbf{rule } \textsc{sTm-checkT-Inf}}$ $\quad P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e$

The goal to be proven is the following:

$$P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e \tag{C.425}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e \tag{C.426}$$

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \tag{C.427}$$

From the rule premise we know that:

$$P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e \tag{C.428}$$

Goals C.425, C.426 and C.427 follow directly by applying Part 1 of this theorem to Equation C.428.

□

Theorem 44 (Equivalence - Contexts)**.**

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$
  and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Gamma'$
  then $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$
  and $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M$
  where $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma; \Gamma_C; \Gamma'$
  and $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma' \vdash_{ty}^{M} \tau' \rightsquigarrow \sigma'$.

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$
  and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Gamma'$
  then $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$
  and $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M$
  where $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma; \Gamma_C; \Gamma'$
  and $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma' \vdash_{ty}^{M} \tau' \rightsquigarrow \sigma'$.

- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$
  and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Gamma'$
  then $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$
  and $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M$
  where $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma; \Gamma_C; \Gamma'$
  and $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma' \vdash_{ty}^{M} \tau' \rightsquigarrow \sigma'$.

- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$
  and $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Gamma'$

then $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$
and $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M$
where $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma; \Gamma_C; \Gamma'$
and $\Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma' \vdash^{M}_{ty} \tau' \rightsquigarrow \sigma'$.

Proof. By straightforward induction on the typing derivation. □

C.7 Coherence Theorems

C.7.1 Compatibility Lemmas

Lemma 120 (Compatibility - Term Abstraction)**.**

$$\frac{\Gamma_C; \Gamma, x : \sigma_1 \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma_2}{\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda x : \sigma_1.e_1 \simeq_{log} \Sigma_2 : \lambda x : \sigma_1.e_2 : \sigma_1 \to \sigma_2}$$

Proof. By the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \tag{C.429}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.430}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(\lambda x : \sigma_1.e_1))), \Sigma_2 : \gamma_2(\phi_2(R(\lambda x : \sigma_1.e_2)))) \in \mathcal{E}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C}$$

By the definition of the $\mathcal{E}$ relation and the fact that term abstractions are values, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(\lambda x : \sigma_1.e_1))) : R(\sigma_1 \to \sigma_2)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(\lambda x : \sigma_1.e_2))) : R(\sigma_1 \to \sigma_2)$$

$$(\Sigma_1 : \gamma_1(\phi_1(R(\lambda x : \sigma_1.e_1))), \Sigma_2 : \gamma_2(\phi_2(R(\lambda x : \sigma_1.e_2)))) \in \mathcal{V}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C}$$

By applying the substitutions, the goal simplifies to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda x : R(\sigma_1).(\gamma_1(\phi_1(R(e_1)))) : R(\sigma_1 \to \sigma_2) \tag{C.431}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda x : R(\sigma_1).(\gamma_2(\phi_2(R(e_2)))) : R(\sigma_1 \to \sigma_2) \tag{C.432}$$

$$(\Sigma_1 : \lambda x : R(\sigma_1).(\gamma_1(\phi_1(R(e_1)))), \Sigma_2 : \lambda x : R(\sigma_1).(\gamma_2(\phi_2(R(e_2))))) \in \mathcal{V}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C} \tag{C.433}$$

By unfolding the definition of logical equivalence in the hypothesis of the theorem, we get:

$$(\Sigma_1 : \gamma_1'(\phi_1'(R'(e_1))), \Sigma_2 : \gamma_2'(\phi_2'(R'(e_2)))) \in \mathcal{E}[\![\sigma_2]\!]_{R'}^{\Gamma_C} \tag{C.434}$$

for any $R' \in \mathcal{F}[\![\Gamma, x : \sigma_1]\!]^{\Gamma_C}$, $\phi' \in \mathcal{G}[\![\Gamma, x : \sigma_1]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, x : \sigma_1]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$.

By the definition of the $\mathcal{F}$-relation and from Equation C.429, we have that $R \in \mathcal{F}[\![\Gamma, x : \sigma_1]\!]^{\Gamma_C}$ and we choose $R' = R$. By case analysis on ϕ', we know that $\phi' = \phi'', x \mapsto (e_3, e_4)$ for some $\phi'' \in \mathcal{G}[\![\Gamma]\!]_{R}^{\Sigma_1,\Sigma_2,\Gamma_C}$ and expressions e_3 and e_4 such that

$$(\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]_{R}^{\Gamma_C} \tag{C.435}$$

We choose $\phi'' = \phi$. Lastly, by the definition of the $\mathcal{H}$-relation and from Equation C.430, we have that $\gamma \in \mathcal{H}[\![\Gamma, x : \sigma_1]\!]_{R}^{\Sigma_1,\Sigma_2,\Gamma_C}$ and we choose $\gamma' = \gamma$.

With the above mentioned choices for γ', ϕ' and R', unfolding the definition of the $\mathcal{E}$-relation in Equation C.434, gives us:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1((\phi_1, x \mapsto e_3)(R(e_1))) : R(\sigma_2) \tag{C.436}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2((\phi_2, x \mapsto e_4)(R(e_2))) : R(\sigma_2) \tag{C.437}$$

$$\begin{aligned} &\exists v_3, v_4 : \Sigma_1 \vdash \gamma_1((\phi_1, x \mapsto e_3)(R(e_1))) \longrightarrow^* v_3 \\ &\quad \wedge \Sigma_2 \vdash \gamma_2((\phi_2, x \mapsto e_4)(R(e_2))) \longrightarrow^* v_4 \\ &\quad \wedge (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![\sigma_2]\!]_{R}^{\Gamma_C} \end{aligned} \tag{C.438}$$

By unfolding the definition of the $\mathcal{E}$-relation in Equation C.435, we know that:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_3 : \sigma_1 \tag{C.439}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_4 : \sigma_1 \tag{C.440}$$

Note that neither e_3 nor e_4 contain any free variables, thus $\gamma_1(\phi_1(R(e_3))) = e_3$ and $\gamma_2(\phi_2(R(e_4))) = e_4$. Taking these equations into account, from the definition of substitution, Equations C.436 and C.437 are rewritten to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} [e_3/x](\gamma_1(\phi_1(R(e_1)))) : R(\sigma_2) \tag{C.441}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} [e_4/x](\gamma_2(\phi_2(R(e_2)))) : R(\sigma_2) \tag{C.442}$$

By applying the substitution Lemma 86 on Equations C.441 and C.442 respectively, in combination with Equations C.439 and C.440, we get:

$$\Sigma_1; \Gamma_C; \bullet, x : \sigma_1 \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma_2) \tag{C.443}$$

$$\Sigma_2; \Gamma_C; \bullet, x : \sigma_1 \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\sigma_2) \tag{C.444}$$

By Lemma 113, it follows from Equation C.443 that $\vdash_{ctx} \Sigma_1; \Gamma_C; \bullet, x : \sigma_1$. By case analysis on this result, rule rule iCtx-tyEnvTm (the rule with which variable x wad inserted in the environment) tells us that:

$$\Gamma_C; \bullet \vdash_{ty} \sigma_1 \tag{C.445}$$

Goals C.431 and C.432 follow from applying the typing rule rule iTm-arrI on Equations C.443 and C.444 respectively, together with Equation C.445.

It remains to show Goal C.433. By unfolding the definition of the $\mathcal{V}$ relation, the goal simplifies to

$$\forall e_5\, e_6,\ \text{if } (\Sigma_1 : e_5, \Sigma_2 : e_6) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} \text{ then} \tag{C.446}$$

$$(\Sigma_1 : \lambda x : R(\sigma_1).(\gamma_1(\phi_1(R(e_1))))\, e_5, \Sigma_2 : \lambda x : R(\sigma_1).(\gamma_2(\phi_2(R(e_2))))\, e_6) \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C} \tag{C.447}$$

Then, suppose expressions e_5 and e_6, such that Equation C.446 holds. By unfolding the definition of the $\mathcal{E}$ relation in Equation C.446, we have

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_5 : \sigma_1 \tag{C.448}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_6 : \sigma_1 \tag{C.449}$$

We also unfold the definition of the $\mathcal{E}$ relation in Goal C.447, to get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\lambda x : R(\sigma_1).(\gamma_1(\phi_1(R(e_1)))))\, e_5 : \sigma_2 \tag{C.450}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\lambda x : R(\sigma_1).(\gamma_2(\phi_2(R(e_2)))))\, e_6 : \sigma_2 \tag{C.451}$$

$$\begin{aligned} \exists v_5, v_6 : \Sigma_1 \vdash (\lambda x : R(\sigma_1).(\gamma_1(\phi_1(R(e_1)))))\, e_5 \longrightarrow^* v_5 \\ \wedge\, \Sigma_2 \vdash (\lambda x : R(\sigma_1).(\gamma_2(\phi_2(R(e_2)))))\, e_6 \longrightarrow^* v_6 \\ \wedge\, (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C} \end{aligned} \tag{C.452}$$

Goals C.450 and C.451 follow by applying the rule iTm-arrE typing rule once on Equations C.443 and C.448 and once more on Equations C.444 and C.449. Note that Equations C.443 and C.444 have been already proven above.

By case analysis, it is easy to see that the first step of the evaluations in Goal C.452 is rule iEval-appAbs, reducing the goal to:

$$\begin{aligned} \exists v_5, v_6 : \Sigma_1 \vdash \gamma_1(\phi_1(R([e_5/x]e_1))) \longrightarrow^* v_5 \\ \wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R([e_6/x]e_2))) \longrightarrow^* v_6 \\ \wedge\, (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C} \end{aligned}$$

We choose $e_3 = e_5$ and $e_4 = e_6$ in Equation C.435. The goal follows by choosing $v_5 = v_3$ and $v_6 = v_4$ from Equation C.438.

□

Lemma 121 (Compatibility - Term Application).

$$\frac{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_1' : \sigma_1 \to \sigma_2 \qquad \Gamma_C; \Gamma \vdash \Sigma_1 : e_2 \simeq_{log} \Sigma_2 : e_2' : \sigma_1}{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1\, e_2 \simeq_{log} \Sigma_2 : e_1'\, e_2' : \sigma_2}$$

Proof. By inlining the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1\, e_2))), \Sigma_2 : \gamma_2(\phi_2(R(e_1'\, e_2')))) \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C}$$

By applying the definition of the $\mathcal{E}$ relation, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1\, e_2))) : R(\sigma_2) \tag{C.453}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_1'\, e_2'))) : R(\sigma_2) \tag{C.454}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 \vdash \gamma_1(\phi_1(R(e_1\, e_2))) &\longrightarrow^* v_1 \\ \wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R(e_1'\, e_2'))) &\longrightarrow^* v_2 \\ \wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) &\in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C} \end{aligned} \tag{C.455}$$

By applying the substitutions in Goals C.453 and C.454, they reduce to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\gamma_1(\phi_1(R(e_1))))\, (\gamma_1(\phi_1(R(e_2)))) : R(\sigma_2) \tag{C.456}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\gamma_2(\phi_2(R(e_1'))))\, (\gamma_2(\phi_2(R(e_2')))) : R(\sigma_2) \tag{C.457}$$

By inlining the definition of logical equivalence in the premise of the rule, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma_1 \to \sigma_2) \tag{C.458}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_1'))) : R(\sigma_1 \to \sigma_2) \tag{C.459}$$

$$\exists v_3, v_4 : \Sigma_1 \vdash \gamma_1(\phi_1(R(e_1))) \longrightarrow^* v_3 \tag{C.460}$$

$$\wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R(e_1'))) \longrightarrow^* v_4 \tag{C.461}$$

$$\wedge\, (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C} \tag{C.462}$$

and

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_2))) : R(\sigma_1) \tag{C.463}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2'))) : R(\sigma_1) \tag{C.464}$$

$$\exists v_5, v_6 : \Sigma_1 \vdash \gamma_1(\phi_1(R(e_2))) \longrightarrow^* v_5$$

$$\wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R(e_2'))) \longrightarrow^* v_6$$

$$\wedge\, (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma_1]\!]_R^{\Gamma_C}$$

By applying both Equation C.458 and C.463 and both Equation C.459 and C.464 respectively to rule iTm-arrE, Goal C.456 and C.457 are proven.

By applying the substitution, Goal C.455 reduces to:

$$\exists v_1, v_2 : \Sigma_1 \vdash (\gamma_1(\phi_1(R(e_1))))\,(\gamma_1(\phi_1(R(e_2)))) \longrightarrow^* v_1$$

$$\wedge\, \Sigma_2 \vdash (\gamma_2(\phi_2(R(e_1'))))\,(\gamma_2(\phi_2(R(e_2')))) \longrightarrow^* v_2$$

$$\wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C}$$

Through case analysis, we see that we should first (repeatedly) apply rule iEval-app and Equation C.460 and C.461 respectively. The goal reduces to:

$$\exists v_1, v_2 : \Sigma_1 \vdash v_3\,(\gamma_1(\phi_1(R(e_2)))) \longrightarrow^* v_1$$

$$\wedge\, \Sigma_2 \vdash v_4\,(\gamma_2(\phi_2(R(e_2')))) \longrightarrow^* v_2 \tag{C.465}$$

$$\wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C}$$

By the definition of the $\mathcal{V}$-relation in Equation C.462, we know that:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} v_3 : R(\sigma_1 \to \sigma_2)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} v_4 : R(\sigma_1 \to \sigma_2)$$

$$\forall (\Sigma_1 : e_5, \Sigma_2 : e_6) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} : (\Sigma_1 : v_3\, e_5, \Sigma_2 : v_4\, e_6) \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C} \tag{C.466}$$

We choose $e_5 = \gamma_1(\phi_1(R(e_2)))$ and $e_6 = \gamma_2(\phi_2(R(e_2')))$. Goal C.465 now follows from the definition of the $\mathcal{E}$-relation in Equation C.466.

□

Lemma 122 (Compatibility - Dictionary Abstraction)**.**

$$\frac{\Gamma_C; \Gamma, \delta : C \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma}{\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda\delta : C.e_1 \simeq_{log} \Sigma_2 : \lambda\delta : C.e_2 : C \Rightarrow \sigma}$$

Proof. By unfolding the definition of logical equivalence in the conclusion of the lemma, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \tag{C.467}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.468}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(\lambda\delta : C.e_1))), \Sigma_2 : \gamma_2(\phi_2(R(\lambda\delta : C.e_2)))) \in \mathcal{E}[\![C \Rightarrow \sigma]\!]_R^{\Gamma_C}$$

By applying the definition of the $\mathcal{E}$ relation (taking into account that $\lambda\delta : C.e$ is a value) and partially applying the substitutions, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))) : R(C \Rightarrow \sigma) \tag{C.469}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2)))) : R(C \Rightarrow \sigma) \tag{C.470}$$

$$(\Sigma_1 : \lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))), \Sigma_2 : \lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2))))) \in \mathcal{V}[\![C \Rightarrow \sigma]\!]_R^{\Gamma_C} \tag{C.471}$$

By unfolding the definition of logical equivalence in the premise of this lemma, we get:

$$(\Sigma_1 : \gamma_1'(\phi_1'(R'(e_1))), \Sigma_2 : \gamma_2'(\phi_2'(R'(e_2)))) \in \mathcal{E}[\![\sigma]\!]_{R'}^{\Gamma_C} \tag{C.472}$$

for any $R' \in \mathcal{F}[\![\Gamma, \delta : C]\!]^{\Gamma_C}$, $\phi' \in \mathcal{G}[\![\Gamma, \delta : C]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, \delta : C]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$.

By the definition of the $\mathcal{F}$ and the $\mathcal{G}$ relations and from Equations C.467 and C.468, we have that $R \in \mathcal{F}[\![\Gamma, \delta : C]\!]^{\Gamma_C}$ and $\phi \in \mathcal{G}[\![\Gamma, \delta : C]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$. We choose $R' = R$ and $\phi' = \phi$. By case analysis on γ', we know that $\gamma' = \gamma'', \delta \mapsto (d_1, d_2)$ for some $\gamma'' \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ and some dictionaries d_1 and d_2 such that

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} \tag{C.473}$$

We choose $\gamma'' = \gamma$. Then, unfolding the definition of the $\mathcal{E}$ relation in Equation C.472 results in:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\gamma_1, \delta \mapsto d_1)(\phi_1(R(e_1))) : R(\sigma) \tag{C.474}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\gamma_2, \delta \mapsto d_2)(\phi_2(R(e_2))) : R(\sigma) \tag{C.475}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 \vdash (\gamma_1, \delta \mapsto d_1)(\phi_1(R(e_1))) \longrightarrow^* v_1 \\ \wedge\, \Sigma_2 \vdash (\gamma_2, \delta \mapsto d_2)(\phi_2(R(e_2))) \longrightarrow^* v_2 \\ \wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \end{aligned} \tag{C.476}$$

From the definition of the $\mathcal{E}$ relation in Equation C.473, it follows that:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : R(C) \tag{C.477}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : R(C) \tag{C.478}$$

Since neither d_1 nor d_2 contain any free variables, we know that $\gamma_1(d_1) = d_1$ and $\gamma_2(d_2) = d_2$. Consequently, Equations C.474 and C.475 are equivalent to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} [d_1/\delta](\gamma_1(\phi_1(R(e_1)))) : R(\sigma) \tag{C.479}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} [d_2/\delta](\gamma_2(\phi_2(R(e_2)))) : R(\sigma) \tag{C.480}$$

By applying the substitution Lemma 88 on Equations C.479 and C.480 respectively, in combination with Equations C.477 and C.478, we find:

$$\Sigma_1; \Gamma_C; \bullet, \delta : C \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma) \tag{C.481}$$

$$\Sigma_2; \Gamma_C; \bullet, \delta : C \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\sigma) \tag{C.482}$$

By Lemma 113, it follows from Equation C.481 that $\vdash_{ctx} \Sigma_1; \Gamma_C; \bullet, \delta : C$. Consequently, we know from rule iCtx-tyEnvD that:

$$\Gamma_C; \bullet \vdash_C C \tag{C.483}$$

Since C does not contain any free variables, it is straightforward to see that $R(C) = C$.

Consequently, goals C.469 and C.470 follow by applying Equation C.481 and C.482 respectively, in combination with Equation C.483, to rule iTm-constrI.

By unfolding the definition of $\mathcal{V}$, Goal C.471 reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))) : R(C \Rightarrow \sigma) \tag{C.484}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2)))) : R(C \Rightarrow \sigma) \tag{C.485}$$

$$\forall d_3\, d_4,\ \text{if } (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} \text{ then} \tag{C.486}$$

$$(\Sigma_1 : (\lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))))\, d_3, \Sigma_2 : (\lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2)))))\, d_4) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.487}$$

Goals C.484 and C.485 are identical to Goals C.469 and C.470, which have been proven above. For the final goal, suppose dictionaries d_3 and d_4 such that Equation C.486 holds. By the definition of the $\mathcal{E}$ relation in Equation C.486, we obtain:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_3 : R(C) \tag{C.488}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_4 : R(C) \tag{C.489}$$

We unfold the definition of the $\mathcal{E}$ relation in Goal C.487, reducing it to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))))\, d_3 : R(\sigma) \tag{C.490}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2)))))\, d_4 : R(\sigma) \tag{C.491}$$

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 \vdash (\lambda\delta : R(C).(\gamma_1(\phi_1(R(e_1)))))\, d_3 \longrightarrow^* v_3 \\ \wedge\, \Sigma_2 \vdash (\lambda\delta : R(C).(\gamma_2(\phi_2(R(e_2)))))\, d_4 \longrightarrow^* v_4 \\ \wedge\, (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \end{aligned} \tag{C.492}$$

Goals C.490 and C.491 follow by applying the rule iTm-constrE typing rule once on Equations C.484 and C.488 and once on Equations C.485 and C.489.

Through case analysis, it is straightforward to note that the first step of the evaluation paths in Equation C.492 should be by rule rule iEval-DAppAbs. The goal reduces to:

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 \vdash [d_3/\delta](\gamma_1(\phi_1(R(e_1)))) \longrightarrow^* v_3 \\ \wedge\, \Sigma_2 \vdash [d_4/\delta](\gamma_2(\phi_2(R(e_2)))) \longrightarrow^* v_4 \\ \wedge\, (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \end{aligned}$$

The above goal follows directly from Equation C.476, by choosing $d_1 = d_3$, $d_2 = d_4$, $v_3 = v_1$ and $v_4 = v_2$.

□

Lemma 123 (Compatibility - Dictionary Application)**.**

$$\frac{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : C \Rightarrow \sigma \qquad \Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C}{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1\, d_1 \simeq_{log} \Sigma_2 : e_2\, d_2 : \sigma}$$

Proof. By inlining the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1\, d_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2\, d_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

By unfolding the definition of the $\mathcal{E}$ relation in the goal above, and by simplifying the substitutions, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\gamma_1(\phi_1(R(e_1))))\, (\gamma_1(R(d_1))) : R(\sigma) \tag{C.493}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\gamma_2(\phi_2(R(e_2))))\, (\gamma_2(R(d_2))) : R(\sigma) \tag{C.494}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 &\vdash (\gamma_1(\phi_1(R(e_1))))\, (\gamma_1(R(d_1))) \longrightarrow^* v_1 \\ &\wedge \Sigma_2 \vdash (\gamma_2(\phi_2(R(e_2))))\, (\gamma_2(R(d_2))) \longrightarrow^* v_2 \\ &\wedge (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \end{aligned} \tag{C.495}$$

By inlining the definitions of logical equivalence and the $\mathcal{E}$ relation in the first premise of this lemma, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(C \Rightarrow \sigma) \tag{C.496}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(C \Rightarrow \sigma) \tag{C.497}$$

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 &\vdash \gamma_1(\phi_1(R(e_1))) \longrightarrow^* v_3 \\ &\wedge \Sigma_2 \vdash \gamma_2(\phi_2(R(e_2))) \longrightarrow^* v_4 \\ &\wedge (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![C \Rightarrow \sigma]\!]_R^{\Gamma_C} \end{aligned} \tag{C.498}$$

Similarly, by unfolding the definition of logical equivalence in the second premise of the rule, we get:

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} \tag{C.499}$$

From the definition of the $\mathcal{E}$ relation in Equation C.499 we have:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R(d_1)) : R(C) \tag{C.500}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R(d_2)) : R(C) \tag{C.501}$$

Note that, by the definition of substitution, we have $R(C \Rightarrow \sigma) = R(C) \Rightarrow R(\sigma)$. This allows the application of the typing rule rule iTm-constrE once on Equations C.496 and C.500 and once more on Equations C.497 and C.501, therefore proving Goals C.493 and C.494.

Through application of the rule iEval-DApp evaluation rule on each step of the two evaluation paths in Equation C.498, Goal C.495 reduces to:

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 \vdash v_3\ (\gamma_1(R(d_1))) \longrightarrow^* v_1 \\ \wedge\ \Sigma_2 \vdash v_4\ (\gamma_2(R(d_2))) \longrightarrow^* v_2 \\ \wedge\ (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \end{aligned} \tag{C.502}$$

Unfolding the definition of the $\mathcal{V}$ relation in Equation C.498 results in:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} v_3 : R(C \Rightarrow \sigma)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} v_4 : R(C \Rightarrow \sigma)$$

$$\forall (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} : (\Sigma_1 : v_3\, d_3, \Sigma_2 : v_4\, d_4) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.503}$$

We take $d_3 = \gamma_1(R(d_1))$ and $d_4 = \gamma_2(R(d_2))$. Goal C.502 follows from the definition of the $\mathcal{E}$ relation in Equation C.503.

□

Lemma 124 (Compatibility - Type Abstraction)**.**

$$\frac{\Gamma_C; \Gamma, a \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma}{\Gamma_C; \Gamma \vdash \Sigma_1 : \Lambda a.e_1 \simeq_{log} \Sigma_2 : \Lambda a.e_2 : \forall a.\sigma}$$

Proof. By unfolding the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \tag{C.504}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.505}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.506}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(\Lambda a.e_1))), \Sigma_2 : \gamma_2(\phi_2(R(\Lambda a.e_2)))) \in \mathcal{E}[\![\forall a.\sigma]\!]_R^{\Gamma_C}$$

Because $a \notin \Gamma$, from Equation C.504 it follows that a is not in the domain of R. Furthermore, from Equations C.505 and C.506 it follows that for every mapping $x \mapsto (e'_1, e'_2) \in \phi$ and for every mapping $\delta \mapsto (d'_1, d'_2) \in \gamma$, we have $a \notin \mathbf{fv}(e'_i)$ and $a \notin \mathbf{fv}(d'_i)$, where $i \in \{1,2\}$. Therefore, we obtain $\gamma_i(\phi_i(R(\Lambda a.e_i))) = \Lambda a.\gamma_i(\phi_i(R(e_i)))$, for $i \in \{1,2\}$. With these equations, the goal above reduces to

$$(\Sigma_1 : \Lambda a.\gamma_1(\phi_1(R(e_1))), \Sigma_2 : \Lambda a.\gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\forall a.\sigma]\!]_R^{\Gamma_C}$$

By applying the definition of the $\mathcal{E}$ relation, taking into account that expressions of the form $\Lambda a.e$ are values, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \Lambda a.\gamma_1(\phi_1(R(e_1))) : R(\forall a.\sigma) \tag{C.507}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \Lambda a.\gamma_2(\phi_2(R(e_2))) : R(\forall a.\sigma) \tag{C.508}$$

$$(\Sigma_1 : \Lambda a.\gamma_1(\phi_1(R(e_1))), \Sigma_2 : \Lambda a.\gamma_2(\phi_2(R(e_2)))) \in \mathcal{V}[\![\forall a.\sigma]\!]_R^{\Gamma_C} \tag{C.509}$$

Suppose any σ' such that

$$\Gamma_C; \bullet \vdash_{ty} \sigma' \tag{C.510}$$

and any $r \in Rel[\sigma']$. Then, inlining the definition of the $\mathcal{V}$ relation in Goal C.509, reduces it to:

$$(\Sigma_1 : (\Lambda a.\gamma_1(\phi_1(R(e_1))))\,\sigma', \Sigma_2 : (\Lambda a.\gamma_2(\phi_2(R(e_2))))\,\sigma') \in \mathcal{E}[\![\sigma]\!]_{R,a\mapsto(\sigma',r)}^{\Gamma_C} \tag{C.511}$$

Unfolding the definition of logical equivalence in the premise of this lemma, gives us:

$$(\Sigma_1 : \gamma'_1(\phi'_1(R'(e_1))), \Sigma_2 : \gamma'_2(\phi'_2(R'(e_2)))) \in \mathcal{E}[\![\sigma]\!]_{R'}^{\Gamma_C} \tag{C.512}$$

for any $R' \in \mathcal{F}[\![\Gamma, a]\!]^{\Gamma_C}$, $\phi' \in \mathcal{G}[\![\Gamma, a]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, a]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$.

By the definition of the $\mathcal{F}$ relation, we know that $R' = R'', a \mapsto (\sigma'', r')$ for some $R'' \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and σ'' such that $\Gamma_C; \bullet \vdash_{ty} \sigma''$ and $r' \in \mathit{Rel}[\sigma'']$. We choose $R'' = R$, $\sigma'' = \sigma'$ and $r' = r$. By the definition of the $\mathcal{G}$ and $\mathcal{H}$ relations and from Equations C.505 and C.506, we have that $\phi \in \mathcal{G}[\![\Gamma, a]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma, a]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$. Then, we choose $\phi' = \phi$ and $\gamma' = \gamma$.

Unfolding the definition of the $\mathcal{E}$ relation in Equation C.512, results in:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R, a \mapsto (\sigma', r)(e_1))) : (R, a \mapsto (\sigma', r))(\sigma) \tag{C.513}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R, a \mapsto (\sigma', r)(e_2))) : (R, a \mapsto (\sigma', r))(\sigma) \tag{C.514}$$

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 &\vdash \gamma_1(\phi_1(R, a \mapsto (\sigma', r)(e_1))) \longrightarrow^* v_3 \\ &\wedge \Sigma_2 \vdash \gamma_2(\phi_2(R, a \mapsto (\sigma', r)(e_2))) \longrightarrow^* v_4 \\ &\wedge (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]^{\Gamma_C}_{(R, a \mapsto (\sigma', r))} \end{aligned} \tag{C.515}$$

By the definition of substitution, and because σ' has no free variables, it follows that $(R, a \mapsto (\sigma', r))(\sigma) = [\sigma'/a](R(\sigma))$. Furthermode, because $a \notin \Gamma$, from Equations C.505 and C.506 it follows that $\gamma_i(\phi_i((R, a \mapsto (\sigma', r))(e))) = [\sigma'/a](\gamma_i(\phi_i(R(e))))$, for any expression e and $i \in \{1, 2\}$. Taking these equalities into account, by applying Equations C.513 and C.514 to reverse substitution Lemma 90 gives us:

$$\Sigma_1; \Gamma_C; \bullet, a \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma) \tag{C.516}$$

$$\Sigma_2; \Gamma_C; \bullet, a \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\sigma) \tag{C.517}$$

Because a is not in the domain of R, we have $R(\forall a.\sigma) = \forall a.R(\sigma)$. Hence, Goals C.507 and C.508 follow by passing Equations C.516 and C.517 to rule ITmForallI, respectively.

Unfolding the definition of the $\mathcal{E}$ relation in Goal C.511 and since $(R, a \mapsto (\sigma', r))(\sigma) = [\sigma'/a](R(\sigma))$, the goal reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\Lambda a.\gamma_1(\phi_1(R(e_1))))\, \sigma' : [\sigma'/a](R(\sigma)) \tag{C.518}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\Lambda a.\gamma_2(\phi_2(R(e_2))))\, \sigma' : [\sigma'/a](R(\sigma)) \tag{C.519}$$

$$\begin{aligned} \exists v_5, v_6 : \Sigma_1 &\vdash (\Lambda a.\gamma_1(\phi_1(R(e_1))))\, \sigma' \longrightarrow^* v_5 \\ &\wedge \Sigma_2 \vdash (\Lambda a.\gamma_2(\phi_2(R(e_2))))\, \sigma' \longrightarrow^* v_6 \\ &\wedge (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma]\!]^{\Gamma_C}_{R, a \mapsto (\sigma', r)} \end{aligned} \tag{C.520}$$

Goals C.518 and C.519 follow by applying Goals C.507 and C.508 (which have previously been proven) to rule iTm-forallE, respectively, together with Equation C.510. The first step of both evaluation paths in Equation C.520 can only be taken by appropriate instantiations of rule rule iEval-tyAppAbs. With this, Goal C.520 can be further reduced to

$$\exists v_5, v_6 : \Sigma_1 \vdash [\sigma'/a](\gamma_1(\phi_1(R(e_1)))) \longrightarrow^* v_5$$

$$\wedge\, \Sigma_2 \vdash [\sigma'/a](\gamma_2(\phi_2(R(e_2)))) \longrightarrow^* v_6$$

$$\wedge\, (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma]\!]^{\Gamma_C}_{R, a \mapsto (\sigma', r)}$$

which follows from Equation C.515 by choosing $v_5 = v_3$ and $v_6 = v_4$.

□

Lemma 125 (Compatibility - Type Application)**.**

$$\frac{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \forall a.\sigma' \qquad \Gamma_C; \Gamma \vdash_{ty} \sigma}{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1\, \sigma \simeq_{log} \Sigma_2 : e_2\, \sigma : [\sigma/a]\sigma'}$$

Proof. By inlining the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_R$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_R$$

Note that, by the definition of the $\mathcal{F}$ relation, a is not in the domain of R, since $a \notin \Gamma$. The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1\, \sigma))), \Sigma_2 : \gamma_2(\phi_2(R(e_2\, \sigma)))) \in \mathcal{E}[\![[\sigma/a]\sigma']\!]^{\Gamma_C}_R \tag{C.521}$$

From the definition of substitution we have that

$$\gamma_i(\phi_i(R(e_i\, \sigma))) = \gamma_i(\phi_i(R(e_i)))\, R(\sigma), \text{ for } i \in \{1, 2\}$$

$$\text{and } \; R([\sigma/a]\sigma') = [R(\sigma)/a]R(\sigma')$$

Taking into account these equalities and by unfolding the definition of the $\mathcal{E}$ relation, Goal C.521 reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1)))\, R(\sigma) : [R(\sigma)/a]R(\sigma') \tag{C.522}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2)))\, R(\sigma) : [R(\sigma)/a]R(\sigma') \tag{C.523}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 \vdash (\gamma_1(\phi_1(R(e_1))))\, R(\sigma) \longrightarrow^* v_1 \\ \wedge\, \Sigma_2 \vdash (\gamma_2(\phi_2(R(e_2))))\, R(\sigma) \longrightarrow^* v_2 \\ \wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![[\sigma/a]\sigma']\!]^{\Gamma_C}_{R} \end{aligned} \tag{C.524}$$

By inlining the definition of logical equivalence in the first premise of this lemma, we get

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\forall a.\sigma']\!]^{\Gamma_C}_{R}$$

Unfolding the definition of the $\mathcal{E}$ relation results in:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\forall a.\sigma') \tag{C.525}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\forall a.\sigma') \tag{C.526}$$

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 \vdash \gamma_1(\phi_1(R(e_1))) \longrightarrow^* v_3 \\ \wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R(e_2))) \longrightarrow^* v_4 \\ \wedge\, (\Sigma_1 : v_3, \Sigma_2 : v_4) \in \mathcal{V}[\![\forall a.\sigma']\!]^{\Gamma_C}_{R} \end{aligned} \tag{C.527}$$

Starting from the second premise of this lemma, by sequentially applying Lemma 82 with the substitutions of R on σ, it follows that $\Gamma_C; \Gamma' \vdash_{ty} R(\sigma)$, where Γ' only contains term variables. Then, starting from this result, by sequentually applying Lemma 105, we obtain

$$\Gamma_C; \bullet \vdash_{ty} R(\sigma) \tag{C.528}$$

Since a is not in the domain of R, we have $R(\forall a.\sigma) = \forall a.R(\sigma)$. Consequently, Goals C.522 and C.523 follow by instantiating rule rule iTm-forallE with Equations C.525 and C.526, respectively, together with Equation C.528.

The definition of the $\mathcal{V}$ relation in Equation C.527 tells us that:

$$\begin{aligned} \forall \sigma'', r \in Rel[\sigma''] : \Gamma_C; \bullet \vdash_{ty} \sigma'' \\ \Rightarrow (\Sigma_1 : v_3\, \sigma'', \Sigma_2 : v_4\, \sigma'') \in \mathcal{E}[\![\sigma']\!]^{\Gamma_C}_{R, a \mapsto (\sigma'', r)} \end{aligned} \tag{C.529}$$

By repeatedly applying rule iEval-tyApp on each step of both evaluation paths in Equation C.527, Goal C.524 reduces to:

$$\exists v_1, v_2 : \Sigma_1 \vdash v_3\, R(\sigma) \longrightarrow^* v_1$$

$$\wedge\, \Sigma_2 \vdash v_4\, R(\sigma) \longrightarrow^* v_2$$

$$\wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![[\sigma/a]\sigma']\!]_R^{\Gamma_C}$$

which follows directly from C.529 by choosing $\sigma'' = \sigma$ and unfolding the definition of the $\mathcal{E}$ relation.

□

Lemma 126 (Compatibility - Let Binding)**.**

$$\frac{\Gamma_C; \Gamma \vdash \Sigma_1 : e_1' \simeq_{log} \Sigma_2 : e_2' : \sigma_1 \qquad \Gamma_C; \Gamma, x : \sigma_1 \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma_2}{\Gamma_C; \Gamma \vdash \Sigma_1 : \textbf{\textit{let}}\ \ x : \sigma_1 = e_1'\ \textbf{\textit{in}}\ \ e_1 \simeq_{log} \Sigma_2 : \textbf{\textit{let}}\ \ x : \sigma_1 = e_2'\ \textbf{\textit{in}}\ \ e_2 : \sigma_2}$$

Proof. By inlining the definition of logical equivalence, suppose we have:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C} \tag{C.530}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$$

The goal to be proven is the following:

$$(\Sigma_1 : e_1'', \Sigma_2 : e_2'') \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C} \tag{C.531}$$

where $e_1'' = \gamma_1(\phi_1(R(\textbf{let}\ \ x : \sigma_1 = e_1'\ \textbf{in}\ \ e_1)))$ and $e_2'' = \gamma_2(\phi_2(R(\textbf{let}\ \ x : \sigma_1 = e_2'\ \textbf{in}\ \ e_2)))$. From the definition of substitution, it follows that

$$\gamma_i(\phi_i(R(\textbf{let}\ \ x : \sigma_1 = e_i'\ \textbf{in}\ \ e_i))) = \tag{C.532}$$

$$\textbf{let}\ \ x : R(\sigma_1) = (\gamma_i(\phi_i(R(e_i'))))\ \textbf{in}\ \ (\gamma_i(\phi_i(R(e_i)))) \tag{C.533}$$

for $i \in \{1, 2\}$ Taking into account this equality, by applying the definition of the $\mathcal{E}$ relation, Goal C.531 reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \textbf{let } x : R(\sigma_1) = (\gamma_1(\phi_1(R(e'_1)))) \textbf{ in } (\gamma_1(\phi_1(R(e_1)))) : R(\sigma_2) \tag{C.534}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \textbf{let } x : R(\sigma_1) = (\gamma_2(\phi_2(R(e'_2)))) \textbf{ in } (\gamma_2(\phi_2(R(e_2)))) : R(\sigma_2) \tag{C.535}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 &\vdash \textbf{let } x : R(\sigma_1) = (\gamma_1(\phi_1(R(e'_1)))) \textbf{ in } (\gamma_1(\phi_1(R(e_1)))) \longrightarrow^* v_1 \\ &\wedge \Sigma_2 \vdash \textbf{let } x : R(\sigma_1) = (\gamma_2(\phi_2(R(e'_2)))) \textbf{ in } (\gamma_2(\phi_2(R(e_2)))) \longrightarrow^* v_2 \\ &\wedge (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C} \end{aligned} \tag{C.536}$$

By inlining the definition of logical equivalence in the two hypotheses of this lemma, we get:

$$(\Sigma_1 : \gamma_1(\phi_1(R(e'_1))), \Sigma_2 : \gamma_2(\phi_2(R(e'_2)))) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} \tag{C.537}$$

$$(\Sigma_1 : \gamma'_1(\phi'_1(R'(e_1))), \Sigma_2 : \gamma'_2(\phi'_2(R'(e_2)))) \in \mathcal{E}[\![\sigma_2]\!]_{R'}^{\Gamma_C} \tag{C.538}$$

for any $R' \in \mathcal{F}[\![\Gamma, x : \sigma_1]\!]^{\Gamma_C}$, $\phi' \in \mathcal{G}[\![\Gamma, x : \sigma_1]\!]_{R'}^{\Sigma_1, \Sigma_2, \Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, x : \sigma_1]\!]_{R'}^{\Sigma_1, \Sigma_2, \Gamma_C}$. Note that in Equation C.537 we have already chosen the substitutions R, ϕ and γ from Equation C.530. By the definition of $\mathcal{F}$, we obtain $R \in \mathcal{F}[\![\Gamma, x : \sigma_1]\!]^{\Gamma_C}$. Therefore, a valid choice for R' is R. From the definitions of $\mathcal{G}$ and $\mathcal{H}$, it must hold that $\phi' = \phi'', x \mapsto (e_3, e_4)$ and $\gamma' = \gamma''$ where $\phi'' \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$, $\gamma'' \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$ and $(\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C}$. We choose $\phi'' = \phi$ and $\gamma'' = \gamma$. It remains to instantiate e_3 and e_4 with concrete choices. For reasons of presentation, we defer this choice to the end of the proof.

From the definition of the $\mathcal{E}$ relation in Equations C.538, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1((\phi_1, x \mapsto e_3)(R(e_1))) : R(\sigma_2) \tag{C.539}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2((\phi_2, x \mapsto e_4)(R(e_2))) : R(\sigma_2) \tag{C.540}$$

$$\begin{aligned} \exists v_5, v_6 : \Sigma_1 &\vdash \gamma_1((\phi_1, x \mapsto e_3)(R(e_1))) \longrightarrow^* v_5 \\ &\wedge \Sigma_2 \vdash \gamma_2((\phi_2, x \mapsto e_4)(R(e_2))) \longrightarrow^* v_6 \\ &\wedge (\Sigma_1 : v_5, \Sigma_2 : v_6) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C} \end{aligned} \tag{C.541}$$

Similarly, from Equation C.537 and from $(\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C}$, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1'))) : R(\sigma_1) \tag{C.542}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2'))) : R(\sigma_1) \tag{C.543}$$

and

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_3 : R(\sigma_1) \tag{C.544}$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_4 : R(\sigma_1) \tag{C.545}$$

Note that from Equations C.544 and C.545 it is evident that expressions e_3 and e_4 contain no free variables. Therefore,

$$\begin{aligned}
&\gamma_1((\phi_1, x \mapsto e_3)(R(e_1))) \\
&\quad = \gamma_1([e_3/x](\phi_1(R(e_1)))) && \text{(by definition)} \\
&\quad = [\gamma_1(e_3)/x](\gamma_1(\phi_1(R(e_1)))) && \text{(distributivity property)} \\
&\quad = [e_3/x](\gamma_1(\phi_1(R(e_1)))) && \text{(no free variables in } e_3\text{)}
\end{aligned}$$

and similarly, $\gamma_2((\phi_2, x \mapsto e_4)(R(e_2))) = [e_4/x]\gamma_2(\phi_2(R(e_2)))$. Taking these equalities into account, we can apply the reverse substitution Lemma 86 on Equations C.539 and C.540, in combination with Equations C.544 and C.545, respectively, to obtain:

$$\Sigma_1; \Gamma_C; \bullet, x : R(\sigma_1) \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma_2) \tag{C.546}$$

$$\Sigma_2; \Gamma_C; \bullet, x : R(\sigma_1) \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\sigma_2) \tag{C.547}$$

Combining Lemma 113 with Equation C.546, yields $\vdash_{ctx} \Sigma_1; \Gamma_C; \bullet, x : R(\sigma_1)$ and, by case analysis on this environment well-formedness judgment, it follows that $\Gamma_C; \bullet \vdash_{ty} R(\sigma_1)$. Using this, Goals C.534 and C.535 follow by applying both Equations C.542 and C.546 and both Equations C.543 and C.547 to rule iTm-let, respectively.

By case analysis, the first step of both evaluation paths in Equation C.536 must be appropriate instantiations of rule rule iEval-let, according to which,

$$\Sigma_i \vdash \textbf{let}\ \ x : R(\sigma_1) = (\gamma_i(\phi_i(R(e_i')))) \ \textbf{in}\ \ (\gamma_i(\phi_i(R(e_i)))) \longrightarrow e_i''$$

where $e_i'' = [\gamma_i(\phi_i(R(e_i')))/x](\gamma_i(\phi_i(R(e_i))))$, for each $i \in \{1, 2\}$. This simplifies Goal C.536 to:

$$\begin{aligned}
\exists v_1, v_2 : \Sigma_1 &\vdash [\gamma_1(\phi_1(R(e_1')))/x](\gamma_1(\phi_1(R(e_1)))) \longrightarrow^* v_1 \\
\wedge\, \Sigma_2 &\vdash [\gamma_2(\phi_2(R(e_2')))/x](\gamma_2(\phi_2(R(e_2)))) \longrightarrow^* v_2 \\
\wedge\, (\Sigma_1 &: v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma_2]\!]_R^{\Gamma_C}
\end{aligned}$$

The goal follows from Equation C.541. Because of Equation C.537, we can choose $e_3 = \gamma_1(\phi_1(R(e_1')))$ and $e_4 = \gamma_2(\phi_2(R(e_2')))$.

□

Lemma 127 (Compatibility - Method)**.**

$$\frac{\begin{array}{c}\Gamma_C ; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : TC\,\sigma \\ (m : TC\,a : \sigma') \in \Gamma_C \\ \Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2\end{array}}{\Gamma_C ; \Gamma \vdash \Sigma_1 : d_1.m \simeq_{log} \Sigma_2 : d_2.m : [\sigma/a]\sigma'}$$

Proof. By inlining the definition of logical equivalence, suppose we have:

$$\begin{gathered} R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \\ \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \\ \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \end{gathered} \tag{C.548}$$

The goal to be proven is the following:

$$(\Sigma_1 : \gamma_1(\phi_1(R(d_1.m))), \Sigma_2 : \gamma_2(\phi_2(R(d_2.m)))) \in \mathcal{E}[\![[\sigma/a]\sigma']\!]_R^{\Gamma_C}$$

By applying the definition of the $\mathcal{E}$ relation, it reduces to:

$$\begin{gathered} \Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(d_1.m))) : R([\sigma/a]\sigma') \\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(d_2.m))) : R([\sigma/a]\sigma') \\ \exists v_1, v_2 : \Sigma_1 \vdash \gamma_1(\phi_1(R(d_1.m))) \longrightarrow^* v_1 \\ \wedge\, \Sigma_2 \vdash \gamma_2(\phi_2(R(d_2.m))) \longrightarrow^* v_2 \\ \wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![[\sigma/a]\sigma']\!]_R^{\Gamma_C} \end{gathered}$$

By applying the substitutions, and because $R([\sigma/a]\sigma') = [R(\sigma)/a]R(\sigma')$, the goal further reduces to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} (\gamma_1(R(d_1))).m : [R(\sigma)/a]R(\sigma') \tag{C.549}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} (\gamma_2(R(d_2))).m : [R(\sigma)/a]R(\sigma') \tag{C.550}$$

$$\begin{aligned} \exists v_1, v_2 : \Sigma_1 &\vdash (\gamma_1(R(d_1))).m \longrightarrow^* v_1 \\ \wedge\, \Sigma_2 &\vdash (\gamma_2(R(d_2))).m \longrightarrow^* v_2 \qquad \text{(C.551)} \\ \wedge\, &(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![[\sigma/a]\sigma']\!]_R^{\Gamma_C} \end{aligned}$$

By inlining the definition of logical equivalence in the first hypothesis of this lemma and choosing R and γ from Equation C.548, we get:

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(TC\,\sigma)]\!]^{\Gamma_C}$$

Then, from the definition of $\mathcal{E}$, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : R(TC\,\sigma) \tag{C.552}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : R(TC\,\sigma) \tag{C.553}$$

$$\begin{aligned} \exists dv_1, dv_2 : \gamma_1(R(d_1)) &\longrightarrow^* dv_1 \\ \wedge\, \gamma_2(R(d_2)) &\longrightarrow^* dv_2, \qquad \text{(C.554)} \\ \wedge\, (\Sigma_1 : dv_1, \Sigma_2 &: dv_2) \in \mathcal{V}[\![R(TC\,\sigma)]\!]^{\Gamma_C} \end{aligned}$$

Unfolding the definition of the $\mathcal{V}$ relation in Equation C.554 results in

$$dv_1 = D\,\overline{\sigma}_j\,\overline{d}_{1\,i}$$

$$dv_2 = D\,\overline{\sigma}_j\,\overline{d}_{2\,i}$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{d}_{1\,i} : R(TC\,\sigma) \tag{C.555}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{d}_{2\,i} : R(TC\,\sigma) \tag{C.556}$$

$$(D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e_1' \in \Sigma_1 \tag{C.557}$$

$$\overline{(\Sigma_1 : d_{1\,i}, \Sigma_2 : d_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Gamma_C}}^{\,i} \tag{C.558}$$

for some $\overline{\sigma}_j$, $\overline{d}_{1\,i}$, $\overline{d}_{2\,i}$, $\overline{C}_i$, e' and σ_q such that $\sigma = [\overline{\sigma}_j/\overline{a}_j]\sigma_q$.

Lemma 114, applied on Equations C.555 and C.556, yields:

$$\vdash_{ctx} \Sigma_1; \Gamma_C; \bullet \tag{C.559}$$

$$\vdash_{ctx} \Sigma_2; \Gamma_C; \bullet \tag{C.560}$$

Also, from the second premise of this lemma's rule, there are Γ_{C1} and Γ_{C2} such that $\Gamma_C = \Gamma_{C1}, m : TC\, a : \sigma', \Gamma_{C2}$. Then, from Lemma 107, we get $\Gamma_{C1}; \bullet, a \vdash_{ty} \sigma'$, which means that the only free variable appearing in σ' is the fresh variable a. Then,

$$R(\sigma') = \sigma' \tag{C.561}$$

Also, since the dictionary $D\, \overline{\sigma}_j\, \overline{d}_{1\,i}$ is closed, types $\overline{\sigma}_j$ can not contain any free variables. Hence, $R(\overline{\sigma}_j) = \overline{\sigma}_j$. In addition, from the last conclusion of Lemma 108, supplied with $\vdash_{ctx} \Sigma_1; \Gamma_C; \bullet$ and Equation C.557, we have that $\Gamma_C; \bullet, \overline{a}_j \vdash_{ty} \sigma_q$. Because the type variables $\overline{a}_j$ are not in Γ, they are not in the domain of R, thus $R(\sigma_q) = \sigma_q$. Then,

$$\begin{aligned} R(\sigma) &= R([\overline{\sigma}_j/\overline{a}_j]\sigma_q) \\ &= [R(\overline{\sigma}_j)/\overline{a}_j]R(\sigma_q) \\ &= [\overline{\sigma}_j/\overline{a}_j]\sigma_q = \sigma \end{aligned} \tag{C.562}$$

We first rewrite Equations C.555 and C.556 with Equation C.562. We then proceed with repeated inversion rule D-CON, rule D-TYAPP and rule D-DAPP to get their premises. Because environment Σ_1 can only contain a unique entry for each dictionary type (in this case, the one shown in Equation C.557), we conclude

$$\overline{\Gamma_C; \bullet \vdash_{ty} \sigma_j}^{j} \tag{C.563}$$

$$\overline{\Sigma_1; \Gamma_C; \bullet \vdash_d d_{1\,i} : [\overline{\sigma}_j/\overline{a}_j]C_i}^{i} \tag{C.564}$$

$$\overline{\Sigma_2; \Gamma_C; \bullet \vdash_d d_{2\,i} : [\overline{\sigma}_j/\overline{a}_j]C_i}^{i} \tag{C.565}$$

$$\Sigma_1'; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e_1 : [\sigma_q/a]\sigma' \tag{C.566}$$

$$\Sigma_2'; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e_2 : [\sigma_q/a]\sigma' \tag{C.567}$$

$$\textit{where } \Sigma_1 = \Sigma_1', (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e_1', \Sigma_1'' \tag{C.568}$$

$$\textit{and } e_1' = \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e_1 \tag{C.569}$$

$$\textit{and } \Sigma_2 = \Sigma_2', (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e_2', \Sigma_2'' \tag{C.570}$$

$$\textit{and } e_2' = \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e_2 \tag{C.571}$$

With Equations C.561 and C.562, Goals C.549 and C.550 follow by using the second hypothesis and Equations C.555 and C.556, respectively, in rule iTm-method.

Using Equations C.568 and C.570 in rule iEval-methodVal, results in:

$$\Sigma_1 \vdash (D\,\overline{\sigma}_j\,d_{1\,i}).m \longrightarrow e_1'\,\overline{\sigma}_j\,d_{1\,i}$$

$$\Sigma_2 \vdash (D\,\overline{\sigma}_j\,d_{2\,i}).m \longrightarrow e_2'\,\overline{\sigma}_j\,d_{2\,i}$$

After applying rule iEval-method and Equation C.554, this reduces Goal C.551 to:

$$\begin{aligned} &\exists v_1, v_2 : \Sigma_1 \vdash e_1'\,\overline{\sigma}_j\,d_{1\,i} \longrightarrow^* v_1 \\ &\wedge \Sigma_2 \vdash e_2'\,\overline{\sigma}_j\,d_{2\,i} \longrightarrow^* v_2 \\ &\wedge (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![[\sigma/a]\sigma']\!]_R^{\Gamma_C} \end{aligned} \tag{C.572}$$

From the definition of logical equivalence in the theorem's third hypothesis, together with Equations C.568 and C.570, we get that:

$$\Gamma_C; \bullet \vdash \Sigma_1' : e_1' \simeq_{log} \Sigma_2' : e_2' : \forall \overline{a}_j.[\sigma_q/a]\sigma' \tag{C.573}$$

Repeatedly applying compatibility Lemma 125 to Equations C.573 and C.563, results in:

$$\Gamma_C; \bullet \vdash \Sigma_1' : e_1'\,\overline{\sigma}_j \simeq_{log} \Sigma_2' : e_2'\,\overline{\sigma}_j : [\overline{\sigma}_j/\overline{a}_j]\overline{C}_i \Rightarrow [\overline{\sigma}_j/\overline{a}_j][\sigma_q/a]\sigma' \tag{C.574}$$

By applying weakening Lemma 100 on this result, in combination with Equations C.559 and C.560, we get:

$$\Gamma_C; \bullet \vdash \Sigma_1 : e_1' \, \overline{\sigma}_j \simeq_{log} \Sigma_2 : e_2' \, \overline{\sigma}_j : [\overline{\sigma}_j / \overline{a}_j]\overline{C}_i \Rightarrow [\overline{\sigma}_j / \overline{a}_j][\sigma_q / a]\sigma' \tag{C.575}$$

From the definition of logical equivalence and Equation C.558, we can derive that:

$$\overline{\Gamma_C; \bullet \vdash \Sigma_1 : d_{1\,i} \simeq_{log} \Sigma_2 : d_{2\,i} : [\overline{\sigma}_j / \overline{a}_j] C_i}^{\,i}$$

Repeatedly applying compatibility Lemma 123 on Equations C.575, together with the above equation, results in:

$$\Gamma_C; \bullet \vdash \Sigma_1 : e_1' \, \overline{\sigma}_j \, \overline{d}_{1\,i} \simeq_{log} \Sigma_2 : e_2' \, \overline{\sigma}_j \, \overline{d}_{2\,i} : [\overline{\sigma}_j / \overline{a}_j][\sigma_q / a]\sigma'$$

Since expressions $e_1' \, \overline{\sigma}_j \, \overline{d}_{1\,i}$ and $e_2' \, \overline{\sigma}_j \, \overline{d}_{2\,i}$ are closed, by the definition of the logical relation, applying any substitutions $R \in \mathcal{F}[\![\bullet]\!]^{\Gamma_C}$, $\phi \in \mathcal{G}[\![\bullet]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$ and $\gamma \in \mathcal{H}[\![\bullet]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$ on both expressions should result in two terms that are related by the $\mathcal{E}$ relation. By case analysis on R, ϕ and γ, only the empty substitutions are valid choices, returning exactly the same expressions. Taking this into account, we have:

$$(\Sigma_1 : e_1' \, \overline{\sigma}_j \, \overline{d}_{1\,i}, \Sigma_2 : e_2' \, \overline{\sigma}_j \, \overline{d}_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_j / \overline{a}_j][\sigma_q / a]\sigma']\!]_R^{\Gamma_C}$$

In turn, unfolding the definition of the $\mathcal{E}$ relation results in:

$$\begin{aligned} \exists v_3, v_4 : \Sigma_1 \vdash e_1' \, \overline{\sigma}_j \, \overline{d}_{1\,i} &\longrightarrow^* v_3 \\ \wedge \, \Sigma_2 \vdash e_2' \, \overline{\sigma}_j \, \overline{d}_{2\,i} &\longrightarrow^* v_4 \\ \wedge \, (\Sigma_1 : v_3, \Sigma_2 : v_4) &\in \mathcal{V}[\![[\overline{\sigma}_j / \overline{a}_j][\sigma / a]\sigma']\!]_R^{\Gamma_C} \end{aligned} \tag{C.576}$$

Goal C.572 follows from Equation C.576 by noting that $[\overline{\sigma}_j / \overline{a}_j][\sigma / a]\sigma' = [[\overline{\sigma}_j / \overline{a}_j]\sigma / a][\overline{\sigma}_j / \overline{a}_j]\sigma' = [[\overline{\sigma}_j / \overline{a}_j]\sigma / a]\sigma'$ and taking $v_3 = v_1$ and $v_4 = v_2$.

□

Lemma 128 (Dictionary Compatibility - Abstraction)**.**

$$\frac{\Gamma_C; \Gamma, \delta : C_1 \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C_2}{\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda \delta : C_1 . d_1 \simeq_{log} \Sigma_2 : \lambda \delta : C_1 . d_2 : C_1 \Rightarrow C_2}$$

Proof. By inlining the definition of logical equivalence, suppose we have

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$$

the goal to be proven is

$$(\Sigma_1 : \gamma_1(R(\lambda\delta : C_1.d_1)), \Sigma_2 : \gamma_2(R(\lambda\delta : C_1.d_2))) \in \mathcal{E}[\![R(C_1 \Rightarrow C_2)]\!]^{\Gamma_C}$$

Unfolding the definition of the $\mathcal{E}$ relation, reduces this goal further

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R(\lambda\delta : C_1.d_1)) : R(C_1 \Rightarrow C_2)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R(\lambda\delta : C_1.d_2)) : R(C_1 \Rightarrow C_2)$$

$$\forall(\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} :$$

$$(\Sigma_1 : (\gamma_1(R(\lambda\delta : C_1.d_1)))\, d_3, \Sigma_2 : (\gamma_2(R(\lambda\delta : C_1.d_2)))\, d_4) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C}$$

Note that $R(C_1 \Rightarrow C_2) = R(C_1) \Rightarrow R(C_2)$. Applying the substitutions simplifies the goal to

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \lambda\delta : R(C_1).\gamma_1(R(d_1)) : R(C_1) \Rightarrow R(C_2) \tag{C.577}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \lambda\delta : R(C_1).\gamma_2(R(d_2)) : R(C_1) \Rightarrow R(C_2) \tag{C.578}$$

$$\forall(\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} :$$

$$(\Sigma_1 : (\lambda\delta : R(C_1).\gamma_1(R(d_1)))\, d_3, \Sigma_2 : (\lambda\delta : R(C_1).\gamma_2(R(d_2)))\, d_4) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C} \tag{C.579}$$

We proceed by applying the same process of inlining the definition of logical equivalence in the rule hypothesis. For some

$$R' \in \mathcal{F}[\![\Gamma, \delta : C_1]\!]^{\Gamma_C}$$

$$\gamma' \in \mathcal{H}[\![\Gamma, \delta : C_1]\!]_{R'}^{\Sigma_1, \Sigma_2, \Gamma_C}$$

we get

$$(\Sigma_1 : \gamma'_1(R'(d_1)), \Sigma_2 : \gamma'_2(R'(d_2))) \in \mathcal{E}[\![R'(C_2)]\!]^{\Gamma_C} \tag{C.580}$$

We take $R' = R$ and $\gamma' = \gamma, \delta \mapsto (d_5, d_6)$ for some d_5 and d_6 where $(\Sigma_1 : d_5, \Sigma_2 : d_6) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C}$ (following their respective definitions). It follows fromt the definition of the $\mathcal{E}$ relation that

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma'_1(R(d_1)) : R(C_2) \tag{C.581}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma'_2(R(d_2)) : R(C_2) \tag{C.582}$$

Furthermore, from the definition of the $\mathcal{E}$ relation, we also get that

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_5 : R(C_1) \tag{C.583}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_6 : R(C_1) \tag{C.584}$$

By Substitution Lemma 92 we can thus derive that

$$\Sigma_1; \Gamma_C; \bullet, \delta : R(C_1) \vdash_d \gamma_1(R(d_1)) : R(C_2)$$

$$\Sigma_2; \Gamma_C; \bullet, \delta : R(C_1) \vdash_d \gamma_2(R(d_2)) : R(C_2)$$

Goals C.577 and C.578 follow from this, in combination with rule D-DABS. We now focus on the remaining Goal C.579. By choosing $d_5 = d_3$ and $d_6 = d_4$, Equation C.580 becomes

$$(\Sigma_1 : \gamma_1([d_3/\delta]R(d_1)), \Sigma_2 : \gamma_2([d_4/\delta]R(d_2))) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C}$$

By Equations C.583 and C.584, this reduces to

$$(\Sigma_1 : [d_3/\delta]\gamma_1(R(d_1)), \Sigma_2 : [d_4/\delta]\gamma_2(R(d_2))) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C} \tag{C.585}$$

We know by rule iDictEval-appAbs that $(\lambda\delta : R(C_1).\gamma_1(R(d_1)))\, d_3 \longrightarrow [d_3/\delta]\gamma_1(R(d_1))$ and $(\lambda\delta : R(C_1).\gamma_2(R(d_2)))\, d_4 \longrightarrow [d_4/\delta]\gamma_2(R(d_2))$ Goal C.579 then follows directly from Theorem 47 and Equation C.585.

□

Lemma 129 (Dictionary Compatibility - Type Abstraction)**.**

$$\frac{\Gamma_C; \Gamma, a \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C}{\Gamma_C; \Gamma \vdash \Sigma_1 : \Lambda a.d_1 \simeq_{log} \Sigma_2 : \Lambda a.d_2 : \forall a.C}$$

Proof. By inlining the definition of logical equivalence, suppose we have

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$$

the goal to be proven is

$$(\Sigma_1 : \gamma_1(R(\Lambda a.d_1)), \Sigma_2 : \gamma_2(R(\Lambda a.d_2))) \in \mathcal{E}[\![R(\forall a.C)]\!]^{\Gamma_C}$$

Unfolding the definition of the $\mathcal{E}$ relation (noting that $R(\forall a.C) = \forall a.R(C)$), reduces this goal further

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R(\Lambda a.d_1)) : R(\forall a.C)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R(\Lambda a.d_2)) : R(\forall a.C)$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow$$

$$(\Sigma_1 : (\gamma_1(R(\Lambda a.d_1)))\, \sigma, \Sigma_2 : (\gamma_2(R(\Lambda a.d_2)))\, \sigma) \in \mathcal{E}[\![[\sigma/a]R(C)]\!]^{\Gamma_C}$$

Applying the substitutions simplifies the goal to

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \Lambda a.\gamma_1(R(d_1)) : \forall a.R(C) \tag{C.586}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \Lambda a.\gamma_2(R(d_2)) : \forall a.R(C) \tag{C.587}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow$$

$$(\Sigma_1 : (\Lambda a.\gamma_1(R(d_1)))\,\sigma, \Sigma_2 : (\Lambda a.\gamma_2(R(d_2)))\,\sigma) \in \mathcal{E}[\![[\sigma/a]R(C)]\!]^{\Gamma_C} \tag{C.588}$$

We proceed by applying the same process of inlining the definition of logical equivalence in the rule hypothesis. For some

$$R' \in \mathcal{F}[\![\Gamma, a]\!]^{\Gamma_C}$$

$$\gamma' \in \mathcal{H}[\![\Gamma, a]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_{R'}$$

we get

$$(\Sigma_1 : \gamma'_1(R'(d_1)), \Sigma_2 : \gamma'_2(R'(d_2))) \in \mathcal{E}[\![R'(C)]\!]^{\Gamma_C} \tag{C.589}$$

We take $R' = R, a \mapsto (\sigma, r)$ and $\gamma' = \gamma$. By the definition of the $\mathcal{E}$ relation we know that

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R'(d_1)) : R'(C) \tag{C.590}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R'(d_2)) : R'(C) \tag{C.591}$$

By Substitution Lemma 94 we can derive that

$$\Sigma_1; \Gamma_C; \bullet, a \vdash_d \gamma_1(R(d_1)) : R(C)$$

$$\Sigma_2; \Gamma_C; \bullet, a \vdash_d \gamma_2(R(d_2)) : R(C)$$

Goals C.586 and C.587 follow from this, in combination with rule D-TYABS. We now focus on the remaining Goal C.588. We can derive from the definition of γ and R that $\gamma_1(R([\sigma/a]d_1)) = [\sigma/a]\gamma_1(R(d_1))$ and $\gamma_2(R([\sigma/a]d_2)) = [\sigma/a]\gamma_2(R(d_2))$. Equation C.589 thus becomes

$$(\Sigma_1 : [\sigma/a]\gamma_1(R(d_1)), \Sigma_2 : [\sigma/a]\gamma_2(R(d_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C} \tag{C.592}$$

We know by rule IDICTEVAL-TYAPPABS that $(\Lambda a.\gamma_1(R(d_1)))\,\sigma \longrightarrow [\sigma/a]\gamma_1(R(d_1))$ and $(\Lambda a.\gamma_2(R(d_2)))\,\sigma \longrightarrow [\sigma/a]\gamma_2(R(d_2))$ Goal C.588 then follows directly from Theorem 47 and Equation C.592.

□

C.7.2 Helper Theorems

Theorem 45 (Congruence - Expressions).
If $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$
and $M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \Gamma' \Rightarrow \sigma')$ *and* $M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \Gamma' \Rightarrow \sigma')$
and $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$
then $\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma'$.

Proof. The goal follows directly from the definition of logical equivalence for contexts.

□

Theorem 46 ($F_{\{\}}$ Context Preserved by Elaboration).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M$
then $\Sigma; \Gamma_C; \Gamma' \vdash_{tm} M[e] : \sigma' \rightsquigarrow M[e]$.

Proof. By structural induction on the typing derivation of M.
$\boxed{\textbf{rule } \textsc{iM-empty}}$ $[\bullet] : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \rightsquigarrow [\bullet]$
We need to show that:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$$

which follows immediately from the first hypothesis of the theorem.
$\boxed{\textbf{rule } \textsc{iM-abs}}$
$\lambda x : \sigma_1 . M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1 \to \sigma') \rightsquigarrow \lambda x : \sigma_1 . M'$

We need to show the following.

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} \lambda x : \sigma_1 . M'[e] : \sigma_1 \to \sigma' \rightsquigarrow \lambda x : \sigma_1 . M'[e]$$

From the premises of rule rule iM-abs, we obtain:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \Rightarrow \sigma') \rightsquigarrow M' \tag{C.593}$$

$$\Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.594}$$

From the induction hypothesis applied on Equation C.593, it follows that:

$$\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e] \tag{C.595}$$

The goal follows from rule iTm-arrI, in combination with Equations C.594 and C.595.
$\boxed{\textbf{rule } \textsc{iM-appL}}$ $M'\, e_2 : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M'\, e_2$
The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} M'[e]\, e_2 : \sigma' \rightsquigarrow M'[e]\, e_2$$

From the premises of rule rule iM-appL, we obtain:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1 \rightarrow \sigma') \rightsquigarrow M' \tag{C.596}$$

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2 \tag{C.597}$$

From the induction hypothesis applied on Equation C.596, it follows that:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} M'[e] : \sigma_1 \rightarrow \sigma' \rightsquigarrow M'[e] \tag{C.598}$$

The goal follows from rule iTm-arrE, in combination with Equations C.597 and C.598.

$\boxed{\textbf{rule } \textsc{iM-appR}}$ $e_1\, M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow e_1\, M'$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1\, M'[e] : \sigma' \rightsquigarrow e_1\, M'[e]$$

From the premises of rule rule iM-appR, we obtain:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M' \tag{C.599}$$

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1 : \sigma_1 \rightarrow \sigma' \rightsquigarrow e_1 \tag{C.600}$$

From the induction hypothesis applied on Equation C.599, it follows that:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} M'[e] : \sigma_1 \rightsquigarrow M'[e] \tag{C.601}$$

The goal follows from rule iTm-arrE, in combination with Equations C.600 and C.601.

$\boxed{\textbf{rule } \textsc{iM-dictAbs}}$

$\lambda\delta : C.M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow C \Rightarrow \sigma') \rightsquigarrow \lambda\delta : \sigma.M'$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} \lambda\delta : C.M'[e] : C \Rightarrow \sigma' \rightsquigarrow \lambda\delta : \sigma.M'[e]$$

From the premises of rule rule iM-dictAbs, we obtain:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', \delta : C \Rightarrow \sigma') \rightsquigarrow M' \tag{C.602}$$

$$\Gamma_C; \Gamma' \vdash_C C \rightsquigarrow \sigma \tag{C.603}$$

From the induction hypothesis applied on Equation C.602, it follows that:

$$\Sigma; \Gamma_C; \Gamma', \delta : C \vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e] \tag{C.604}$$

The goal follows from rule iTm-constrI, in combination with Equations C.603 and C.604.

$\boxed{\textbf{rule } \text{IM-DICTAPP}}$ $M'\,d : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \sigma') \rightsquigarrow M'\,e$
The goal to be proven is the following.

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} M'[e]\,d : \sigma' \rightsquigarrow M'[e]\,e'$$

From the premises of rule rule IM-DICTAPP, we obtain:

$$M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow C \Rightarrow \sigma') \rightsquigarrow M' \tag{C.605}$$

$$\Sigma;\Gamma_C;\Gamma' \vdash_d d : C \rightsquigarrow e' \tag{C.606}$$

From the induction hypothesis applied on Equation C.605, it follows that:

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} M'[e] : C \Rightarrow \sigma' \rightsquigarrow M'[e] \tag{C.607}$$

The goal follows from rule ITM-CONSTRE, in combination with Equations C.606 and C.607.
$\boxed{\textbf{rule } \text{IM-TYABS}}$ $\Lambda a.M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \forall a.\sigma') \rightsquigarrow \Lambda a.M'$
The goal to be proven is the following:

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} \Lambda a.M'[e] : \forall a.\sigma' \rightsquigarrow \Lambda a.M'[e]$$

From the premises of rule rule IM-TYABS, we obtain:

$$M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma', a \Rightarrow \sigma') \rightsquigarrow M'$$

Applying the induction hypothesis on the above context typing, yields:

$$\Sigma;\Gamma_C;\Gamma', a \vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e]$$

Using this result with rule rule ITM-FORALLI, we reach the goal.
$\boxed{\textbf{rule } \text{IM-TYAPP}}$ $M'\,\sigma'' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow [\sigma''/a]\sigma') \rightsquigarrow M'\,\sigma''$
The goal to be proven is the following.

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} M'[e]\,\sigma'' : [\sigma''/a]\sigma' \rightsquigarrow M'[e]\,\sigma''$$

From the premises of rule rule IM-TYAPP, we obtain:

$$M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma' \Rightarrow \forall a.\sigma') \rightsquigarrow M' \tag{C.608}$$

$$\Gamma_C;\Gamma' \vdash_{ty} \sigma'' \rightsquigarrow \sigma'' \tag{C.609}$$

From the induction hypothesis applied on Equation C.608, we have that:

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} M'[e] : \forall a.\sigma' \rightsquigarrow M'[e] \tag{C.610}$$

The goal follows from rule ITM-FORALLE, in combination with Equations C.609 and C.610.

$\boxed{\textbf{rule } \textsc{iM-let}\text{L}}$

$\textbf{let } x : \sigma_1 = M' \textbf{ in } e_2 : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow \textbf{let } x : \sigma_1 = M' \textbf{ in } e_2$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} \textbf{let } x : \sigma_1 = M'[e] \textbf{ in } e_2 : \sigma' \rightsquigarrow \textbf{let } x : \sigma_1 = M'[e] \textbf{ in } e_2$$

From the premises of rule rule iM-letL, we obtain:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma_1) \rightsquigarrow M' \tag{C.611}$$

$$\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \vdash_{tm} e_2 : \sigma' \rightsquigarrow e_2 \tag{C.612}$$

$$\Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.613}$$

From the induction hypothesis applied on Equation C.611, we have that:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} M'[e] : \sigma_1 \rightsquigarrow M'[e] \tag{C.614}$$

The goal follows from rule iTm-let, in combination with Equations C.612, C.613 and C.614.

$\boxed{\textbf{rule } \textsc{iM-let}\text{R}}$

$\textbf{let } x : \sigma_1 = e_1 \textbf{ in } M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow \textbf{let } x : \sigma_1 = e_1 \textbf{ in } M'$

The goal to be proven is the following:

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} \textbf{let } x : \sigma_1 = e_1 \textbf{ in } M'[e] : \sigma' \rightsquigarrow \textbf{let } x : \sigma_1 = e_1 \textbf{ in } M'[e]$$

From the rule premise:

$$M' : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \Rightarrow \sigma') \rightsquigarrow M' \tag{C.615}$$

$$\Sigma; \Gamma_C; \Gamma' \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \tag{C.616}$$

$$\Gamma_C; \Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.617}$$

From the induction hypothesis applied on Equation C.615, we have that:

$$\Sigma; \Gamma_C; \Gamma', x : \sigma_1 \vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e] \tag{C.618}$$

The goal follows from rule iTm-let, in combination with Equations C.616, C.617 and C.618. □

Lemma 130 (Closed Dictionary Relation Preserved by Forward/Backward Reduction)**.**
Given $d_1 \longrightarrow d_1'$ *and* $d_2 \longrightarrow d_2'$*,*

- *If* $(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C]\!]^{\Gamma_C}$*, then* $(\Sigma_1 : d_1', \Sigma_2 : d_2') \in \mathcal{E}[\![C]\!]^{\Gamma_C}$*.*
- *If* $(\Sigma_1 : d_1', \Sigma_2 : d_2') \in \mathcal{E}[\![C]\!]^{\Gamma_C}$ *and* $\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : C$ *and* $\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : C$*, then* $(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C]\!]^{\Gamma_C}$*.*

Proof. Proof by induction on the size of the constraint C.

Part 1 By case analysis on the $\mathcal{E}$ relation in the hypothesis.

$\boxed{C = C_1 \Rightarrow C_2}$ $\quad (\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![C_1 \Rightarrow C_2]\!]^{\Gamma_C}$

By unfolding the definition of the $\mathcal{E}$, the goal to be proven is

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1' : C_1 \Rightarrow C_2 \tag{C.619}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2' : C_1 \Rightarrow C_2 \tag{C.620}$$

$$\forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![C_1]\!]^{\Gamma_C} \Rightarrow (\Sigma_1 : d_1'\, d_3, \Sigma_2 : d_2'\, d_4) \in \mathcal{E}[\![C_2]\!]^{\Gamma_C} \tag{C.621}$$

while the hypotheses tell us

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : C_1 \Rightarrow C_2 \tag{C.622}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : C_1 \Rightarrow C_2 \tag{C.623}$$

$$\forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![C_1]\!]^{\Gamma_C} \Rightarrow (\Sigma_1 : d_1\, d_3, \Sigma_2 : d_2\, d_4) \in \mathcal{E}[\![C_2]\!]^{\Gamma_C} \tag{C.624}$$

Goals C.619 and C.620 follow by Theorem 29 and Equation C.622 and C.623 respectively.

We know by rule iDictEval-app that $d_1\, d_3 \longrightarrow d_1'\, d_3$ and $d_2\, d_4 \longrightarrow d_2'\, d_4$. Goal C.621 thus follows by applying the induction hypothesis to Equation C.624 (note that C_2 is trivially smaller than $C_1 \Rightarrow C_2$).

$\boxed{C = \forall a.C'}$ $\quad (\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![\forall a.C']\!]^{\Gamma_C}$

By unfolding the definition of the $\mathcal{E}$, the goal to be proven is

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1' : \forall a.C' \tag{C.625}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2' : \forall a.C' \tag{C.626}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow (\Sigma_1 : d_1'\, \sigma, \Sigma_2 : d_2'\, \sigma) \in \mathcal{E}[\![[\sigma/a]C']\!]^{\Gamma_C} \tag{C.627}$$

while the hypotheses tell us

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : \forall a.C' \tag{C.628}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : \forall a.C' \tag{C.629}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow (\Sigma_1 : d_1\,\sigma, \Sigma_2 : d_2\,\sigma) \in \mathcal{E}[\![[\sigma/a]C']\!]^{\Gamma_C} \tag{C.630}$$

Goals C.625 and C.626 follow by Theorem 29 and Equation C.628 and C.629 respectively.

We know by rule iDictEval-tyApp that $d_1\,\sigma \longrightarrow d_1'\,\sigma$ and $d_2\,\sigma \longrightarrow d_2'\,\sigma$. Goal C.627 thus follows by applying the induction hypothesis to Equation C.630. Note that we define the size of $TC\,\sigma$ to be 1, independently of the choice of σ. This way $[\sigma/a]C'$ is guaranteed to be smaller than $\forall a.C'$.

$\boxed{C = Q}$ $(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![Q]\!]^{\Gamma_C}$

By unfolding the definition of the $\mathcal{E}$, the goal to be proven is

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1' : Q \tag{C.631}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2' : Q \tag{C.632}$$

$$\exists dv_1', dv_2', d_1' \longrightarrow^* dv_1', d_2' \longrightarrow^* dv_2', (\Sigma_1 : dv_1', \Sigma_2 : dv_2') \in \mathcal{V}[\![Q]\!]^{\Gamma_C} \tag{C.633}$$

while the hypotheses tell us

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : Q \tag{C.634}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : Q \tag{C.635}$$

$$\exists dv_1, dv_2, d_1 \longrightarrow^* dv_1, d_2 \longrightarrow^* dv_2, (\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]^{\Gamma_C} \tag{C.636}$$

Goals C.631 and C.626 follow by Theorem 29 and Equation C.634 and C.635 respectively.

As the evaluation of dictionaries is deterministic, we know that $dv_1' = dv_1$, $dv_2' = dv_2$. Furthermore, this fact also tells us that $d_1 \longrightarrow d_1'$ is the first step of $d_1 \longrightarrow^* dv_1$, and similarly for d_2. Goal C.633 thus follows directly by Equation C.636.

Part 2 Similar to Part 1.

□

Theorem 47 (Logical Equivalence for Dictionaries Preserved by Forward/Backward Reduction)**.**
Given $d_1 \longrightarrow d'_1$ *and* $d_2 \longrightarrow d'_2$*,*

- *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C$*,*
  then $\Gamma_C; \Gamma \vdash \Sigma_1 : d'_1 \simeq_{log} \Sigma_2 : d'_2 : C$*.*
- *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : d'_1 \simeq_{log} \Sigma_2 : d'_2 : C$ *and* $\Sigma_1; \Gamma_C; \Gamma \vdash_d d_1 : C$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_d d_2 : C$*, then* $\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C$*.*

Proof. **Part 1** By unfolding the definition of logical equivalence, the goal to be proven becomes

$$(\Sigma_1 : \gamma_1(R(d'_1)), \Sigma_2 : \gamma_2(R(d'_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

while the hypothesis gives us

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

for any $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$. As substitution preserves evaluation, we know that $\gamma_1(R(d_1)) \longrightarrow \gamma_1(R(d'_1))$ and $\gamma_2(R(d_2)) \longrightarrow \gamma_2(R(d'_2))$. The goal thus follows directly by Lemma 130.

Part 2 Similar to Part 1.

□

Theorem 48 (Logical Equivalence for Expressions Preserved by Forward/Backward Reduction)**.**
Given $\Sigma_1 \vdash e_1 \longrightarrow e'_1$ *and* $\Sigma_2 \vdash e_2 \longrightarrow e'_2$*,*

- *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$*,*
  then $\Gamma_C; \Gamma \vdash \Sigma_1 : e'_1 \simeq_{log} \Sigma_2 : e'_2 : \sigma$*.*
- *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e'_1 \simeq_{log} \Sigma_2 : e'_2 : \sigma$ *and* $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma$*, then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$*.*

Proof. **Part 1** By unfolding the definition of logical relation, we get:

$$R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C} \tag{C.637}$$

$$\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.638}$$

$$\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} \tag{C.639}$$

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.640}$$

Unfolding the definition of the closed expression relation in C.640 results in:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1))) : R(\sigma) \tag{C.641}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2))) : R(\sigma) \tag{C.642}$$

$$\exists v_1, v_2, \Sigma_1 \vdash \gamma_1(\phi_1(R(e_1))) \longrightarrow^* v_1, \tag{C.643}$$

$$\Sigma_2 \vdash \gamma_2(\phi_2(R(e_2))) \longrightarrow^* v_2, \tag{C.644}$$

$$(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.645}$$

By induction on e_1, it is easy to verify that

$$\Sigma_1 \vdash \gamma_1(\phi_1(R(e_1))) \longrightarrow \gamma_1(\phi_1(R(e_1'))) \tag{C.646}$$

By preservation (Theorem 30) we have:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \gamma_1(\phi_1(R(e_1'))) : R(\sigma) \tag{C.647}$$

Because the evaluation in $F_{\mathbf{D}}$ is deterministic (Lemma 110), we know that:

$$\Sigma_1 \vdash \gamma_1(\phi_1(R(e_1'))) \longrightarrow^* v_1 \tag{C.648}$$

Similarly:

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \gamma_2(\phi_2(R(e_2'))) : R(\sigma) \tag{C.649}$$

$$\Sigma_2 \vdash \gamma_2(\phi_2(R(e_2'))) \longrightarrow^* v_2 \tag{C.650}$$

Combining those equations, results in:

$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1'))), \Sigma_2 : \gamma_2(\phi_2(R(e_2')))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.651}$$

The goal follows from the definition of logical equivalence.

Part 2 Similar to Part 1.

□

Theorem 49 (Dictionary Reflexivity).
If $\Sigma_1; \Gamma_C; \Gamma \vdash_d d : C$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_d d : C$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$*, then* $\Gamma_C; \Gamma \vdash \Sigma_1 : d \simeq_{log} \Sigma_2 : d : C$*.*

Proof. Proof by structural induction on the dictionary d and consequently, since $F_{\mathbf{D}}$ dictionary typing is syntax directed, on both typing derivations.
$\boxed{d = \delta \textbf{ (rule } \text{D-VAR}\textbf{)}}$ $\Sigma_1; \Gamma_C; \Gamma \vdash_d \delta : C \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d \delta : C$
The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \delta \simeq_{log} \Sigma_2 : \delta : C$$

By unfolding the definition of logical equivalence, the goal reduces to:

$$(\Sigma_1 : \gamma_1(R(\delta)), \Sigma_2 : \gamma_2(R(\delta))) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

where $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$.

From the given we know that $(\delta : C) \in \Gamma$. Because of this, it follows from the definition of $\mathcal{H}$ that:

$$\gamma_1(R(\delta)) = d_1$$

$$\gamma_2(R(\delta)) = d_2$$

$$(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![R(C)]\!]^{\Gamma_C}$$

$\boxed{d = D \textbf{ (rule } \text{D-CON}\textbf{)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : D \simeq_{log} \Sigma_2 : D : \forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q$$

By unfolding the definition of logical equivalence, the goal reduces to:

$$(\Sigma_1 : \gamma_1(R(D)), \Sigma_2 : \gamma_2(R(D))) \in \mathcal{E}[\![R(\forall \overline{a}_j . \overline{C}_i \Rightarrow TC\,\sigma_q)]\!]^{\Gamma_C}$$

where $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$.

Note that as D does not depend on any dictionary variables, not on any type variables, we know that $\gamma_1(R(D)) = \gamma_2(R(D)) = D$.

Repeatedly unfolding the $\mathcal{E}$ relation in the goal reduces it to

$$\Sigma_1; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{d}_{1\,i} : TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q \tag{C.652}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{d}_{2\,i} : TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q \tag{C.653}$$

$$\exists dv_1, dv_2, D\,\overline{\sigma}_j\,\overline{d}_{1\,i} \longrightarrow^* dv_1, D\,\overline{\sigma}_j\,\overline{d}_{2\,i} \longrightarrow^* dv_2, \tag{C.654}$$

$$(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q]\!]^{\Gamma_C}$$

for any $\overline{d}_{1\,i}$, $\overline{d}_{2\,i}$ and $\overline{\sigma}_j$ where

$$\overline{(\Sigma_1 : d_{1\,i}, \Sigma_2 : d_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_j/\overline{a}_j]C_i]\!]^{\Gamma_C}}^{\,i} \tag{C.655}$$

$$\overline{\Gamma_C; \bullet \vdash_{ty} \sigma_j}^{\,j} \tag{C.656}$$

Goals C.652 and C.653 follow by repeated application of rule D-DAPP and rule D-TYAPP, in combination with the first two hypotheses.

We will thus focus on Goal C.654 from here on out. Note that $D\,\overline{\sigma}_j\,\overline{d}_{1\,i}$ and $D\,\overline{\sigma}_j\,\overline{d}_{2\,i}$ are already dictionary values. We can thus take $dv_1 = D\,\overline{\sigma}_j\,\overline{d}_{1\,i}$ and $dv_2 = D\,\overline{d}_j\,\overline{d}_{2\,i}$.

Unfolding the definition of the $\mathcal{V}$ relation in Goal C.654 results in a repeat of Goals C.652 and C.653 (which have previously been proven), in addition to

$$\overline{(\Sigma_1 : d_{1\,i}, \Sigma_2 : d_{2\,i}) \in \mathcal{E}[\![[\overline{\sigma}_j/\overline{b}_j]C'_i]\!]^{\Gamma_C}}^{\,i} \tag{C.657}$$

$$(D : \forall\overline{b}_j.\overline{C}'_i \Rightarrow Q').m' \mapsto e'_1 \in \Sigma_1 \tag{C.658}$$

$$TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q = [\overline{\sigma}_j/\overline{b}_j]Q' \tag{C.659}$$

By inversion on the first hypothesis, we know that $(D : \forall\overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto e_1 \in \Sigma_1$. By Lemma 114 we also know that $\vdash_{ctx} \Sigma_1; \Gamma_C; \Gamma$. From these results, as D is unique in Σ_1, Goal C.658 follows with $\overline{b}_j = \overline{a}_j$, $\overline{C}'_i = \overline{C}_i$ and $Q' = TC\,\sigma_q$.

Goal C.659 follows directly from this result. Goal C.657 is proven by Equation C.655.

$\boxed{d = \lambda\delta : C_1.d' \ \textbf{(rule } \text{D-DABS}\textbf{)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_d \lambda\delta : C_1.d' : C_1 \Rightarrow C_2 \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d \lambda\delta : C_1.d' : C_1 \Rightarrow C_2$$

The goal to be proven is the following

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda\delta : C_1.d' \simeq_{log} \Sigma_2 : \lambda\delta : C_1.d' : C_1 \Rightarrow C_2$$

Inlining the definition of logical equivalence reduces the goal to

$$(\Sigma_1 : \gamma_1(R(\lambda\delta : C_1.d')), \Sigma_2 : \gamma_2(R(\lambda\delta : C_1.d'))) \in \mathcal{E}[\![R(C_1 \Rightarrow C_2)]\!]^{\Gamma_C}$$

For any $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$. Noting that $R(C_1 \Rightarrow C_2) = R(C_1) \Rightarrow R(C_2)$, we reduce the goal further by inlining the definition of the $\mathcal{E}$ relation

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R(\lambda\delta : C_1.d')) : R(C_1) \Rightarrow R(C_2) \tag{C.660}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R(\lambda\delta : C_1.d')) : R(C_1) \Rightarrow R(C_2) \tag{C.661}$$

$$\forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} \Rightarrow \tag{C.662}$$

$$(\Sigma_1 : (\gamma_1(R(\lambda\delta : C_1.d')))\, d_3, \Sigma_2 : (\gamma_2(R(\lambda\delta : C_1.d')))\, d_4) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C}$$

From the rule premise we know that

$$\Sigma_1; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d' : C_2$$

$$\Sigma_2; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d' : C_2$$

$$\Gamma_C; \Gamma \vdash_C C_1$$

By applying the induction hypothesis to this, we thus get

$$\Gamma_C; \Gamma, \delta : C_1 \vdash \Sigma_1 : d' \simeq_{log} \Sigma_2 : d' : C_2$$

Unfolding the definition of logical equivalence reduces this to

$$(\Sigma_1 : \gamma'_1(R'(d')), \Sigma_2 : \gamma'_2(R'(d'))) \in \mathcal{E}[\![R'(C_2)]\!]^{\Gamma_C}$$

For any $R' \in \mathcal{F}[\![\Gamma, \delta : C_1]\!]^{\Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, \delta : C_1]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$. Following their respective definitions, we take $R' = R$ and $\gamma' = \gamma, \delta \mapsto (d'_3, d'_4)$, for some d'_3 and d'_4 such that $(\Sigma_1 : d'_3, \Sigma_2 : d'_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C}$.

Inlining these definitions and working out the substitutions thus gives us

$$(\Sigma_1 : \gamma_1([d'_3/\delta]R(d')), \Sigma_2 : \gamma_2([d'_4/\delta]R(d'))) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C} \tag{C.663}$$

However, as we know from the definition of the $\mathcal{E}$ relation that $\Sigma_1; \Gamma_C; \bullet \vdash_d d'_3 : R(C_1)$ and $\Sigma_2; \Gamma_C; \bullet \vdash_d d'_4 : R(C_1)$, we can derive that $\gamma_1([d'_3/\delta]R(d')) = [d'_3/\delta]\gamma_1(R(d'))$ and $\gamma_2([d'_4/\delta]R(d')) = [d'_4/\delta]\gamma_2(R(d'))$. From the definition of the $\mathcal{E}$ relation, we thus get

$$\Sigma_1; \Gamma_C; \bullet \vdash_d [d'_3/\delta]\gamma_1(R(d')) : R(C_2)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d [d'_4/\delta]\gamma_2(R(d')) : R(C_2)$$

Using Lemma 92 we derive

$$\Sigma_1; \Gamma_C; \bullet, \delta : R(C_1) \vdash_d \gamma_1(R(d')) : R(C_2)$$

$$\Sigma_2; \Gamma_C; \bullet, \delta : R(C_1) \vdash_d \gamma_2(R(d')) : R(C_2)$$

Goals C.660 and C.661 follow from this result using rule D-DABS. We now focus on proving Goal C.662. Simplifying the substitutions reduces the remaining goal to

$$\forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} \Rightarrow$$

$$(\Sigma_1 : (\lambda\delta : R(C_1).\gamma_1(R(d')))\, d_3, \Sigma_2 : (\lambda\delta : R(C_1).\gamma_2(R(d')))\, d_4) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C}$$

As both d'_3 and d'_4 are typed under an empty context, we can also derive that $\gamma_1([d'_3/\delta]R(d')) = [d'_3/\delta]\gamma_1(R(d'))$ and $\gamma_2([d'_4/\delta]R(d')) = [d'_4/\delta]\gamma_1(R(d'))$. By rule iDICTEVAL-APPABS, we know $(\lambda\delta : R(C_1).\gamma_1(R(d')))\, d_3 \longrightarrow [d_3/\delta]\gamma_1(R(d'))$ and $(\lambda\delta : R(C_1).\gamma_2(R(d')))\, d_4 \longrightarrow [d_4/\delta]\gamma_2(R(d'))$. By taking $d'_3 = d_3$ and $d'_4 = d_4$, Goal C.662 follows from Lemma 130 and Equation C.663 (the well-typedness of $(\lambda\delta : R(C_1).\gamma_1(R(d')))\, d_3$ and $(\lambda\delta : R(C_1).\gamma_2(R(d')))\, d_4$ follows by rule D-DAPP and Equations C.660 and C.661 respectively).

$\boxed{d = d_1\, d_2 \textbf{ (rule D-DAPP)}}$ $\Sigma_1; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2$

The goal to be proven is the following

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d_1\, d_2 \simeq_{log} \Sigma_2 : d_1\, d_2 : C_2$$

Unfolding the definition of logical equivalence reduces this to

$$(\Sigma_1 : \gamma_1(R(d_1\, d_2)), \Sigma_2 : \gamma_2(R(d_1\, d_2))) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C} \tag{C.664}$$

for any $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1, \Sigma_2, \Gamma_C}$. Applying the induction hypothesis to the rule premises gives us

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_1 : C_1 \Rightarrow C_2$$

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d_2 \simeq_{log} \Sigma_2 : d_2 : C_1$$

Again unfolding the definition of logical equivalence, and choosing an identical R and γ thus gives us

$$(\Sigma_1 : \gamma_1(R(d_1)), \Sigma_2 : \gamma_2(R(d_1))) \in \mathcal{E}[\![R(C_1 \Rightarrow C_2)]\!]^{\Gamma_C} \tag{C.665}$$

$$(\Sigma_1 : \gamma_1(R(d_2)), \Sigma_2 : \gamma_2(R(d_2))) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} \tag{C.666}$$

Unfolding the definition of the $\mathcal{E}$ relation in Equation C.665 (note that $R(C_1 \Rightarrow C_2) = R(C_1) \Rightarrow R(C_2)$) teaches us that

$$\forall d_3, d_4 : (\Sigma_1 : d_3, \Sigma_2 : d_4) \in \mathcal{E}[\![R(C_1)]\!]^{\Gamma_C} \Rightarrow$$

$$(\Sigma_1 : (\gamma_1(R(d_1)))\, d_3, \Sigma_2 : (\gamma_2(R(d_1)))\, d_4) \in \mathcal{E}[\![R(C_2)]\!]^{\Gamma_C}$$

By taking $d_3 = \gamma_1(R(d_2))$ and $d_4 = \gamma_2(R(d_2))$, Goal C.664 follows directly from this result.

$\boxed{d = \Lambda a.d' \ \textbf{(rule } \text{D-TYABS}\textbf{)}}$ $\quad \Sigma_1; \Gamma_C; \Gamma \vdash_d \Lambda a.d' : \forall a.C' \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d \Lambda a.d' : \forall a.C'$

The goal to be proven is the following

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \Lambda a.d' \simeq_{log} \Sigma_2 : \Lambda a.d' : \forall a.C'$$

Unfolding the definition of logical equivalence reduces this to

$$(\Sigma_1 : \gamma_1(R(\Lambda a.d')), \Sigma_2 : \gamma_2(R(\Lambda a.d'))) \in \mathcal{E}[\![R(\forall a.C')]\!]^{\Gamma_C}$$

for any $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$. We proceed by simplifying the substitutions and unfolding the definition of the $\mathcal{E}$ relation (note that $R(\forall a.C') = \forall a.R(C')$) to reduce the goal to

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \Lambda a.\gamma_1(R(d')) : \forall a.R(C') \tag{C.667}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \Lambda a.\gamma_2(R(d')) : \forall a.R(C') \tag{C.668}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow (\Sigma_1 : (\Lambda a.\gamma_1(R(d')))\, \sigma, \Sigma_2 : (\Lambda a.\gamma_2(R(d')))\, \sigma) \in \mathcal{E}[\![[\sigma/a]R(C')]\!]^{\Gamma_C} \tag{C.669}$$

By applying the induction hypothesis to the rule premise, we get that

$$\Gamma_C; \Gamma, a \vdash \Sigma_1 : d' \simeq_{log} \Sigma_2 : d' : C'$$

Unfolding the definition of logical equivalence thus gives us

$$(\Sigma_1 : \gamma_1'(R'(d')), \Sigma_2 : \gamma_2'(R'(d'))) \in \mathcal{E}[\![R'(C')]\!]^{\Gamma_C}$$

for any $R' \in \mathcal{F}[\![\Gamma, a]\!]^{\Gamma_C}$ and $\gamma' \in \mathcal{H}[\![\Gamma, a]\!]_{R'}^{\Sigma_1,\Sigma_2,\Gamma_C}$. Following their respective definitions, we take $\gamma' = \gamma$ and $R' = R, a \mapsto (\sigma, r)$ where $\Gamma_C; \bullet \vdash_{ty} \sigma$. By inlining these definitions we thus have

$$(\Sigma_1 : \gamma_1(R([\sigma/a]d')), \Sigma_2 : \gamma_2(R([\sigma/a]d'))) \in \mathcal{E}[\![R([\sigma/a]C')]\!]^{\Gamma_C}$$

By noting that σ is well-formed under an empty typing context, we know that $\gamma_1(R([\sigma/a]d')) = [\sigma/a]\gamma_1(R(d'))$, $\gamma_2(R([\sigma/a]d')) = [\sigma/a]\gamma_2(R(d'))$ and $R([\sigma/a]C') = [\sigma/a]R(C')$. The previous result thus becomes

$$(\Sigma_1 : [\sigma/a]\gamma_1(R(d')), \Sigma_2 : [\sigma/a]\gamma_2(R(d'))) \in \mathcal{E}[\![[\sigma/a]R(C')]\!]^{\Gamma_C} \tag{C.670}$$

By the definition of the $\mathcal{E}$ relation, we know that

$$\Sigma_1; \Gamma_C; \bullet \vdash_d [\sigma/a]\gamma_1(R(d')) : [\sigma/a]R(C')$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d [\sigma/a]\gamma_2(R(d')) : [\sigma/a]R(C')$$

Goals C.667 and C.668 then follow by this result and Lemma 94. Furthermore, as we know from rule iDictEval-tyAppAbs that $(\Lambda a.\gamma_1(R(d')))\,\sigma \longrightarrow [\sigma/a]\gamma_1(R(d'))$ and $(\Lambda a.\gamma_2(R(d')))\,\sigma \longrightarrow [\sigma/a]\gamma_2(R(d'))$, Goal C.669 follows by Lemma 130 and Equation C.670.

$\boxed{d = d'\,\sigma \textbf{ (rule } \text{D-TYAPP}\textbf{)}}$ $\Sigma_1; \Gamma_C; \Gamma \vdash_d d'\,\sigma : [\sigma/a]C' \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_d d'\,\sigma : [\sigma/a]C'$
The goal to be proven is

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d'\,\sigma \simeq_{log} \Sigma_2 : d'\,\sigma : [\sigma/a]C'$$

Unfolding the definition of logical equivalence reduces this to

$$(\Sigma_1 : \gamma_1(R(d'\,\sigma)), \Sigma_2 : \gamma_2(R(d'\,\sigma))) \in \mathcal{E}[\![R([\sigma/a]C')]\!]^{\Gamma_C} \tag{C.671}$$

Applying the induction hypothesis to the rule premise gives us

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d' \simeq_{log} \Sigma_2 : d' : \forall a.C'$$

Unfolding the definition of logical equivalence thus gives

$$(\Sigma_1 : \gamma_1(R(d')), \Sigma_2 : \gamma_2(R(d'))) \in \mathcal{E}[\![R(\forall a.C')]\!]^{\Gamma_C}$$

Noting that $R(\forall a.C') = \forall a.R(C')$, we can further reduce this by inlining the definition of the $\mathcal{E}$ relation.

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R(d')) : \forall a.R(C') \tag{C.672}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R(d')) : \forall a.R(C') \tag{C.673}$$

$$\forall \sigma : \Gamma_C; \bullet \vdash_{ty} \sigma \Rightarrow (\Sigma_1 : (\gamma_1(R(d')))\,\sigma, \Sigma_2 : (\gamma_2(R(d')))\,\sigma) \in \mathcal{E}[\![[\sigma/a]R(C')]\!]^{\Gamma_C} \tag{C.674}$$

As σ is well-formed under the empty typing environment, we can conclude that $(\gamma_1(R(d')))\,\sigma = \gamma_1(R(d'\,\sigma))$, $(\gamma_2(R(d')))\,\sigma = \gamma_2(R(d'\,\sigma))$ and $[\sigma/a]R(C') = R([\sigma/a]C')$. Goal C.671 thus follows directly from Equation C.674.

□

Theorem 50 (Expression Reflexivity)**.**
If $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$*, then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e \simeq_{log} \Sigma_2 : e : \sigma$*.*

Proof. The proof proceeds by induction on e and consequently, since $F_{\mathbf{D}}$ term typing is syntax directed, on both typing derivations.

$\boxed{e = \mathit{True}\ \textbf{(rule } \text{ITM-TRUE}\textbf{)}}$
$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}$

The goal to prove is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \mathit{True} \simeq_{log} \Sigma_2 : \mathit{True} : \mathit{Bool}$$

Unfolding the definition of logical equivalence in the above, results in the following goal:

$$(\Sigma_1 : \gamma_1(\phi_1(R(\mathit{True}))), \Sigma_2 : \gamma_2(\phi_2(R(\mathit{True})))) \in \mathcal{E}[\![\mathit{Bool}]\!]_R^{\Gamma_C} \tag{C.675}$$

where $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$, $\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$. However, since True does not contain any free variables, we know that $\gamma_1(\phi_1(R(\mathit{True}))) = \gamma_2(\phi_2(R(\mathit{True}))) = \mathit{True}$. Similarly, it follows that $R(\mathit{Bool}) = \mathit{Bool}$.

Unfolding the definition of the $\mathcal{E}$ relation in Goal C.675, reduces the goal to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \mathit{True} : \mathit{Bool} \tag{C.676}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} \mathit{True} : \mathit{Bool} \tag{C.677}$$

$$\exists v_1, v_2 : \Sigma_1 \vdash \mathit{True} \longrightarrow^* v_1$$

$$\wedge\, \Sigma_2 \vdash \mathit{True} \longrightarrow^* v_2$$

$$\wedge\, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

Goals C.676 and C.677 are satisified from the first and second hypotheses of the theorem. We set $v_1 = v_2 = \mathit{True}$ and since True is a value, the term reductions above hold. Then, the last goal follows directly from the definition of the $\mathcal{V}$ relation, according to which the following holds trivially.

$$(\Sigma_1 : \mathit{True}, \Sigma_2 : \mathit{True}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

$\boxed{e = \mathit{False}\ \textbf{(rule } \text{ITM-FALSE}\textbf{)}}$
$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool}$

The proof is similar to the rule ITM-TRUE case.

$\boxed{e = x\ \textbf{(rule } \text{ITM-VAR}\textbf{)}}$ $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} x : \sigma \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} x : \sigma$
The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : x \simeq_{log} \Sigma_2 : x : \sigma$$

By unfolding the definition of logical equivalence in the above, we have

$$(\Sigma_1 : \gamma_1(\phi_1(R(x))), \Sigma_2 : \gamma_2(\phi_2(R(x)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

for any $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$, $\phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$. From the definition of the $\mathcal{G}$ relation, we know that:

$$\gamma_1(\phi_1(R(x))) = e_1$$

$$\gamma_2(\phi_2(R(x))) = e_2$$

$$(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \tag{C.678}$$

The goal follows directly from Equation C.678.

$\boxed{e = \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 \ \textbf{(rule } \text{iTm-let}\textbf{)}}$

$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 : \sigma_2 \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 : \sigma_2$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 \simeq_{log} \Sigma_2 : \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 : \sigma_2$$

By applying the induction hypothesis in the premises of the two rule iTm-let rules, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_1 : \sigma_1$$

$$\Gamma_C; \Gamma, x : \sigma_1 \vdash \Sigma_1 : e_2 \simeq_{log} \Sigma_2 : e_2 : \sigma_2$$

The goal follows directly by passing the above two Equations to compatibility Lemma 126.

$\boxed{e = d.m \ \textbf{(rule } \text{iTm-method}\textbf{)}}$

$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma' \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma'$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d.m \simeq_{log} \Sigma_2 : d.m : [\sigma/a]\sigma'$$

From the premises of rule rule iTm-method we have that

$$\Sigma_1; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma \tag{C.679}$$

$$\Sigma_2; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma \tag{C.680}$$

$$(m : TC\,a : \sigma') \in \Gamma_C \tag{C.681}$$

Applying the Dictionary Reflexivity (Theorem 49) to Equations C.679 and C.680, in combination with the theorem's third hypothesis, results in:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d \simeq_{log} \Sigma_2 : d : TC\,\sigma \tag{C.682}$$

The goal follows directly from compatibility Lemma 127 and Equations C.681 and C.682, in combination with the third hypothesis.

$\boxed{e = \lambda x : \sigma_1 . e' \textbf{ (rule } \textsc{iTm-arrI}\textbf{)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1 . e' : \sigma_1 \to \sigma_2 \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1 . e' : \sigma_1 \to \sigma_2$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda x : \sigma_1 . e' \simeq_{log} \Sigma_2 : \lambda x : \sigma_1 . e' : \sigma_1 \to \sigma_2$$

By applying the induction hypothesis to the premises of the two rule iTm-arrI rules, we get:

$$\Gamma_C; \Gamma, x : \sigma_1 \vdash \Sigma_1 : e' \simeq_{log} \Sigma_2 : e' : \sigma_2$$

The goal follows by applying the above to compatibility Lemma 120.

$\boxed{e = e_1\, e_2 \textbf{ (rule } \textsc{iTm-arrE}\textbf{)}}$ $\quad \Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1\, e_2 \simeq_{log} \Sigma_2 : e_1\, e_2 : \sigma_2$$

By applying the induction hypothesis to the premises of the two rule iTm-arrE rules, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_1 : \sigma_1 \to \sigma_2$$

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_2 \simeq_{log} \Sigma_2 : e_2 : \sigma_1$$

The goal follows by passing the above two equations to compatibility Lemma 121.

$\boxed{e = \lambda \delta : C . e' \textbf{ (rule } \textsc{iTm-constrI}\textbf{)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \lambda \delta : C . e' : C \Rightarrow \sigma \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \lambda \delta : C . e' : C \Rightarrow \sigma$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda \delta : C . e' \simeq_{log} \Sigma_2 : \lambda \delta : C . e' : C \Rightarrow \sigma$$

By applying the induction hypothesis to the premises of the two rule iTm-constrI rules, we get:

$$\Gamma_C; \Gamma, \delta : C \vdash \Sigma_1 : e' \simeq_{log} \Sigma_2 : e' : \sigma$$

The goal follows directly by passing the above equation to compatibility Lemma 122.

$\boxed{e = e'\ d \text{ (rule ITM-CONSTRE)}}$ $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e'\ d : \sigma \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e'\ d : \sigma$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e'\ d \simeq_{log} \Sigma_2 : e'\ d : \sigma$$

By applying the induction hypothesis to the premises of the two rule iTm-constrE rules, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e' \simeq_{log} \Sigma_2 : e' : C \Rightarrow \sigma \tag{C.683}$$

Furthermore, applying Dictionary Reflexivity (Theorem 49) in the premises of the two rule iTm-constrE rules, in combination with the theorem's third hypothesis, results in:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d \simeq_{log} \Sigma_2 : d : C \tag{C.684}$$

The goal follows from compatibility Lemma 123 and Equations C.683 and C.684.

$\boxed{e = \Lambda a.e' \text{ (rule ITM-FORALLI)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e' : \forall a.\sigma \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e' : \forall a.\sigma$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \Lambda a.e' \simeq_{log} \Sigma_2 : \Lambda a.e' : \forall a.\sigma$$

By applying the induction hypothesis to the premises of the two rule iTm-forallI rules, we get:

$$\Gamma_C; \Gamma, a \vdash \Sigma_1 : e' \simeq_{log} \Sigma_2 : e' : \sigma_1$$

The goal follows directly by applying the above equation to compatibility Lemma 124.

$\boxed{e = e'\ \sigma \text{ (rule ITM-FORALLE)}}$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e'\ \sigma : [\sigma/a]\sigma' \wedge \Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e'\ \sigma : [\sigma/a]\sigma'$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e'\ \sigma \simeq_{log} \Sigma_2 : e'\ \sigma : [\sigma/a]\sigma'$$

From the premises of rule iTm-forallE, we obtain

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \tag{C.685}$$

By applying the induction hypothesis to the premises of the two rule iTmForallE rules, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e'_1 \simeq_{log} \Sigma_2 : e'_1 : \sigma_1 \tag{C.686}$$

The goal follows from compatibility Lemma 125 and Equations C.686 and C.685. □

Theorem 51 (Context Reflexivity)**.**
Suppose $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma'$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma' \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma'$ *and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma' \vdash^M_{ty} \tau' \rightsquigarrow \sigma'$,

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_2$
then $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$.

- *If* $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_2$
then $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$.

- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_2$
then $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$.

- *If* $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_2$
then $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma')$.

Proof. The theorem is stated in a nested fashion, where all common hypotheses are introduced in the outer statement. Each of the four inner statements extends the outer statement with two context-typing hypotheses, and sets the conclusion of the theorem, which is identical for each of the four cases.

Suppose expressions e_1 and e_2 such that

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \tag{C.687}$$

Then, by unfolding the defintion of logical equivalence in the goal of all four sub-statements, it suffices to show that

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma' \tag{C.688}$$

We assume all hypotheses of the outer statement and we proceed by mutual induction on the first hypothesis of the nested statements. Note that context typing derivations are of finite size, thus mutual induction over them is safe.

Part 1

$\boxed{\textbf{rule } \textsc{sM-inf-inf-empty}}$ $[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$

By case analysis on the second hypothesis of the nested statement, its last context typing rule must be rule sM-INF-INF-EMPTY as well. Therefore, the first and second hypotheses of the nested statement become

$$[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$$

$$[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$$

and we have $M_1 = M_2 = [\bullet]$. Thus, Goal C.688 becomes

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$$

The above logical equivalence follows directly from Equation C.687.

$\boxed{\textbf{rule } \textsc{sM-inf-inf-appL}}$

$M' \, e_2' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1' \, e_{21}$

By case analysis on the second hypothesis of the nested statement, its last context typing rule must be rule sM-INF-INF-APPL as well. Therefore, the first and second hypotheses of the nested statement become

$$M' \, e_2' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1' \, e_{21}$$

$$M' \, e_2' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_2' \, e_{22}$$

and Goal C.688 becomes

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1'[e_1] \, e_{21} \simeq_{log} \Sigma_2 : M_2'[e_2] \, e_{22} : \sigma_2 \tag{C.689}$$

From the premises of the two rule sM-INF-INF-APPL rules, we obtain:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_1 \to \tau_2) \rightsquigarrow M_1' \tag{C.690}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_1' \to \tau_2) \rightsquigarrow M_2' \tag{C.691}$$

$$P; \Gamma_C; \Gamma' \vdash_{tm}^{M} e_2' \Leftarrow \tau_1 \rightsquigarrow e_{21} \tag{C.692}$$

$$P; \Gamma_C; \Gamma' \vdash_{tm}^{M} e_2' \Leftarrow \tau_1' \rightsquigarrow e_{22} \tag{C.693}$$

By applying Lemma 67 to Equations C.690 and C.691, we know that $\tau_1' \to \tau_2 = \tau_1 \to \tau_2$.

Applying the induction hypothesis on Equations C.690 and C.691, yields:

$$\Sigma_1 : M'_1 \simeq_{log} \Sigma_2 : M'_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma_1 \to \sigma_2)$$

By unfolding the definition of logical equivalence in the above equation and applying it on Equation C.687, we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M'_1[e_1] \simeq_{log} \Sigma_2 : M'_2[e_2] : \sigma_1 \to \sigma_2 \tag{C.694}$$

By applying Expression Coherence Theorem A (Theorem 58) on Equations C.692 and C.693 (and on the second and fourth hypotheses of the outer statement), we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : e_{21} \simeq_{log} \Sigma_2 : e_{22} : \sigma_1 \tag{C.695}$$

Goal C.689 follows from compatibility of term applications (Lemma 121, together with Equations C.694 and C.695).

$\boxed{\textbf{rule}\ \textsc{sM-inf-inf-appR}}$

$e'_1\, M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow e_{11}\, M'_1$

By case analysis on the second hypothesis of the nested statement, its last context typing rule must be rule sM-inf-inf-appR as well. Therefore, the first and second hypotheses of the nested statement become

$$e'_1\, M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow e_{11}\, M'_1$$

$$e'_1\, M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow e_{12}\, M'_2$$

and we need to show that

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : e_{11}\, M'_1[e_1] \simeq_{log} \Sigma_2 : e_{12}\, M'_2[e_2] : \sigma_2 \tag{C.696}$$

From the premises of the two rule sM-inf-inf-appR rules, we obtain:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1) \rightsquigarrow M'_1 \tag{C.697}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau'_1) \rightsquigarrow M'_2 \tag{C.698}$$

$$P; \Gamma_C; \Gamma' \vdash^{M}_{tm} e'_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_{11} \tag{C.699}$$

$$P; \Gamma_C; \Gamma' \vdash^{M}_{tm} e'_1 \Rightarrow \tau'_1 \to \tau_2 \rightsquigarrow e_{12} \tag{C.700}$$

By Lemma 67, we know that $\tau'_1 = \tau_1$.

By applying Part 2 of this theorem to Equations C.697 and C.698, we get:

$$\Sigma_1 : M'_1 \simeq_{log} \Sigma_2 : M'_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma' \Rightarrow \sigma_1)$$

By unfolding the definition of logical equivalence in the above and applying it on Equation C.687, we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M'_1[e_1] \simeq_{log} \Sigma_2 : M'_2[e_2] : \sigma_1 \tag{C.701}$$

By applying Expression Coherence Theorem A (Theorem 58) to Equations C.699 and C.700 (and on the second and fourth hypotheses of the outer statement), we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : e_{11} \simeq_{log} \Sigma_2 : e_{12} : \sigma_1 \to \sigma_2 \tag{C.702}$$

Goal C.696 follows from compatibility of term applications (Lemma 121, together with Equations C.701 and C.702).

rule sM-inf-inf-letL

let $x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M'$ **in** $e'_2 : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1$

where $M_1 =$ **let** $x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.M'_1$ **in** e_{21}.

By case analysis on the second hypothesis of the nested statement, its last context typing rule must be rule sM-inf-inf-letL as well. Therefore, the first and second hypotheses of the nested statement become

let $x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M'$ **in** $e'_2 : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1$

let $x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = M'$ **in** $e'_2 : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_2$

where $M_2 =$ **let** $x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.M'_2$ **in** e_{22}.

Goal C.688 becomes

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma_2 \tag{C.703}$$

From the premises of the two rule sM-inf-inf-letL rules, we obtain:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \rightsquigarrow M'_1 \tag{C.704}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Leftarrow \tau_1) \rightsquigarrow M'_2 \tag{C.705}$$

$$P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash^{M}_{tm} e'_2 \Rightarrow \tau_2 \rightsquigarrow e_{21} \tag{C.706}$$

$$P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \vdash^{M}_{tm} e'_2 \Rightarrow \tau_2 \rightsquigarrow e_{22} \tag{C.707}$$

$$\Gamma_C; \Gamma' \vdash^{M}_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \tag{C.708}$$

$$\overline{\delta}_i \ \textbf{fresh} \tag{C.709}$$

$$x \notin \textbf{dom}(\Gamma') \tag{C.710}$$

Through repeated case analysis on Equation C.708 (rule sTy-scheme and rule sTy-qual), we know that

$$\overline{a}_j \notin \Gamma'$$

$$\overline{\Gamma_C; \Gamma', \overline{a}_j \vdash^M_C C_i \rightsquigarrow C_i}^{\,i}$$

$$\Gamma_C; \Gamma' \vdash^M_{ty} \tau_1 \rightsquigarrow \sigma_1 \tag{C.711}$$

By repeated case analysis on the second hypothesis, we get that

$$\vdash^M_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet; \Gamma_C; \bullet \tag{C.712}$$

By rule sCtx-tyEnvTy and rule sCtx-tyEnvD, in combination with these results and Equation C.709, we obtain $\vdash^M_{ctx} \bullet; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \bullet; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i$. Finally, Lemma 66, together with this result and the second and fourth hypothesis, teaches us that:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \tag{C.713}$$

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \tag{C.714}$$

By applying Part 2 of this theorem to Equations C.704 and C.705, in combination with Equations C.711, C.713 and C.714 and the first, third and fifth hypothesis, we get:

$$\Sigma_1 : M'_1 \simeq_{log} \Sigma_2 : M'_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \Rightarrow \sigma_1)$$

By unfolding the definition of logical equivalence in the above and applying it on Equation C.687, we get:

$$\Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash \Sigma_1 : M'_1[e_1] \simeq_{log} \Sigma_2 : M'_2[e_2] : \sigma_1 \tag{C.715}$$

By repeatedly applying compatibility Lemmas 122 and 124, we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.M'_1[e_1] \simeq_{log} \Sigma_2 : \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.M'_2[e_2] : \forall\overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \tag{C.716}$$

It follows from rule sCtx-tyEnvTm, in combination with Equations C.712, C.710 and C.708, that $\vdash^M_{ctx} \bullet; \Gamma_C; \bullet, x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \bullet; \Gamma_C; \bullet, x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \sigma_1$. Similarly to before, by applying Lemma 66 to this result, together with the second and fourth hypothesis, we get:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \tag{C.717}$$

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma', x : \forall\overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \tag{C.718}$$

By applying Expression Coherence Theorem A (Theorem 58) to Equations C.706 and C.707, together with Equations C.717 and C.718, we get:

$$\Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \vdash \Sigma_1 : e_{21} \simeq_{log} \Sigma_2 : e_{22} : \sigma_2 \tag{C.719}$$

Goal C.703 follows from compatibility of let expressions (Lemma 126, together with Equations C.715 and C.719).

$\boxed{\textbf{rule } \textsc{sM-inf-inf-letR}}$

$$\textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1' \textbf{ in } \ M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1$$

where $M_1 = \textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e_{11} \textbf{ in } \ M_1'$.

By case analysis on the second hypothesis of the nested statement, its last context typing rule must be rule sM-inf-inf-letR as well. Therefore, the first and second hypotheses of the nested statement become

$$\textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1' \textbf{ in } \ M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_1$$

$$\textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 = e_1' \textbf{ in } \ M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau_2) \rightsquigarrow M_2$$

where $M_2 = \textbf{let } \ x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 = \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e_{12} \textbf{ in } \ M_2'$.

Goal C.688 becomes

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma_2 \tag{C.720}$$

From the premises of the two rule sM-inf-inf-letR rules, we obtain:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M_1' \tag{C.721}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \Rightarrow \tau_2) \rightsquigarrow M_2' \tag{C.722}$$

$$P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash^{M}_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_{11} \tag{C.723}$$

$$P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash^{M}_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_{12} \tag{C.724}$$

$$\Gamma_C; \Gamma' \vdash^{M}_{ty} \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma_1 \tag{C.725}$$

$$\overline{\delta}_i \textbf{ fresh} \tag{C.726}$$

$$x \notin \textbf{dom}(\Gamma') \tag{C.727}$$

By repeated case analysis on the second hypothesis, we get that

$$\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet \rightsquigarrow \bullet; \Gamma_C; \bullet \tag{C.728}$$

By rule sCtx-tyEnvTm, in combination with this result and Equations C.725 and C.727, we obtain $\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet, x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \bullet; \Gamma_C; \bullet, x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1$. Lemma 66, together with this result and the second and fourth hypothesis, teaches us that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \tag{C.729}$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \tag{C.730}$$

By applying the induction hypothesis to Equations C.721 and C.722, in combination with Equations C.729 and C.730, we get:

$$\Sigma_1 : M'_1 \simeq_{log} \Sigma_2 : M'_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \Rightarrow \sigma_2)$$

By unfolding the definition of logical equivalence in the above and then applying it on Equation C.687, we get:

$$\Gamma_C; \Gamma', x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \vdash \Sigma_1 : M'_1[e_1] \simeq_{log} \Sigma_2 : M'_2[e_2] : \sigma_2 \tag{C.731}$$

Through repeated case analysis on Equation C.725 (rule sTy-scheme and rule sTy-qual), we know that

$$\overline{a}_j \notin \Gamma'$$

$$\overline{\Gamma_C; \Gamma', \overline{a}_j \vdash^{M}_{C} C_i \rightsquigarrow C_i}^{\,i}$$

$$\Gamma_C; \Gamma' \vdash^{M}_{ty} \tau_1 \rightsquigarrow \sigma_1 \tag{C.732}$$

By rule sCtx-tyEnvTy and rule sCtx-tyEnvD, in combination with these results and Equations C.726 and C.728, we obtain $\vdash^{M}_{ctx} \bullet; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \bullet; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i$. Lemma 66, together with this result and the second and fourth hypothesis, teaches us that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \tag{C.733}$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \tag{C.734}$$

By applying Expression Coherence Theorem A (Theorem 58) to Equations C.723 and C.724, in combination with Equations C.733 and C.734, we get:

$$\Gamma_C; \Gamma', \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash \Sigma_1 : e_{11} \simeq_{log} \Sigma_2 : e_{12} : \sigma_1 \tag{C.735}$$

By repeatedly applying compatibility Lemmas 122 and 124, we get:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e_{11} \simeq_{log} \Sigma_2 : \Lambda \overline{a}_j . \lambda \overline{\delta}_i : \overline{C}_i . e_{12} : \forall \overline{a}_j . \overline{C}_i \Rightarrow \sigma_1 \tag{C.736}$$

Goal C.720 follows from Lemma 126, together with Equations C.731 and C.736.

$\boxed{\textbf{rule } \textsc{sM-inf-inf-ann}}$ $M' :: \tau' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_1$

By case analysis, we know the final step in the second derivation has to be rule sM-inf-inf-ann as well. This means that:

$$M' :: \tau' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_1 \tag{C.737}$$

$$M' :: \tau' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_2 \tag{C.738}$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma' \tag{C.739}$$

From the rule premise we know that:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_1 \tag{C.740}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_2 \tag{C.741}$$

Goal C.739 follows directly from Part 2 of this theorem, in combination with Equations C.740 and C.741.

Part 2 By case analysis on the first typing derivation.

$\boxed{\textbf{rule } \textsc{sM-inf-check-abs}}$

$$\lambda x . M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1 \to \tau_2) \rightsquigarrow \lambda x : \sigma_1 . M_1'$$

By case analysis, we know that the final step in the second derivation has to be either rule sM-inf-check-abs or rule sM-inf-check-inf. Note however that no matching inference rules exist. The final step in the second derivation thus has to be rule sM-inf-check-abs as well. This means that:

$$\lambda x . M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1 \to \tau_2) \rightsquigarrow \lambda x : \sigma_1 . M_1' \tag{C.742}$$

$$\lambda x . M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau_1 \to \tau_2) \rightsquigarrow \lambda x : \sigma_1 . M_2' \tag{C.743}$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma' \vdash \Sigma_1 : \lambda x : \sigma_1 . M'_1[e_1] \simeq_{log} \Sigma_2 : \lambda x : \sigma_1 . M'_2[e_2] : \sigma_1 \to \sigma_2 \tag{C.744}$$

From the rule premise we know that:

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \tau_1 \Leftarrow \tau_2) \rightsquigarrow M'_1 \tag{C.745}$$

$$M' : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma', x : \tau_1 \Leftarrow \tau_2) \rightsquigarrow M'_2 \tag{C.746}$$

$$\Gamma_C; \Gamma' \vdash^M_{ty} \tau_1 \rightsquigarrow \sigma_1 \tag{C.747}$$

We know from rule sCtx-tyEnvTm, in combination with Equation C.747 and the 4th and 6th hypothesis that:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', x : \tau_1 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma', x : \sigma_1 \tag{C.748}$$

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma', x : \tau_1 \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma', x : \sigma_1 \tag{C.749}$$

By applying the induction hypothesis to Equations C.745 and C.746, in combination with Equations C.748 and C.749, we get:

$$\Sigma_1 : M'_1 \simeq_{log} \Sigma_2 : M'_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \Gamma', x : \sigma_1 \Rightarrow \sigma_2) \tag{C.750}$$

By unfolding the definition of logical equivalence in Equation C.750, we get:

$$\forall e'_1, e'_2 : \Gamma_C; \Gamma \vdash \Sigma_1 : e'_1 \simeq_{log} \Sigma_2 : e'_2 : \sigma \tag{C.751}$$

$$\Rightarrow \Gamma_C; \Gamma', x : \sigma_1 \vdash \Sigma_1 : M'_1[e'_1] \simeq_{log} \Sigma_2 : M'_2[e'_2] : \sigma_2 \tag{C.752}$$

This result, together with Equation C.687, tells us that:

$$\Gamma_C; \Gamma', x : \sigma_1 \vdash \Sigma_1 : M'_1[e_1] \simeq_{log} \Sigma_2 : M'_2[e_2] : \sigma_2 \tag{C.753}$$

Goal C.744 follows from Lemma 120, together with Equation C.753.

$\boxed{\textbf{rule } \text{sM-inf-check-inf}}$ $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_1$

By case analysis, we know that the final step in the second derivation has to be either rule sM-inf-check-abs or rule sM-inf-check-inf. Note however that in the case of rule sM-inf-check-abs, M would have to be of the form $M = \lambda x . M'$. In this case, no matching inference rules exist, meaning that this is an impossible case. Consequently, the final step in the second derivation can only be rule sM-inf-check-inf. This means that:

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_1 \tag{C.754}$$

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M_2 \tag{C.755}$$

The goal to be proven is the following:

$$\Gamma_C ; \Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma' \tag{C.756}$$

From the rule premise we know that:

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_1 \tag{C.757}$$

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M_2 \tag{C.758}$$

Goal C.756 follows directly by applying Part 1 of this theorem to Equations C.757 and C.758.

Part 3 By case analysis on the first typing derivation.
Similar to Part 1.

Part 4 By case analysis on the first typing derivation.
Similar to Part 2.

□

Lemma 131 (Class Constraint Dictionary Canonical Form)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_d d : Q$ *then* $\exists dv : d \longrightarrow^* dv$
where $dv = D\,\overline{\sigma}_m\,\overline{d}_n$ *and* $\Sigma; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_m\,\overline{d}_n : Q$*.*

Proof. By applying Strong Normalization Theorem 34, we get

$$d \longrightarrow^* dv$$

Repeatedly applying Type Preservation Theorem 29 to this derivation gives us

$$\Sigma; \Gamma_C; \bullet \vdash_d dv : Q$$

Case analysis on this result makes it clear that $dv = D\,\overline{\sigma}_m\,\overline{d}_n$ for some D, $\overline{\sigma}_m$ and $\overline{d}_n$.

□

Theorem 52 (Closed Class Constraint Dictionary Relation)**.**
If $\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : Q$ *and* $\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : Q$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$
then $(\Sigma_1 : d_1, \Sigma_2 : d_2) \in \mathcal{E}[\![Q]\!]^{\Gamma_C}$*.*

Proof. By induction on the size of d_1 and d_2.

By applying Lemma 131, we know that

$$d_1 \longrightarrow^* D_1\,\overline{\sigma}_{m\,1}\,\overline{d}_{n\,1} \tag{C.759}$$

$$\Sigma;\Gamma_C;\bullet \vdash_d D_1\,\overline{\sigma}_{m\,1}\,\overline{d}_{n\,1} : Q \tag{C.760}$$

$$d_2 \longrightarrow^* D_2\,\overline{\sigma}_{p\,2}\,\overline{d}_{q\,2} \tag{C.761}$$

$$\Sigma;\Gamma_C;\bullet \vdash_d D_2\,\overline{\sigma}_{p\,2}\,\overline{d}_{q\,2} : Q \tag{C.762}$$

For some D_1, D_2, $\overline{\sigma}_{m\,1}$, $\overline{\sigma}_{p\,2}$, $\overline{d}_{n\,1}$ and $\overline{d}_{q\,2}$.

By repeated case analysis on Equations C.760 and C.762, we know that:

$$(D_1 : \forall\overline{a}_{m\,1}.\overline{C}_{n\,1} \Rightarrow Q_1).m_1 \mapsto e_1 \in \Sigma_1 \;\; \textit{where}\;\; Q = [\overline{\sigma}_{m\,1}/\overline{a}_{m\,1}]Q_1 \tag{C.763}$$

$$\overline{\Gamma_C;\bullet \vdash_{ty} \sigma_{i\,1}}^{\,i<m} \tag{C.764}$$

$$\overline{\Sigma_1;\Gamma_C;\bullet \vdash_d d_{i\,1} : [\overline{\sigma}_{m\,1}/\overline{a}_{m\,1}]C_{i\,1}}^{\,i<n} \tag{C.765}$$

$$\Sigma_{11};\Gamma_C;\bullet \vdash_{tm} e_1 : \forall\overline{a}_{m\,1}.\overline{Q}_{n\,1} \Rightarrow \sigma_1$$

$$\textit{where}\;\; \Sigma_1 = \Sigma_{11}, (D_1 : \forall\overline{a}_{m\,1}.\overline{C}_{n\,1} \Rightarrow Q_1).m_1 \mapsto e_1, \Sigma_{12} \tag{C.766}$$

$$\vdash_{ctx} \Sigma_1;\Gamma_C;\bullet \tag{C.767}$$

$$(D_2 : \forall\overline{a}_{p\,2}.\overline{C}_{q\,2} \Rightarrow Q_2).m_2 \mapsto e_2 \in \Sigma_2 \;\; \textit{where}\;\; Q = [\overline{\sigma}_{p\,2}/\overline{a}_{q\,2}]Q_2 \tag{C.768}$$

$$\overline{\Gamma_C;\bullet \vdash_{ty} \sigma_{i\,2}}^{\,i<p} \tag{C.769}$$

$$\overline{\Sigma_2;\Gamma_C;\bullet \vdash_d d_{i\,2} : [\overline{\sigma}_{p\,2}/\overline{a}_{p\,2}]C_{i\,2}}^{\,i<q} \tag{C.770}$$

$$\Sigma_{21};\Gamma_C;\bullet \vdash_{tm} e_2 : \forall\overline{a}_{p\,2}.\overline{C}_{q\,2} \Rightarrow \sigma_1$$

$$\textit{where}\;\; \Sigma_2 = \Sigma_{21}, (D_2 : \forall\overline{a}_{p\,2}.\overline{C}_{q\,2} \Rightarrow Q_2).m_2 \mapsto e_2, \Sigma_{22} \tag{C.771}$$

$$\vdash_{ctx} \Sigma_2;\Gamma_C;\bullet \tag{C.772}$$

By case analysis on Equations C.767 and C.772 (rule iCtx-MEnv) and the definition of logical equivalence in the 3rd hypothesis, it follows from

Equations C.763 and C.768 that:

$$D_1 = D_2$$

$$\overline{a}_{m\,1} = \overline{a}_{p\,2}$$

$$\overline{C}_{n\,1} = \overline{C}_{q\,2}$$

$$Q_1 = Q_2$$

$$m_1 = m_2$$

$$\Gamma_C \vdash \Sigma_{11} \simeq_{log} \Sigma_{21}$$

$$\Gamma_C \vdash \Sigma_{12} \simeq_{log} \Sigma_{22}$$

$$\Gamma_C; \bullet \vdash \Sigma_{11} : e_1 \simeq_{log} \Sigma_{21} : e_2 : \forall \overline{a}_{m\,1}.\overline{C}_{n\,1} \Rightarrow \sigma_1$$

Consequently, we also know that $m = p$ and $n = q$.

Furthermore, rule rule iCtx-MEnv also tells us that **unambig**$(\forall \overline{a}_{m\,1}.\overline{C}_{n\,1} \Rightarrow Q_1)$. The definition of unambiguity thus gives us $\overline{a}_{m\,1} \in \mathbf{fv}(Q_1)$. This, in combination with Equations C.763 and C.768 tells us that $\overline{\sigma}_{m\,1} = \overline{\sigma}_{p\,2}$.

Unfolding the definition of the $\mathcal{E}$, followed by the $\mathcal{V}$ relation, reduces the goal to be proven to:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d d_1 : Q \tag{C.773}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d d_2 : Q \tag{C.774}$$

$$d_1 \longrightarrow^* D_1\, \overline{\sigma}_{m\,1}\, \overline{d}_{n\,1} \tag{C.775}$$

$$d_2 \longrightarrow^* D_2\, \overline{\sigma}_{p\,2}\, \overline{d}_{q\,2} \tag{C.776}$$

$$\overline{(\Sigma_1 : d_{i\,1}, \Sigma_2 : d_{i\,2}) \in \mathcal{E}[\![[\overline{\sigma}_{m\,1}/\overline{a}_m] C_i]\!]^{\Gamma_C}}^{\,i<n} \tag{C.777}$$

$$\Sigma_1; \Gamma_C; \bullet \vdash_d D_1\, \overline{\sigma}_{m\,1}\, \overline{d}_{n\,1} : Q \tag{C.778}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d D_2\, \overline{\sigma}_{p\,2}\, \overline{d}_{q\,2} : Q \tag{C.779}$$

$$(D : \forall \overline{a}_{m\,1}.\overline{C}_{n\,1} \Rightarrow Q_1).m_1 \mapsto e_1 \in \Sigma_1 \ \textit{where}\ Q = [\overline{\sigma}_{m\,1}/\overline{a}_{m\,1}]Q' \tag{C.780}$$

Goals C.773 and C.774 follow directly from the 1st and 2nd hypothesis. Goals C.775 and C.776 are given by Equations C.759 and C.761. Goals C.778 and C.779 are given by Equations C.760 and C.762. Goal C.780 follows directly

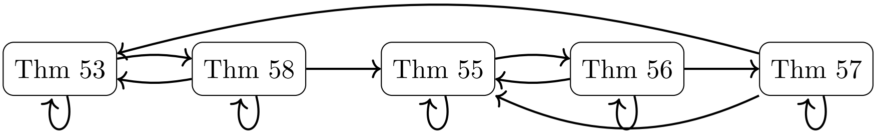

Figure C.4: Dependency graph for Coherence Theorems

from Equation C.763. Finally, Goal C.777 follows by applying the induction hypothesis on Equations C.765 and C.770.

□

Theorem 53 (Environment Equivalence Preservation)**.**
If $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$
then $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$*.*

Proof. By structural induction on P and mutually proven with Theorems 58, 55, 56 and 57 (see Figure C.4). Note that at the dependency between Theorem 53 and 58, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Theorems 55, 56 and 57 perform induction on the given derivation, which we assume to be finite. Consequently, the induction remains well-founded.

$\boxed{P = \bullet}$

By case analysis on the 1st and 2nd hypothesis:

$$\Sigma_1 = \Sigma_2 = \bullet$$

The goal follows from rule CTXLOG-EMPTY.

$\boxed{P = P', (D : C).m \mapsto \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}_y : e}$

By case analysis on the 1st and 2nd hypothesis (rule SCTX-PGMINST):

$$\Sigma_1 = \Sigma_1', (D : C).m \mapsto \Lambda\overline{b}_j.\lambda\overline{\delta}_i : \overline{C}_i.\Lambda\overline{a}_k.\lambda\overline{\delta}_y : [\sigma/a]\overline{C}_y.e_1$$

$$\Sigma_2 = \Sigma_2', (D : C').m \mapsto \Lambda\overline{b}_j.\lambda\overline{\delta}_i : \overline{C}'_i.\Lambda\overline{a}_k.\lambda\overline{\delta}_y : [\sigma/a]\overline{C}'_y.e_2$$

$$P'; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}_y \vdash^{M}_{tm} e \Leftarrow [\tau/a]\tau' \rightsquigarrow e_1 \quad \text{(C.781)}$$

$$P'; \Gamma_C; \bullet, \overline{b}_j, \overline{\delta}_i : \overline{C}_i, \overline{a}_k, \overline{\delta}_y : [\tau/a]\overline{C}_y \vdash^{M}_{tm} e \Leftarrow [\tau/a]\tau' \rightsquigarrow e_2 \quad \text{(C.782)}$$

$$\vdash^{M}_{ctx} P'; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1'; \Gamma_C; \Gamma \quad \text{(C.783)}$$

$$\vdash^{M}_{ctx} P'; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2'; \Gamma_C; \Gamma \quad \text{(C.784)}$$

Since the elaboration from $\lambda^{\Rightarrow}_{\mathbf{TC}}$ constraints to $F_{\mathbf{D}}$ constraints is entirely deterministic (Lemma 70), we know that $C' = C$, $\overline{C}'_i = \overline{C}_i$, $\overline{C}'_y = \overline{C}_y$ and consequently that $[\sigma/a]\overline{C}'_y = [\sigma/a]\overline{C}_y$.

From the induction hypothesis, together with Equations C.783 and C.784, we get that:

$$\Gamma_C \vdash \Sigma'_1 \simeq_{log} \Sigma'_2 \tag{C.785}$$

From Expression Coherence Theorem A (Theorem 58), in combination with Equations C.781, C.782, C.783 and C.784, we know:

$$\Gamma_C; \Gamma \vdash \Sigma'_1 : e_1 \simeq_{log} \Sigma'_2 : e_2 : [\sigma/a]\sigma' \tag{C.786}$$

where $\Gamma_C; \Gamma \vdash^M_{ty} [\tau/a]\tau' \rightsquigarrow [\sigma/a]\sigma'$.

The goal follows from rule ctxLog-cons, together with Equations C.785 and C.786.

□

> **Theorem 54** (Contextual Equivalence in $F_{\mathbf{D}}$ Implies Contextual Equivalence in $\lambda^{\Rightarrow}_{\mathbf{TC}}$).
> *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
> *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$
> *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
> *and there exists an* τ *such that* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$
> *then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Proof. By unfolding the definition of contextual equivalence, the goal becomes:

$$\forall M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_1 \tag{C.787}$$

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow Bool) \rightsquigarrow M_2 \tag{C.788}$$

$$\textit{then } M_1[e_1] \simeq M_2[e_2] \tag{C.789}$$

We thus assume Equations C.787 and C.788 and prove Equation C.789.

By unfolding the definition of contextual equivalence in the first hypothesis, we get that:

$$\forall M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1$$

$$\forall M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2$$

$$\mathit{if}\ \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow \mathit{Bool})$$

$$\mathit{then}\ M_1[e_1] \simeq M_2[e_2] \tag{C.790}$$

By applying Lemma 80 to the fourth hypothesis, we know that:

$$\vdash_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \bullet$$

By applying context equivalence (Theorem 44) on Equations C.787 and C.788, together with this result and hypotheses 2, 3 and 4, we know that:

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1' \tag{C.791}$$

$$M_1' : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1' \tag{C.792}$$

$$M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2' \tag{C.793}$$

$$M_2' : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2' \tag{C.794}$$

Similarly, by applying Lemma 81 to the first and second hypothesis, we get:

$$\vdash_{ctx}^{M} P; \Gamma_C; \bullet \rightsquigarrow \Sigma_1; \Gamma_C; \bullet$$

$$\vdash_{ctx}^{M} P; \Gamma_C; \bullet \rightsquigarrow \Sigma_1; \Gamma_C; \bullet$$

By applying Theorem 51 to Equations C.791 and C.793, together with this result, hypotheses 2, 3 and 5, and rule sTy-bool, we know that:

$$\Sigma_1 : M_1' \simeq_{log} \Sigma_2 : M_2' : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow \mathit{Bool}) \tag{C.795}$$

We take $M_1 = M_1'$ and $M_2 = M_2'$. Consequently, since $F_{\mathbf{D}}$ context elaboration is deterministic (Theorem 68), we get that $M_1 = M_1'$ and $M_2 = M_2'$. Goal C.789 follows from Equations C.790, C.792, C.794 and C.795.

□

C.7.3 Partial Coherence Theorems

Theorem 55 (Coherence - Dictionaries - Constraint Entailment)**.**
If $P;\Gamma_C;\Gamma \vDash^M [C] \rightsquigarrow d_1$ *and* $P;\Gamma_C;\Gamma \vDash^M [C] \rightsquigarrow d_2$
and $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$
then $\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : C$
where $\Gamma_C;\Gamma \vdash^M_C C \rightsquigarrow C$.

Proof. By induction on the first constraint entailment derivation. This theorem is mutually proven with Theorems 53, 58, 56 and 57 (see Figure C.4). Note that at the dependency between Theorem 53 and 58, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Theorems 55, 56 and 57 perform induction on the given derivation, which we assume to be finite. Consequently, the induction remains well-founded.

$\boxed{\textbf{rule } \text{sEntail-arrow} \textbf{ : } P;\Gamma_C;\Gamma \vDash^M [C_1 \Rightarrow C_2] \rightsquigarrow \lambda\delta : C_1.d'_1}$

From the rule premise:

$$P;\Gamma_C;\Gamma,\delta : C_1 \vDash^M [C_2] \rightsquigarrow d'_1 \tag{C.796}$$

$$\tag{C.797}$$

$$\Gamma_C;\Gamma \vdash^M_C C_1 \rightsquigarrow C_1 \tag{C.798}$$

By case analysis on the second derivation (rule sEntail-arrow), we get:

$$P;\Gamma_C;\Gamma \vDash^M [C_1 \Rightarrow C_2] \rightsquigarrow \lambda\delta : C_1.d'_1 \tag{C.799}$$

$$P;\Gamma_C;\Gamma,\delta : C_1 \vDash^M [C_2] \rightsquigarrow d'_1 \tag{C.800}$$

$$\tag{C.801}$$

$$\Gamma_C;\Gamma \vdash^M_C C_1 \rightsquigarrow C_1 \tag{C.802}$$

Note that as we follow the convention that the choice of variable names is unimportant, we simplify the discussion above by picking the same name δ and δ for the freshly generated dictionary variable name.

We continue by applying environment weakening in the 3rd and 4th hypothesis (Lemma 66, in combination with rule sCtx-pgmInst and rule sCtx-tyEnvD):

$$\vdash^M_{ctx} P;\Gamma_C;\Gamma,\delta : C_1 \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma,\delta : C_1 \tag{C.803}$$

$$\vdash^M_{ctx} P;\Gamma_C;\Gamma,\delta : C_1 \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma,\delta : C_1 \tag{C.804}$$

Applying the induction hypothesis to Equations C.797, C.801, C.803 and C.804 results in

$$\Gamma_C; \Gamma, \delta : C_1 \vdash \Sigma_1 : d_1' \simeq_{log} \Sigma_2 : d_2' : C_2$$

where $\Gamma_C; \Gamma, \delta : C_1 \vdash^M_C C_2 \rightsquigarrow C_2$.

The goal follows directly by Compatibility Lemma 128.

$\boxed{\textbf{rule } \textsc{sEntail-forall} \textbf{ :} \; P; \Gamma_C; \Gamma \vDash^M [\forall a.C'] \rightsquigarrow \Lambda a.d_1'}$

From the rule premise:

$$P; \Gamma_C; \Gamma, a \vDash^M [C'] \rightsquigarrow d_1' \tag{C.805}$$

By case analysis on the second derivation (rule sEntail-forall), we get:

$$P; \Gamma_C; \Gamma \vDash^M [\forall a.C'] \rightsquigarrow \Lambda a.d_2' \tag{C.806}$$

$$P; \Gamma_C; \Gamma, a \vDash^M [C'] \rightsquigarrow d_2' \tag{C.807}$$

Similarly to the previous case, we apply environment weakening in the 3rd and 4th hypothesis (Lemma 66, in combination with rule sCtx-pgmInst and rule sCtx-tyEnvTy) to get

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma, a \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, a \tag{C.808}$$

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma, a \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma, a \tag{C.809}$$

Applying the induction hypothesis to Equations C.805, C.807, C.808 and C.809 gives us

$$\Gamma_C; \Gamma, a \vdash \Sigma_1 : d_1' \simeq_{log} \Sigma_2 : d_2' : C'$$

where $\Gamma_C; \Gamma, a \vdash^M_C C' \rightsquigarrow C'$.

The goal follows by Compatibility Lemma 129.

$\boxed{\textbf{rule } \textsc{sEntail-inst} \textbf{ :} \; P; \Gamma_C; \Gamma \vDash^M [Q] \rightsquigarrow d_1}$

From the premise:

$$P = P_1, (D : C).m \mapsto \Gamma' : e, P_2 \tag{C.810}$$

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash D : C] \vDash^M Q \rightsquigarrow \bullet \vdash d_1 \tag{C.811}$$

By Lemma 79 we get

$$\Sigma_1; \Gamma_C; \Gamma \vdash_d D : C \tag{C.812}$$

By case analysis, the final step of the second derivation can either be rule sEntail-inst or rule sEntail-local:

- rule sEntail-inst: The rule premise thus gives us

  $$P;\Gamma_C;\Gamma \models^M [Q]\leadsto d_2 \tag{C.813}$$

  $$P = P_1', (D' : C').m' \mapsto \Gamma'' : e', P_2' \tag{C.814}$$

  $$P;\Gamma_C;\Gamma;[\bullet;\bullet \vdash D' : C'] \models^M Q\leadsto \bullet \vdash d_2 \tag{C.815}$$

  We know from Lemma 79 that

  $$\Sigma_2;\Gamma_C;\Gamma \vdash_d D' : C' \tag{C.816}$$

  where $\Gamma_C;\Gamma \vdash_C^M C' \leadsto C'$. The goal follow directly by applying Theorem 56 to Equations C.811, C.812,C.815 and C.816 in combination with the hypotheses.

- rule sEntail-local: The rule premise thus gives us

  $$P;\Gamma_C;\Gamma \models^M [Q]\leadsto d_2 \tag{C.817}$$

  $$(\delta : C') \in \Gamma \tag{C.818}$$

  $$P;\Gamma_C;\Gamma;[\bullet;\bullet \vdash \delta : C'] \models^M Q\leadsto \bullet \vdash d_2 \tag{C.819}$$

  From Lemma 77, we know that

  $$(\delta : C') \in \Gamma$$

  $$\Gamma_C;\Gamma \vdash_C^M C' \leadsto C'$$

  It then follows directly from rule D-var that

  $$\Sigma_2;\Gamma_C;\Gamma \vdash_d \delta : C' \tag{C.820}$$

  The goal follow directly by applying Theorem 56 to Equations C.811, C.812, C.819 and C.820 in combination with the hypotheses.

$\boxed{\textbf{rule}\ \textsc{sEntail-local}\ \mathbf{:}\ P;\Gamma_C;\Gamma \models^M [Q]\leadsto d_1}$

From the premise:

$$(\delta : C) \in \Gamma \tag{C.821}$$

$$P;\Gamma_C;\Gamma;[\bullet;\bullet \vdash \delta : C] \models^M Q\leadsto \bullet \vdash d_1 \tag{C.822}$$

It follows, similarly to the previous case, that

$$\Sigma_1;\Gamma_C;\Gamma \vdash_d \delta : C \tag{C.823}$$

By case analysis, the final step of the second derivation can either be rule sEntail-inst or rule sEntail-local:

- rule sENTAIL-INST: This case is entirely identical to the previous case.
- rule sENTAIL-LOCAL: The rule premise thus gives us

$$P; \Gamma_C; \Gamma \vDash^M [Q] \rightsquigarrow d_2 \tag{C.824}$$

$$(\delta' : C') \in \Gamma \tag{C.825}$$

$$P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash \delta' : C'] \vDash^M Q \rightsquigarrow \bullet \vdash d_2 \tag{C.826}$$

And again similarly, we get

$$\Sigma_2; \Gamma_C; \Gamma \vdash_d \delta' : C' \tag{C.827}$$

The goal follows by applying Theorem 56 to Equations C.822, C.823, C.826 and C.827 in combination with the hypotheses.

□

Theorem 56 (Coherence - Dictionaries - Constraint Matching Left)**.**
If $P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3 \vdash d_3 : C_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash d_1$
and $P; \Gamma_C; \Gamma; [\bullet; \bullet \vdash d_4 : C_4] \vDash^M Q \rightsquigarrow \bullet \vdash d_2$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma, \overline{a}_3 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, \overline{a}_3$ *and* $\Sigma_1; \Gamma_C; \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3 \vdash_d d_3 : C_3$
where $\Gamma_C; \Gamma, \overline{a}_3 \vdash^M_C C_3 \rightsquigarrow C_3$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_d d_4 : C_4$
where $\Gamma_C; \Gamma \vdash^M_C C_4 \rightsquigarrow C_4$
then $\forall R_1 \in \mathcal{F}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]^{\Gamma_C}$, $R_2 \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$,
$\gamma_1 \in \mathcal{H}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_1}$, $\gamma_2 \in \mathcal{H}[\![\Gamma]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_2}$:
$(\Sigma_1 : \gamma_1(R_1(d_1)), \Sigma_2 : \gamma_2(R_2(d_2))) \in \mathcal{E}[\![R_1(Q)]\!]^{\Gamma_C}$ *where* $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$.

Proof. By induction on the first constraint matching derivation. This theorem is mutually proven with Theorems 53, 58, 55 and 57 (see Figure C.4). Note that at the dependency between Theorem 53 and 58, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Theorems 55, 56 and 57 perform induction on the given derivation, which we assume to be finite. Consequently, the induction remains well-founded.

rule sMATCH-ARROW **:** $P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3 \vdash d_3 : C'_3 \Rightarrow C''_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash [d'_3/\delta_3]d_1$

From the rule premise:

$$P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3, \delta_3 : C'_3 \vdash d_3\, \delta_3 : C''_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash d_1 \tag{C.828}$$

$$P; \Gamma_C; \Gamma \vDash^M [[\overline{\tau}_3/\overline{a}_3]C'_3] \rightsquigarrow d'_3 \tag{C.829}$$

By case analysis on the hypothesis (rule sC-arrow), we know that

$$\Gamma_C; \Gamma, \overline{a}_3 \vdash^M_C C'_3 \rightsquigarrow C'_3$$

Using rule D-var, it is easy to see that $\Sigma_1; \Gamma_C; \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3, \delta_3 : C'_3 \vdash_d \delta_3 : C'_3$. Together with rule D-dapp and the hypotheses, this result leads to

$$\Sigma_1; \Gamma_C; \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3, \delta_3 : C'_3 \vdash_d d_3\, \delta_3 : C''_3$$

Applying the induction hypothesis then gives us

$$(\Sigma_1 : \gamma'_1(R_1(d_1)), \Sigma_2 : \gamma_2(R_2(d_2))) \in \mathcal{E}[\![R_1(Q)]\!]^{\Gamma_C} \tag{C.830}$$

with $\gamma'_1 \in \mathcal{H}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3, \delta_3 : C'_3]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_{R_1}$ and where $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$. From the definition of $\mathcal{H}$ we know that $\gamma'_1 = \gamma_1, \delta_3 \mapsto (d_5, d_6)$ for any $(\Sigma_1 : d_5, \Sigma_2 : d_6) \in \mathcal{E}[\![R_1(C'_3)]\!]^{\Gamma_C}$.

We now apply Theorem 55 to Equation C.829 twice, as well as to the 3rd and 6th hypotheses. This gives us

$$(\Sigma_1 : d'_3, \Sigma_2 : d'_3) \in \mathcal{E}[\![R_1([\overline{\sigma}_3/\overline{a}_3]C'_3)]\!]^{\Gamma_C}$$

The goal follows directly by chosing $d_5 = d_6 = d'_3$.

rule sMatch-forall **:** $P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3 \vdash d_3 : \forall a. C'_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash d_1$

From the rule premise:

$$P; \Gamma_C; \Gamma; [\overline{a}_3, a; \overline{\delta}_3 : \overline{C}_3 \vdash d_3\, a : C'_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3, \tau_3 \vdash d_1$$

Combining rule D-tyapp, rule iTy-var and the hypotheses, gives us

$$\Sigma_1; \Gamma_C; \Gamma, \overline{a}_3, a \vdash_d d_3\, a : C'_3$$

From the induction hypothesis we get that $\forall R'_1 \in \mathcal{F}[\![\Gamma, \overline{a}_3, a, \overline{\delta}_3 : \overline{C}_3]\!]^{\Gamma_C}$, $R_2 \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$, $\gamma_1 \in \mathcal{H}[\![\Gamma, \overline{a}_3, a, \overline{\delta}_3 : \overline{C}_3]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_{R'_1}$, $\gamma_2 \in \mathcal{H}[\![\Gamma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_{R_2}$:

$$(\Sigma_1 : \gamma_1(R'_1(d_1)), \Sigma_2 : \gamma_2(R_2(d_2))) \in \mathcal{E}[\![R'_1(Q)]\!]^{\Gamma_C} \tag{C.831}$$

However, we know from Typing Preservation Theorem 27 that d_1 does not depend on a. Because of this, the goal follows directly from Equation C.831.

rule sMatch-classconstr **:** $P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3 \vdash d_3 : Q_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash [\overline{\sigma}/\overline{a}]d_3$

The goal follows directly from Theorem 57. □

Theorem 57 (Coherence - Dictionaries - Constraint Matching Right)**.**
If $P; \Gamma_C; \Gamma; [\overline{a}_3; \overline{\delta}_3 : \overline{C}_3 \vdash d_3 : Q_3] \vDash^M Q \rightsquigarrow \overline{\tau}_3 \vdash d_1$
and $P; \Gamma_C; \Gamma; [\overline{a}_4; \overline{\delta}_4 : \overline{C}_4 \vdash d_4 : C_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4 \vdash d_2$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma, \overline{a}_3 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, \overline{a}_3$ *and* $\Sigma_1; \Gamma_C; \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3 \vdash_d d_3 : Q_3$
where $\Gamma_C; \Gamma, \overline{a}_3 \vdash^M_C C_3 \rightsquigarrow C_3$
and $\vdash^M_{ctx} P; \Gamma_C; \Gamma, \overline{a}_4 \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma, \overline{a}_4$ *and* $\Sigma_2; \Gamma_C; \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4 \vdash_d d_4 : C_4$
where $\Gamma_C; \Gamma, \overline{a}_4 \vdash^M_C C_4 \rightsquigarrow C_4$
then $\forall R_1 \in \mathcal{F}\llbracket \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3 \rrbracket^{\Gamma_C}$, $R_2 \in \mathcal{F}\llbracket \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4 \rrbracket^{\Gamma_C}$,
$\gamma_1 \in \mathcal{H}\llbracket \Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3 \rrbracket^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_1}$, $\gamma_2 \in \mathcal{H}\llbracket \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4 \rrbracket^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_2}$:
$(\Sigma_1 : \gamma_1(R_1(d_1)), \Sigma_2 : \gamma_2(R_2(d_2))) \in \mathcal{E}\llbracket R_1(Q) \rrbracket^{\Gamma_C}$ *where* $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$.

Proof. By induction on the second constraint matching derivation. This theorem is mutually proven with Theorems 53, 58, 55 and 56 (see Figure C.4). Note that at the dependency between Theorem 53 and 58, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Theorems 55, 56 and 57 perform induction on the given derivation, which we assume to be finite. Consequently, the induction remains well-founded.

$$\boxed{\textbf{rule } \textsc{sMatch-arrow} \textbf{ :}\ P; \Gamma_C; \Gamma; [\overline{a}_4; \overline{\delta}_4 : \overline{C}_4 \vdash d_4 : C'_4 \Rightarrow C''_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4 \vdash [d'_4/\delta_4]d_2}$$

From the rule premise:

$$P; \Gamma_C; \Gamma; [\overline{a}_4; \overline{\delta}_4 : \overline{C}_4, \delta_4 : C'_4 \vdash d_4\, \delta_4 : C''_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4 \vdash d_2 \tag{C.832}$$

$$P; \Gamma_C; \Gamma \vDash^M [[\overline{\tau}_4/\overline{a}_4]C'_4] \rightsquigarrow d'_4 \tag{C.833}$$

By case analysis on the hypothesis (rule sC-arrow), we know that

$$\Gamma_C; \Gamma, \overline{a}_4 \vdash^M_C C'_4 \rightsquigarrow C'_4$$

Using rule D-var, it is easy to see that $\Sigma_2; \Gamma_C; \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4, \delta_4 : C'_4 \vdash_d \delta_4 : C'_4$. Together with rule D-dapp and the hypotheses, this result leads to

$$\Sigma_2; \Gamma_C; \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4, \delta_4 : C'_4 \vdash_d d_4\, \delta_4 : C''_4$$

Applying the induction hypothesis then gives us

$$(\Sigma_1 : \gamma_1(R_1(d_1)), \Sigma_2 : \gamma'_2(R_2(d_2))) \in \mathcal{E}\llbracket R_1(Q) \rrbracket^{\Gamma_C} \tag{C.834}$$

with $\gamma'_2 \in \mathcal{H}\llbracket \Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4, \delta_4 : C'_4 \rrbracket^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_1}$ and where $\Gamma_C; \Gamma \vdash^M_Q Q \rightsquigarrow Q$. From the definition of $\mathcal{H}$ we know that $\gamma'_2 = \gamma_2, \delta_4 \mapsto (d_5, d_6)$ for any $(\Sigma_1 : d_5, \Sigma_2 : d_6) \in \mathcal{E}\llbracket R_1(C'_4) \rrbracket^{\Gamma_C}$.

We now apply Theorem 55 to Equation C.833 twice, as well as to the 3rd and 6th hypotheses. This gives us

$$(\Sigma_1 : d'_4, \Sigma_2 : d'_4) \in \mathcal{E}[\![R_1([\overline{\sigma}_4/\overline{a}_4]C'_4)]\!]^{\Gamma_C}$$

The goal follows directly by chosing $d_5 = d_6 = d'_4$.

$$\boxed{\textbf{rule}\ \textsc{sMatch-forall} : P; \Gamma_C; \Gamma; [\overline{a}_4; \overline{\delta}_4 : \overline{C}_4 \vdash d_4 : \forall a. C'_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4 \vdash d_2}$$

From the rule premise:

$$P; \Gamma_C; \Gamma; [\overline{a}_4, a; \overline{\delta}_4 : \overline{C}_4 \vdash d_4\, a : C'_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4, \tau_4 \vdash d_2$$

Combining rule D-tyapp, rule iTy-var and the hypotheses, gives us

$$\Sigma_2; \Gamma_C; \Gamma, \overline{a}_4, a \vdash_d d_4\, a : C'_4$$

From the induction hypothesis we get that $\forall R_1 \in \mathcal{F}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]^{\Gamma_C}$, $R'_2 \in \mathcal{F}[\![\Gamma, \overline{a}_4, a, \overline{\delta}_4 : \overline{C}_4]\!]^{\Gamma_C}$, $\gamma_1 \in \mathcal{H}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_{R_1}$, $\gamma_2 \in \mathcal{H}[\![\Gamma, \overline{a}_4, a, \overline{\delta}_4 : \overline{C}_4]\!]^{\Sigma_1, \Sigma_2, \Gamma_C}_{R'_2}$:

$$(\Sigma_1 : \gamma_1(R_1(d_1)), \Sigma_2 : \gamma_2(R'_2(d_2))) \in \mathcal{E}[\![R_1(Q)]\!]^{\Gamma_C} \tag{C.835}$$

However, we know from Typing Preservation Theorem 27 that d_1 does not depend on a. Because of this, the goal follows directly from Equation C.835.

$$\boxed{\textbf{rule}\ \textsc{sMatch-classconstr} : P; \Gamma_C; \Gamma; [\overline{a}_4; \overline{\delta}_4 : \overline{C}_4 \vdash d_4 : Q_4] \vDash^M Q \rightsquigarrow \overline{\tau}_4 \vdash [\overline{\sigma}_4/\overline{a}_4]d_4}$$

From the rule premise:

$$Q = [\overline{\tau}_4/\overline{a}_4]Q_4$$

$$\overline{\Gamma_C; \Gamma \vdash^M_{ty} \tau_{i\,4} \rightsquigarrow \sigma_{i\,4}}^{\,i}$$

Case analysis on the first derivation (rule sMatch-classconstr) tells us

$$d_1 = [\overline{\sigma}_3/\overline{a}_3]d_3$$

$$Q = [\overline{\tau}_3/\overline{a}_3]Q_3$$

$$\overline{\Gamma_C; \Gamma \vdash^M_{ty} \tau_{i\,3} \rightsquigarrow \sigma_{i\,3}}^{\,i}$$

Applying Substitution Lemma 93 to the hypotheses gives us

$$\Sigma_1; \Gamma_C; \Gamma, \overline{\delta}_3 : [\overline{\sigma}_3/\overline{a}_3]\overline{C}_3 \vdash_d [\overline{\sigma}_3/\overline{a}_3]d_3 : [\overline{\sigma}_3/\overline{a}_3]Q_3 \tag{C.836}$$

$$\Sigma_2; \Gamma_C; \Gamma, \overline{\delta}_4 : [\overline{\sigma}_4/\overline{a}_4]\overline{C}_4 \vdash_d [\overline{\sigma}_4/\overline{a}_4]d_4 : [\overline{\sigma}_4/\overline{a}_4]Q_4 \tag{C.837}$$

The goal to be proven is

$$\forall R_1 \in \mathcal{F}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]^{\Gamma_C}, R_2 \in \mathcal{F}[\![\Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4]\!]^{\Gamma_C}, \tag{C.838}$$

$$\gamma_1 \in \mathcal{H}[\![\Gamma, \overline{a}_3, \overline{\delta}_3 : \overline{C}_3]\!]_{R_1}^{\Sigma_1,\Sigma_2,\Gamma_C}, \gamma_2 \in \mathcal{H}[\![\Gamma, \overline{a}_4, \overline{\delta}_4 : \overline{C}_4]\!]_{R_2}^{\Sigma_1,\Sigma_2,\Gamma_C} : \tag{C.839}$$

$$(\Sigma_1 : \gamma_1(R_1([\overline{\sigma}_3/\overline{a}_3]d_3)), \Sigma_2 : [\overline{\sigma}_4/\overline{a}_4]\gamma_2(R_2(d_4))) \in \mathcal{E}[\![R_1(Q)]\!]^{\Gamma_C} \tag{C.840}$$

We thus proceed by repeatedly applying Substitution Lemmas 91 and 93 to Equations C.836 and C.837:

$$\Sigma_1; \Gamma_C; \bullet \vdash_d \gamma_1(R([\overline{\sigma}_3/\overline{a}_3]d_3)) : R([\overline{\sigma}_3/\overline{a}_3]Q_3)$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_d \gamma_2(R([\overline{\sigma}_4/\overline{a}_4]d_4)) : R([\overline{\sigma}_4/\overline{a}_4]Q_4)$$

Furthermore, by applying Theorem 53 to the hypotheses, we get

$$\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$$

Applying Theorem 52 to these results gives us

$$(\Sigma_1 : \gamma_1(R([\overline{\sigma}_3/\overline{a}_3]d_3)), \Sigma_2 : [\overline{\sigma}_4/\overline{a}_4]\gamma_2(R(d_4))) \in \mathcal{E}[\![R(Q)]\!]^{\Gamma_C}$$

where $Q = [\overline{\sigma}_3/\overline{a}_3]Q_3$. Goal C.840 thus follows directly. □

Theorem 58 (Coherence - Expressions - Part A)**.**

- *If* $P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Rightarrow \tau \rightsquigarrow e_2$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$ *then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *where* $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma$.
- *If* $P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e_2$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma$ *and* $\vdash_{ctx}^{M} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma$ *then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *where* $\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma$.

Proof. By mutual induction on the first typing derivation. This theorem is mutually proven with Theorems 53, 55, 56 and 57 (see Figure C.4). Note that at the dependency between Theorem 53 and 58, the size of P is strictly decreasing, whereas P remains constant at every other dependency. Theorems 55, 56 and 57 perform induction on the given derivation, which we assume to be finite. Consequently, the induction remains well-founded.

By applying Lemma 74 to the 1st hypothesis, we get that:

$$\Gamma_C; \Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \tag{C.841}$$

Part 1

$\boxed{\textbf{rule}\ \textsc{sTm-inf-true}}$ $\quad P; \Gamma_C; \Gamma \vdash^{M}_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}$
Through case analysis, it is straightforward to see that the final step in the second derivation is rule sTm-inf-true as well. This means that $e_1 = e_2 = \mathit{True}$. From Theorem 53, we know that $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$. The goal follows from reflexivity Theorem 50.
$\boxed{\textbf{rule}\ \textsc{sTm-inf-false}}$ $\quad P; \Gamma_C; \Gamma \vdash^{M}_{tm} \mathit{True} \Rightarrow \mathit{Bool} \rightsquigarrow \mathit{True}$
The proof is identical to the rule sTm-inf-true case.
$\boxed{\textbf{rule}\ \textsc{sTm-inf-let}}$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1\, \textbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e_3$$

where $e_3 = \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1\, \textbf{in}\ \ e_2$. Through case analysis, we know that the final step in the second derivation has to be rule sTm-inf-let as well. This means that:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1\, \textbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e_3$$

$$\mathit{where}\ e_3 = \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e_1\, \textbf{in}\ \ e_2$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_i \Rightarrow \tau_1 = e_1\, \textbf{in}\ \ e_2 \Rightarrow \tau_2 \rightsquigarrow e_4$$

$$\mathit{where}\ e_4 = \textbf{let}\ \ x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \sigma = \Lambda \overline{a}_j . \lambda \overline{\delta}_k : \overline{C}_k . e'_1\, \textbf{in}\ \ e'_2$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_3 \simeq_{log} \Sigma_2 : e_4 : \sigma_2\ \mathit{where}\ \Gamma_C; \Gamma \vdash^{M}_{ty} \sigma_2 \rightsquigarrow \sigma_2 \tag{C.842}$$

The rule premise tells us that:

$$P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash^{M}_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \tag{C.843}$$

$$P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \vdash^{M}_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \tag{C.844}$$

$$P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash^{M}_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e'_1 \tag{C.845}$$

$$P; \Gamma_C; \Gamma, x : \forall \overline{a}_j . \overline{C}_k \Rightarrow \tau_1 \vdash^{M}_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e'_2 \tag{C.846}$$

$$\mathit{where}\ \textbf{closure}(\Gamma_C; \overline{C}_i) = \overline{C}_k$$

From Lemma 73, together with Equations C.843, C.844, C.845 and C.846, and through repeated case analysis on the results to discover the contents of the elaborated environments, we get that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \tau_1 \rightsquigarrow \Sigma_2; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma$$

By applying the induction hypothesis to Equations C.844 and C.846, we get:

$$\Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma \vdash \Sigma_1 : e_2 \simeq_{log} \Sigma_2 : e_2' : \sigma_2 \tag{C.847}$$

Furthermore, applying Part 2 of this lemma to Equations C.843 and C.845 results in:

$$\Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{C}_k \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_1' : \sigma \tag{C.848}$$

Applying compatibility Lemma 126, together with Equation C.847, reduces Goal C.842 to:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \Lambda \overline{a}_j.\lambda \overline{\delta}_k : \overline{C}_k.e_1 \simeq_{log} \Sigma_2 : \Lambda \overline{a}_j.\lambda \overline{\delta}_k : \overline{C}_k.e_1' : \forall \overline{a}_j.\overline{C}_k \Rightarrow \sigma \tag{C.849}$$

Combining compatibility Lemma 122 with Equation C.848 gives us:

$$\Gamma_C; \Gamma, \overline{a}_j \vdash \Sigma_1 : \lambda \overline{\delta}_k : \overline{C}_k.e_1 \simeq_{log} \Sigma_2 : \lambda \overline{\delta}_k : \overline{C}_k.e_1' : \overline{C}_k \Rightarrow \sigma \tag{C.850}$$

Goal C.849 follows directly by applying Equation C.850 to compatibility Lemma 124.

$\boxed{\textbf{rule}\ \textsc{sTm-inf-ArrE}}$ $\quad P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2$

Through case analysis, we see that the final step in the second derivation can only be rule sTm-inf-ArrE. This means that:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1\, e_2$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_1\, e_2 \Rightarrow \tau_2 \rightsquigarrow e_1'\, e_2'$$

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1\, e_2 \simeq_{log} \Sigma_2 : e_1'\, e_2' : \sigma_2 \ \textit{where}\ \Gamma_C; \Gamma \vdash^{M}_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.851}$$

The rule premise tells us that:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1 \tag{C.852}$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2 \tag{C.853}$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_1 \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow e_1' \tag{C.854}$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e_2 \Leftarrow \tau_1 \rightsquigarrow e_2' \tag{C.855}$$

Applying the induction hypothesis on Equations C.852 and C.854 results in:

$$\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_1' : \sigma_1 \to \sigma_2 \textit{ where } \Gamma_C;\Gamma \vdash_{ty}^{M} \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2 \tag{C.856}$$

By applying Part 2 of this lemma to Equations C.853 and C.855 we get:

$$\Gamma_C;\Gamma \vdash \Sigma_1 : e_2 \simeq_{log} \Sigma_2 : e_2' : \sigma_1 \textit{ where } \Gamma_C;\Gamma \vdash_{ty}^{M} \tau_1 \rightsquigarrow \sigma_1 \tag{C.857}$$

Goal C.851 follows by applying Equations C.856 and C.857 to compatibility Lemma 121.

$\boxed{\textbf{rule } \text{sTm-inf-Ann}}$ $P;\Gamma_C;\Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e$

Through case analysis we know that the final step in the second derivation is rule sTm-inf-Ann. This means that:

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e$$

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} e :: \tau \Rightarrow \tau \rightsquigarrow e'$$

The goal to be proven is the following:

$$\Gamma_C;\Gamma \vdash \Sigma_1 : e \simeq_{log} \Sigma_2 : e' : \sigma \textit{ where } \Gamma_C;\Gamma \vdash_{ty}^{M} \tau \rightsquigarrow \sigma \tag{C.858}$$

From the rule premise we know:

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e \tag{C.859}$$

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} e \Leftarrow \tau \rightsquigarrow e' \tag{C.860}$$

The goal follows directly from Part 2 of this lemma, applied to Equation C.859 and C.860.

Part 2

$\boxed{\textbf{rule } \text{sTm-check-var}}$ $P;\Gamma_C;\Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j/\overline{a}_j]\tau \rightsquigarrow x\,\overline{\sigma}_j\,\overline{d}_i$

Through case analysis, we see that the final step in the second typing derivation can either be rule sTm-check-var or rule sTm-check-Inf. However, noting that no matching inference rules exist, we conclude that the final derivation step has to be rule sTm-check-var. This means that:

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j/\overline{a}_j]\tau \rightsquigarrow x\,\overline{\sigma}_j\,\overline{d}_i$$

$$P;\Gamma_C;\Gamma \vdash_{tm}^{M} x \Leftarrow [\overline{\tau}_j/\overline{a}_j]\tau \rightsquigarrow x\,\overline{\sigma}_j'\,\overline{d}_i'$$

The goal to be proven is the following:

$$\Gamma_C;\Gamma \vdash \Sigma_1 : x\,\overline{\sigma}_j\,\overline{d}_i \simeq_{log} \Sigma_2 : x\,\overline{\sigma}_j'\,\overline{d}_i' : \sigma' \textit{ where } \Gamma_C;\Gamma \vdash_{ty}^{M} [\overline{\tau}_j/\overline{a}_j]\tau \rightsquigarrow \sigma' \tag{C.861}$$

By inversion on Equation C.841, we know that $\sigma' = [\overline{\sigma}_j/\overline{a}_j]\sigma$.

The rule premise tells us that:

$$(x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \tau) \in \Gamma \tag{C.862}$$

$$\overline{P;\Gamma_C;\Gamma \vDash^{M} [[\overline{\tau}_j/\overline{a}_j]C_i] \rightsquigarrow d_i}^{i} \tag{C.863}$$

$$\overline{\Gamma_C;\Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma_j}^{j} \tag{C.864}$$

$$\overline{P;\Gamma_C;\Gamma \vDash^{M} [[\overline{\tau}_j/\overline{a}_j]C_i] \rightsquigarrow d'_i}^{i} \tag{C.865}$$

$$\overline{\Gamma_C;\Gamma \vdash^{M}_{ty} \tau_j \rightsquigarrow \sigma'_j}^{j} \tag{C.866}$$

Since type elaboration is completely deterministic (Lemma 69), we know that $\overline{\sigma}_j = \overline{\sigma}'_j$.

From Lemma 75, combined with the 3rd hypothesis and Equation C.862, we know that:

$$(x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma) \in \Gamma \tag{C.867}$$

By applying Theorem 28 to the 3rd and 4th hypothesis, we get:

$$\vdash_{ctx} \Sigma_1;\Gamma_C;\Gamma \tag{C.868}$$

$$\vdash_{ctx} \Sigma_2;\Gamma_C;\Gamma \tag{C.869}$$

Applying Equations C.867, C.868 and C.869 to rule ITM-VAR, results in:

$$\Sigma_1;\Gamma_C;\Gamma \vdash_{tm} x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma \tag{C.870}$$

$$\Sigma_2;\Gamma_C;\Gamma \vdash_{tm} x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma \tag{C.871}$$

From Theorem 53, we know that:

$$\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2 \tag{C.872}$$

From reflexivity Theorem 50, applied on Equations C.870, C.871 and C.872, we know that:

$$\Gamma_C;\Gamma \vdash \Sigma_1 : x \simeq_{log} \Sigma_2 : x : \forall \overline{a}_j.\overline{C}_i \Rightarrow \sigma \tag{C.873}$$

From repeated case analysis on the 3rd hypothesis (rule SCTX-PGMINST), we get:

$$\vdash^{M}_{ctx} \bullet;\Gamma_C;\Gamma \rightsquigarrow \bullet;\Gamma_C;\Gamma \tag{C.874}$$

By applying Theorem 25 to Equations C.864 and C.874, we know that:

$$\overline{\Gamma_C; \Gamma \vdash_{ty} \sigma_j}^{\,j} \tag{C.875}$$

Applying compatibility Lemma 125 j times to Equations C.873 and C.875 results in:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : x\,\overline{\sigma}_j \simeq_{log} \Sigma_2 : x\,\overline{\sigma}_j : [\overline{\sigma}_j/\overline{a}_j]\overline{C}_i \Rightarrow [\overline{\sigma}_j/\overline{a}_j]\sigma \tag{C.876}$$

Applying Theorem 55 to Equations C.863 and C.865 gives us:

$$\overline{\Gamma_C; \Gamma \vdash \Sigma_1 : d_i \simeq_{log} \Sigma_2 : d'_i : C'_i}^{\,i} \tag{C.877}$$

$$\textit{where}\ \overline{\Gamma_C; \Gamma \vdash^{M}_{C} [\overline{\tau}_j/\overline{a}_j]C_i \rightsquigarrow C'_i}^{\,i} \tag{C.878}$$

By inversion on Equation C.878, we know that $\overline{C'_i = [\overline{\sigma}_j/\overline{a}_j]C_i}^{\,i}$.

Goal C.861 follows by repeatedly applying compatibility Lemma 123 to Equation C.876, combined with Equation C.877.

$\boxed{\textbf{rule}\ \textsc{sTm-check-Meth}}$ $\quad P; \Gamma_C; \Gamma \vdash^{M}_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\,\overline{\sigma}_j\,\overline{d}_i$

The proof is similar to the rule sTm-check-Var case.

$\boxed{\textbf{rule}\ \textsc{sTm-check-ArrI}}$ $\quad P; \Gamma_C; \Gamma \vdash^{M}_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma_1.e$

Through case analysis, it is straightforward to note that the final step in the second typing derivation can be either rule sTm-check-ArrI or rule sTm-check-Inf. In the latter case however, no matching inference rules exist. The rule sTm-check-ArrI case is the only remaining possibility. This means that:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma_1.e$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} \lambda x.e \Leftarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \sigma'_1.e'$$

From the 3rd rule premise we know that:

$$\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightsquigarrow \sigma_1$$

$$\Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightsquigarrow \sigma'_1$$

Since type elaboration is entirely deterministic (Lemma 69), it is straightforward to note that $\sigma_1 = \sigma'_1$.

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : \lambda x : \sigma_1.e \simeq_{log} \Sigma_2 : \lambda x : \sigma_1.e' : \sigma'\ \textit{where}\ \Gamma_C; \Gamma \vdash^{M}_{ty} \tau_1 \rightarrow \tau_2 \rightsquigarrow \sigma' \tag{C.879}$$

By inversion on Equation C.841, we know that $\sigma' = \sigma_1 \to \sigma_2$.

From the rule premise we know:

$$P; \Gamma_C; \Gamma, x : \tau_1 \vdash^M_{tm} e \Leftarrow \tau_2 \rightsquigarrow e \tag{C.880}$$

$$P; \Gamma_C; \Gamma, x : \tau_1 \vdash^M_{tm} e \Leftarrow \tau_2 \rightsquigarrow e' \tag{C.881}$$

By applying Equation C.880 to Lemma 73, and through repeated case analysis on the result to discover the contents of the elaborated environments, we know that:

$$\vdash^M_{ctx} P; \Gamma_C; \Gamma, x : \tau_1 \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma, x : \sigma_1$$

Applying the induction hypothesis on Equations C.880 and C.881 results in:

$$\Gamma_C; \Gamma, x : \sigma_1 \vdash \Sigma_1 : e \simeq_{log} \Sigma_2 : e' : \sigma_2 \textit{ where } \Gamma_C; \Gamma, x : \tau_1 \vdash^M_{ty} \tau_2 \rightsquigarrow \sigma_2 \tag{C.882}$$

Goal C.879 follows directly from compatibility Lemma 120.

$\boxed{\textbf{rule } \textsc{sTm-check-Inf}}$ $\quad P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e$

Through case analysis, we note that the final step in the second typing derivation can either be rule sTm-check-var, rule sTm-check-meth, rule sTm-check-ArrI or rule sTm-check-Inf. In the first 3 cases, the proof is symmetrical to the corresponding proof cases described above. We proceed with the last case:

The goal to be proven is the following:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e \simeq_{log} \Sigma_2 : e' : \sigma \textit{ where } \Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma \tag{C.883}$$

where we know that:

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e \tag{C.884}$$

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e' \tag{C.885}$$

The rule premise tells us that:

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e \tag{C.886}$$

$$P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e' \tag{C.887}$$

The goal follows directly by applying Part 1 of this lemma to Equations C.886 and C.887. Part 1 can be applied on e, even though the term size did not decrease, because inference is defined to be smaller than type checking in our proof by induction.

□

Theorem 59 (Coherence - Expressions - Part B). *If* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *then* $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

Proof. By unfolding the definition of contextual equivalence, the goal becomes:

$$\forall M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow \mathit{Bool})$$

$$\forall M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool})$$

$$\mathit{if}\ \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow \mathit{Bool}) \tag{C.888}$$

$$\mathit{then}\ \Sigma_1 : M_1[e_1] \simeq \Sigma_2 : M_2[e_2] \tag{C.889}$$

We select any M_1 and M_2 such that Equation C.888 holds, and thus need to prove Goal C.889.

From the congruence Theorem 45 and the 1st hypothesis, we know that:

$$\Gamma_C; \bullet \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \mathit{Bool}$$

By applying the definition of logical equivalence, we get:

$$(\Sigma_1 : \gamma_1(\phi_1(R(M_1[e_1]))), \Sigma_2 : \gamma_2(\phi_2(R(M_2[e_2])))) \in \mathcal{E}[\![\mathit{Bool}]\!]_R^{\Gamma_C} \tag{C.890}$$

for any $R \in \mathcal{F}[\![\bullet]\!]^{\Gamma_C}$, $\phi \in \mathcal{G}[\![\bullet]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$ and $\gamma \in \mathcal{H}[\![\bullet]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C}$.

However, from the definition of $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$, it follows that $R = \bullet$, $\phi = \bullet$ and $\gamma = \bullet$.

Equation C.890 thus simplifies to:

$$(\Sigma_1 : M_1[e_1], \Sigma_2 : M_2[e_2]) \in \mathcal{E}[\![\mathit{Bool}]\!]_\bullet^{\Gamma_C}$$

Unfolding the definition of the $\mathcal{E}$ relation, tells us that:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} M[e_1] : \mathit{Bool}$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} M[e_2] : \mathit{Bool}$$

$$\exists v_1, v_2 : \Sigma_1 \vdash M[e_1] \longrightarrow^* v_1$$

$$\wedge\ \Sigma_2 \vdash M[e_2] \longrightarrow^* v_2$$

$$\wedge\ (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\mathit{Bool}]\!]_\bullet^{\Gamma_C}$$

From the definition of $\mathcal{V}$, we know that either $v_1 = v_2 = \mathit{True}$ or $v_1 = v_2 = \mathit{False}$. The goal follows immediately.

□

Theorem 60 (Coherence - Expressions - Part C)**.**
If $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
and $\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma \rightsquigarrow e_1$ *and* $\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2$
and $\Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

Proof. By unfolding the definition of contextual equivalence, the goal becomes:

$$\forall M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_1; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1$$

$$\forall M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2$$

$$\mathit{if}\ \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow \mathit{Bool}) \tag{C.891}$$

$$\mathit{then}\ M_1[e_1] \simeq M_2[e_2] \tag{C.892}$$

We select any M_1 and M_2 such that Equation C.891 holds, and thus need to prove Goal C.892.

By unfolding the definition of contextual equivalence in the 1st hypothesis, we get that:

$$\forall M_1 : (\Sigma_1; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1$$

$$\forall M_2 : (\Sigma_2; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma_2; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2$$

$$\mathit{if}\ \Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Gamma_C; \bullet \Rightarrow \mathit{Bool}) \tag{C.893}$$

$$\mathit{then}\ \Sigma_1 : M_1[e_1] \simeq \Sigma_2 : M_2[e_2] \tag{C.894}$$

By applying Theorem 46 to M_1 and M_2, together with the 2nd and 3rd hypothesis, we get:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} M_1[e_1] : \mathit{Bool} \rightsquigarrow M_1[e_1]$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} M_2[e_2] : \mathit{Bool} \rightsquigarrow M_2[e_2]$$

From the definition of kleene equivalence, Equation C.894 reduces to:

$$\exists v : \Sigma_1 \vdash M_1[e_1] \longrightarrow^* v \wedge \Sigma_2 \vdash M_2[e_2] \longrightarrow^* v$$

Finally, Lemma 71 applied to these results, tells us that:

$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} v : \mathit{Bool} \rightsquigarrow v_1$$

$$\Sigma_2; \Gamma_C; \bullet \vdash_{tm} v : \mathit{Bool} \rightsquigarrow v_2$$

$$M_1[e_1] \longrightarrow^* v_1$$

$$M_2[e_2] \longrightarrow^* v_2$$

Goal C.892 follows from the definition of kleene equivalence since either $v_1 = v_2 = \mathit{True}$ or $v_1 = v_2 = \mathit{False}$.

□

C.7.4 Main Coherence Theorems

Theorem 61 (Coherence). *If* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$ *then* $\Gamma_{C1} = \Gamma_{C2}$, $P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Proof. Since we know from rule sCtxT-empty that

$$\vdash_{ctx} \bullet; \bullet; \bullet \rightsquigarrow \bullet$$

the goal follows directly from the Program Coherence Theorem (Theorem 63).

□

Theorem 62 (Coherence - Expressions).

- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$ *then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.
- *If* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e_2$ *then* $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Proof. By applying Lemma 72 to the 1st hypothesis, we know that:

$$\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \qquad \text{(C.895)}$$

Part 1 From environment equivalence (Theorem 39), we get that:

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_1; \Gamma_C; \Gamma \tag{C.896}$$

$$\vdash^{M}_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma_2; \Gamma_C{}'; \Gamma' \tag{C.897}$$

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.898}$$

Since class and typing environment elaboration is entirely deterministic (Lemma 71), it is easy to see that $\Gamma_C{}' = \Gamma_C$ and $\Gamma' = \Gamma$.

We know from expression equivalence (Theorem 43) that:

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e_1 \tag{C.899}$$

$$\Sigma_1; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \tag{C.900}$$

$$P; \Gamma_C; \Gamma \vdash^{M}_{tm} e \Leftarrow \tau \rightsquigarrow e_2 \tag{C.901}$$

$$\Sigma_2; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \tag{C.902}$$

$$\textit{where } \Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma_1 \tag{C.903}$$

$$\textit{and } \Gamma_C; \Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma_2$$

Since type elaboration is entirely deterministic (Lemma 69), it is easy to see that $\sigma = \sigma_1 = \sigma_2$.

By applying Expression Coherence Theorem A (Theorem 58) to Equations C.896, C.897, C.899 and C.901, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \tag{C.904}$$

By applying Expression Coherence Theorem B (Theorem 59) to Equation C.904, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma \tag{C.905}$$

By applying Expression Coherence Theorem C (Theorem 60) to Equations C.898, C.900, C.902 and C.905, we get:

$$\Gamma_C; \Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma \tag{C.906}$$

The goal follows directly from Theorem 54, together with Equations C.906, C.895, C.896, C.897 and C.903.

Part 2 Similar to Part 1.

□

Theorem 63 (Coherence - Programs)**.**
If $P; \Gamma_C \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$,
$P; \Gamma_C \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$,
$\vdash_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \bullet$,
then $\Gamma_{C1} = \Gamma_{C2}$, $P_1 = P_2$
and $P, P_1; \Gamma_C, \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Proof. By structural induction on pgm.

$\boxed{pgm = cls; pgm'}$

By case analysis on the program typing derivations (rule sPgmT-cls):

$$\Gamma_C \vdash_{cls} cls : \Gamma_{C}{}_1' \tag{C.907}$$

$$\Gamma_C \vdash_{cls} cls : \Gamma_{C}{}_2' \tag{C.908}$$

$$P; \Gamma_C, \Gamma_{C}{}_1' \vdash_{pgm} pgm' : \tau; P_1; \Gamma_{C}{}_1'' \rightsquigarrow e_1$$

$$P; \Gamma_C, \Gamma_{C}{}_2' \vdash_{pgm} pgm' : \tau; P_2; \Gamma_{C}{}_2'' \rightsquigarrow e_2$$

$$\Gamma_{C1} = \Gamma_{C}{}_1', \Gamma_{C}{}_1''$$

$$\Gamma_{C2} = \Gamma_{C}{}_2', \Gamma_{C}{}_2''$$

Since class typing is entirely deterministic, we know that $\Gamma_{C}{}_1' = \Gamma_{C}{}_2'$. From Theorem 23, in combination with Equations C.907 and C.908, we know that:

$$\vdash_{ctx} P; \Gamma_C, \Gamma_{C}{}_1'; \bullet \rightsquigarrow \bullet$$

$$\vdash_{ctx} P; \Gamma_C, \Gamma_{C}{}_2'; \bullet \rightsquigarrow \bullet$$

The goal follows from the induction hypothesis.

$\boxed{pgm = inst; pgm'}$

By case analysis on the program typing derivations (rule sPgmT-inst):

$$P; \Gamma_C \vdash_{inst} inst : P_{11} \tag{C.909}$$

$$P; \Gamma_C \vdash_{inst} inst : P_{21} \tag{C.910}$$

$$P, P_{11}; \Gamma_C \vdash_{pgm} pgm' : \tau; P_{12}; \Gamma_{C1} \rightsquigarrow e_1 \tag{C.911}$$

$$P, P_{21}; \Gamma_C \vdash_{pgm} pgm' : \tau; P_{22}; \Gamma_{C2} \rightsquigarrow e_2 \tag{C.912}$$

$$P_1 = P_{11}, P_{12}$$

$$P_2 = P_{21}, P_{22}$$

The goal to be proven is the following:

$$\Gamma_{C1} = \Gamma_{C2} \tag{C.913}$$

$$P_{11}, P_{12} = P_{21}, P_{22} \tag{C.914}$$

$$P, P_{11}, P_{12}; \Gamma_C, \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau \tag{C.915}$$

By case analysis on Equations C.909 and C.910 (rule sInstT-inst), we know that:

$$inst = \mathbf{instance}\ \overline{C}_p \Rightarrow TC\,\tau\,\mathbf{where}\ \{m = e\}$$

$$(m : \overline{C}'_i \Rightarrow TC\,a : \forall \overline{a}_j.\overline{C}'_y \Rightarrow \tau_1) \in \Gamma_C \tag{C.916}$$

$$\overline{b}_k = \mathbf{fv}(\tau')$$

$$\mathbf{closure}(\Gamma_C; \overline{C}_p) = \overline{C}_{q\,1}$$

$$\mathbf{closure}(\Gamma_C; \overline{C}_p) = \overline{C}_{q\,2}$$

$$\mathbf{unambig}(\forall \overline{b}_k.\overline{C}_{q\,1} \Rightarrow TC\,\tau') \tag{C.917}$$

$$\mathbf{unambig}(\forall \overline{b}_k.\overline{C}_{q\,2} \Rightarrow TC\,\tau')$$

$$\overline{P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,1} : \overline{C}_{q\,1} \vDash [[\tau'/a]C'_i] \rightsquigarrow e_{1\,i}}^{\,i}$$

$$\overline{P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,2} : \overline{C}_{q\,2} \vDash [[\tau'/a]C'_i] \rightsquigarrow e_{2\,i}}^{\,i}$$

$$D, \overline{\delta}_{q\,1}, \overline{\delta}_{q\,2}, \overline{\delta}'_y\ \mathbf{fresh} \tag{C.918}$$

$$P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,1} : \overline{C}_{q\,1}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau'/a]\tau_1 \rightsquigarrow e'_1 \tag{C.919}$$

$$P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,2} : \overline{C}_{q\,2}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau'/a]\tau_1 \rightsquigarrow e'_2 \tag{C.920}$$

$$P_{11} = (D : \forall \overline{b}_k . \overline{C}_{q\,1} \Rightarrow TC\,\tau').m \mapsto \bullet, \overline{b}_k, \overline{\delta}_{q\,1} : \overline{C}_{q\,1}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y : e_1 \tag{C.921}$$

$$P_{21} = (D : \forall \overline{b}_k . \overline{C}_{q\,2} \Rightarrow TC\,\tau').m \mapsto \bullet, \overline{b}_k, \overline{\delta}_{q\,2} : \overline{C}_{q\,2}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y : e_2 \tag{C.922}$$

$$\Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau' \rightsquigarrow \sigma' \tag{C.923}$$

$$\frac{}{\Gamma_C; \bullet, \overline{b}_k \vdash_C C_{q\,1} \rightsquigarrow \sigma_{q\,1}}\,q \tag{C.924}$$

$$\frac{}{\Gamma_C; \bullet, \overline{b}_k \vdash_C C_{q\,2} \rightsquigarrow \sigma_{q\,2}}\,q \tag{C.925}$$

$$(D' : \forall \overline{b}'_w . \overline{C}'_n \Rightarrow TC\,\tau_2).m' \mapsto \Gamma' : e' \notin P \textit{ where } [\overline{\tau}'_w/\overline{b}'_w]\tau_2 = [\overline{\tau}'_k/\overline{b}_k]\tau' \tag{C.926}$$

Note that since instance typing is entirely deterministic, $P_{11} = P_{21}$. Similarly, since the closure over the superclass relation is deterministic, we know that $\overline{C}_{q\,1} = \overline{C}_{q\,2}$. Note that we assume that the fresh variables are identical in both program typing derivations.

We can derive from rule sCtxT-pgmInst that

$$\vdash_{ctx} P, P_{11}; \Gamma_C; \bullet \rightsquigarrow \bullet \tag{C.927}$$

$$\vdash_{ctx} P, P_{21}; \Gamma_C; \bullet \rightsquigarrow \bullet \tag{C.928}$$

assuming we can show that:

$$\mathbf{unambig}(\forall \overline{b}_k.\overline{C}_{q\,1} \Rightarrow TC\,\tau') \tag{C.929}$$

$$\Gamma_C; \bullet \vdash_C \forall \overline{b}_k.\overline{C}_{q\,1} \Rightarrow TC\,\tau' \rightsquigarrow \forall \overline{b}_k.\overline{\sigma}_{q\,1} \to [\sigma'/a]\{m : \forall \overline{a}_j.\overline{\sigma}'_y \to \sigma_1\} \tag{C.930}$$

$$(m : \overline{C}'_i \Rightarrow TC\,a : \forall \overline{a}_j.\overline{C}'_y \Rightarrow \tau_1) \in \Gamma_C \tag{C.931}$$

$$\Gamma_C; \bullet, a \vdash_{ty} \forall \overline{a}_j.\overline{C}'_y \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}'_y \to \sigma_1 \tag{C.932}$$

$$\Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau' \rightsquigarrow \sigma' \tag{C.933}$$

$$P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,1} : \overline{C}_{q\,1}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau'/a]\tau_1 \rightsquigarrow e'_1 \tag{C.934}$$

$$P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_{q\,2} : \overline{C}_{q\,2}, \overline{a}_j, \overline{\delta}'_y : [\tau'/a]\overline{C}'_y \vdash_{tm} e \Leftarrow [\tau'/a]\tau_1 \rightsquigarrow e'_2 \tag{C.935}$$

$$D \notin \mathbf{dom}(P) \tag{C.936}$$

$$(D' : \forall \overline{b}'_k.\overline{C}''_y \Rightarrow TC\,\tau'').m' \mapsto \Gamma' : e' \notin P \textit{ where } [\overline{\tau}_k/\overline{b}_k]\tau' = [\overline{\tau}'_k/\overline{b}'_k]\tau'' \tag{C.937}$$

$$\vdash_{ctx} P; \Gamma_C; \bullet \rightsquigarrow \bullet \tag{C.938}$$

Goals C.929, C.931, C.933, C.934, C.935, C.936 and C.937 follow directly from Equations C.917, C.916, C.923, C.919, C.920, C.918 and C.926 respectively.

Goal C.938 follows directly from the 3rd hypothesis. Goal C.932 follows by applying case analysis on Equation C.938 (rule sCtxT-clsEnv), in combination with Equation C.931. Goal C.930 follows from rule sCT-Abs and rule sQT-TC, in combination with Equations C.931, C.932, C.923 and C.924.

Finally, Goals C.913, C.914 and C.915 follow by applying the induction hypothesis on Equations C.911 and C.912, in combination with Equations C.927 and C.928.

$\boxed{pgm = e}$

The goal follows directly from coherence Theorem 62. □

C.8 F_{D}-to-$F_{\{\}}$ Theorems

C.8.1 Lemmas

Determinism / Uniqueness

Lemma 132 (Dictionary Elaboration Uniqueness).
If $\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_1$ *and* $\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_2$*, then* $\sigma_1 = \sigma_2$*.*

Proof. By straightforward induction on the well-formedness derivation. □

Lemma 133 (Type Elaboration Uniqueness).
If $\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma_1$ *and* $\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma_2$*, then* $\sigma_1 = \sigma_2$*.*

Proof. By straightforward induction on the well-formedness derivation. □

Lemma 134 (Context Elaboration Uniqueness).
If $\Gamma_C ; \Gamma \rightsquigarrow \Gamma_1$ *and* $\Gamma_C ; \Gamma \rightsquigarrow \Gamma_2$ *then* $\Gamma_1 = \Gamma_2$*.*

Proof. By straightforward induction on the well-formedness derivation. □

Lemma 135 (Determinism of Evaluation).
If $e \longrightarrow e_1$ *and* $e \longrightarrow e_2$ *then* $e_1 = e_2$*.*

Proof. By straightforward induction on both evaluation derivations. □

Soundness

Lemma 136 (Constraint Elaboration Soundness).
If $\Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma$ *and* $\Gamma_C ; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma \vdash_{ty} \sigma$*.*

Proof. By straightforward induction on the dictionary typing derivation. □

Lemma 137 (Type Elaboration Soundness)**.**
If $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *then* $\Gamma \vdash_{ty} \sigma$*.*

Proof. By straightforward induction on the type well-formedness derivation. □

Lemma 138 (Term Variable Elaboration Soundness)**.**
If $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$ *and* $(x : \sigma) \in \Gamma$*, then there are unique* Γ *and* σ *such that* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $(x : \sigma) \in \Gamma$*.*

Proof. By straightforward induction on the environment well-formedness derivation. □

Lemma 139 (Dictionary Variable in Environment Elaboration Soundness)**.**

If $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$ *and* $(\delta : C) \in \Gamma$*, then there are unique* Γ *and* σ *such that* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} \Gamma$ *and* $\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma$ *and* $(\delta : \sigma) \in \Gamma$*.*

Proof. By straightforward induction on the environment well-formedness derivation. □

Lemma 140 (Environment Elaboration Soundness)**.**
If $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$*, then there is a unique* Γ *such that* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\vdash_{ctx} \Gamma$*.*

Proof. By straightforward induction on the environment well-formedness derivation. □

Canonical Forms Lemmas

Lemma 141 (Canonical Forms for Functions)**.**
If $\Gamma \vdash_{tm} v : \sigma_1 \to \sigma_2$ *for some value* v*, then* v *is of the form* $\lambda x : \sigma_1 . e$*, for some* x *and* e*.*

Proof. By straightforward induction on the typing derivation. □

Lemma 142 (Canonical Forms for Type Abstractions)**.**
If $\Gamma \vdash_{tm} v : \forall a . \sigma$ *for some value* v*, then* v *is of the form* $\Lambda a . e$*, for some* e*.*

Proof. By straightforward induction on the typing derivation. □

Lemma 143 (Canonical Forms for Records)**.**
If $\Gamma \vdash_{tm} v : \{m : \sigma\}$ *for some value* v*, then* v *is of the form* $\{m = e\}$*, for some* e*.*

Proof. By straightforward induction on the typing derivation. □

Evaluation Lemmas

Lemma 144 (Distribution of tEval-App)**.**
If $\bullet \vdash_{tm} e : \sigma_1 \to \sigma_2$ *and* $\bullet \vdash_{tm} e' : \sigma_1 \to \sigma_2$ *and* $\bullet \vdash_{tm} e_1 : \sigma_1$ *and* $e \longrightarrow^* e'$*, then* $e\, e_1 \longrightarrow^* e'\, e_1$*.*

Proof. The goal follows from Canonical Forms Lemma 141, together with the well-known strong normalization of System F with records. □

Lemma 145 (Distribution of tEval-Tapp)**.**
If $\bullet \vdash_{tm} e : \forall a . \sigma'$ *and* $\bullet \vdash_{tm} e' : \forall a . \sigma'$ *and* $\bullet \vdash_{ty} \sigma$ *and* $e \longrightarrow^* e'$*, then* $e\, \sigma \longrightarrow^* e'\, \sigma$*.*

Proof. The goal follows from Canonical Forms Lemma 142, together with the well-known strong normalization of System F with records. □

Lemma 146 (Distribution of tEVAL-REC)**.**
If $\bullet \vdash_{tm} e : \{m : \sigma\}$ *and* $\bullet \vdash_{tm} e' : \{m : \sigma\}$ *and* $e \longrightarrow^{*} e'$, *then* $e.m \longrightarrow^{*} e'.m$.

Proof. The goal follows from Canonical Forms Lemma 143, together with the well-known strong normalization of System F with records.

□

C.8.2 Soundness

Theorem 64 (Term Elaboration Soundness)**.**

If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$ (C.939)

then, there are unique Γ *and* σ *such that*

$\Gamma_C; \Gamma \rightsquigarrow \Gamma$ (C.940)

and $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ (C.941)

and $\Gamma \vdash_{tm} e : \sigma$ (C.942)

Proof. This theorem is proved mutually with Theorem 65. The proof follows structural induction on Hypothesis C.939 of the theorem.

Case rule ITM-TRUE

$$\textsc{iTm-true}\quad \frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \rightsquigarrow \mathit{True}}$$

We need to show that there are unique Γ and σ such that $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ and $\Gamma_C; \Gamma \vdash_{ty} \mathit{Bool} \rightsquigarrow \sigma$ and $\Gamma \vdash_{tm} \mathit{True} : \sigma$.

Obviously, σ can only be equal to Bool. By Lemma 140 applied on the premise of rule rule ITM-TRUE, there is a unique Γ such that $\Gamma_C; \Gamma \rightsquigarrow \Gamma$ and $\vdash_{ctx} \Gamma$. We use the latter result to instantiate rule rule TTM-TRUE, which concludes with $\Gamma \vdash_{tm} \mathit{True} : \mathit{Bool}$.

Case rule ITM-FALSE

$$\textsc{iTm-false}\quad \frac{\vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \rightsquigarrow \mathit{False}}$$

Similar to case rule ITM-TRUE.

$$\textsc{iTm-var}\quad \frac{\begin{array}{c}(x : \sigma) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma \rightsquigarrow x}$$

Case rule iTm-var

By Lemma 140, there is a unique Γ such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma$$

$$\text{and } \vdash_{ctx} \Gamma \tag{C.943}$$

With Lemma 138 applied on the two premises of rule rule iTm-var, we also find a unique σ such that $\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ and $(x : \sigma) \in \Gamma$. Then, by rule rule tTm-Var applied on the latter result and on Equation C.943, we reach the goal.

Case rule iTm-let

$$\textsc{iTm-let}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2 \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2 : \sigma_2 \rightsquigarrow \textbf{let } x : \sigma_1 = e_1 \textbf{ in } e_2}$$

The induction hypothesis for the first premise of rule rule iTm-let is

$$\text{There are unique } \Gamma,\ \sigma_1,\ \text{such}$$

$$\text{that } \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.944}$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.945}$$

$$\text{and } \Gamma \vdash_{tm} e_1 : \sigma_1 \tag{C.946}$$

The induction hypothesis for the second premise of rule rule iTm-let is

$$\text{There are unique } \Gamma',\ \sigma_2,\ \text{such}$$

$$\text{that } \Gamma_C; \Gamma, x : \sigma_1 \rightsquigarrow \Gamma' \tag{C.947}$$

$$\text{and } \Gamma_C; \Gamma, x : \sigma_1 \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2 \tag{C.948}$$

$$\text{and } \Gamma' \vdash_{tm} e_2 : \sigma_2 \tag{C.949}$$

By inversion on Equation C.947, there are Γ_0 and σ_1' such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma_0 \tag{C.950}$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1' \tag{C.951}$$

$$\text{and } \Gamma' = \Gamma_0, x : \sigma_1'$$

By uniqueness (Lemma 134 on Equations C.944 and C.950 and Lemma 133 on Equations C.945 and C.951), we get $\Gamma_0 = \Gamma$ and $\sigma'_1 = \sigma_1$. This refines Equation C.949 into

$$\Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \tag{C.952}$$

By applying Lemma 137 on Equations C.945 and C.944, we get

$$\Gamma \vdash_{ty} \sigma_1 \tag{C.953}$$

By applying rule rule TTM-LET on Equations C.946, C.952 and C.953, we get

$$\Gamma \vdash_{tm} \textbf{let}\ \ x : \sigma_1 = e_1\ \textbf{in}\ \ e_2 : \sigma_2 \tag{C.954}$$

It remains to show that $\Gamma_C; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2$. This is easily derived from Equation C.948, since the existence of variable x in the context does not affect the well-formedness nor the translation of σ_2.

$$\boxed{\textbf{Case rule}\ \textsc{iTm-Method}} \qquad \begin{array}{c}\textsc{iTm-method}\\ \dfrac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma \ \rightsquigarrow e\\ (m : TC\,a : \sigma') \in \Gamma_C\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma/a]\sigma' \rightsquigarrow e.m}\end{array}$$

Applying Theorem 65 to the first premise of rule rule ITM-METHOD results in

There are unique Γ and σ_1 such

$$\text{that } \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.955}$$

$$\text{and } \Gamma_C; \Gamma \vdash_Q TC\,\sigma \ \rightsquigarrow \sigma_1 \tag{C.956}$$

$$\text{and } \Gamma \vdash_{tm} e : \sigma_1 \tag{C.957}$$

By inversion on Equation C.956, we get

$$\Gamma_C = \Gamma_{C1}, m' : TC\,a' : \sigma_m, \Gamma_{C2} \tag{C.958}$$

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.959}$$

$$\Gamma_{C1}; \bullet, a' \vdash_{ty} \sigma_m \rightsquigarrow \sigma_m \tag{C.960}$$

for some m', a', σ_m, σ and σ_m. However, each dictionary TC corresponds to a unique entry in the class environment Γ_C. By this uniqueness, we get $m' = m$, $a = a'$, $\sigma_m = \sigma'$. Then, $\sigma_1 = [\sigma/a]\{m : \sigma_m\}$, and Equations C.956 and C.957 become

$$\Gamma_C; \Gamma \vdash_Q TC\,\sigma \ \rightsquigarrow [\sigma/a]\{m : \sigma_m\}$$

$$\text{and } \Gamma \vdash_{tm} e : [\sigma/a]\{m : \sigma_m\} \tag{C.961}$$

Lemma 82 applied on Equations C.959 and C.960, results in

$$\Gamma_C; \Gamma \vdash_{ty} [\sigma/a]\sigma' \rightsquigarrow [\sigma/a]\sigma_m$$

and rule rule tTm-Proj instantiated with Equation C.961 gives $\Gamma \vdash_{tm} e.m : [\sigma/a]\sigma_m$.

Case rule iTm-arrI

$$\textsc{iTm-arrI}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.e}$$

The induction hypothesis from the first premise of rule rule iTm-arrI is

$$\text{There are unique } \Gamma' \text{ and } \sigma_1 \text{ such}$$

$$\text{that } \Gamma_C; \Gamma, x : \sigma_1 \rightsquigarrow \Gamma'$$

$$\text{and } \Gamma_C; \Gamma, x : \sigma_1 \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2 \tag{C.962}$$

$$\text{and } \Gamma' \vdash_{tm} e : \sigma_2 \tag{C.963}$$

Similarly to the rule iTm-Let case, Γ' is of the form $\Gamma, x : \sigma_1$, where

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.964}$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.965}$$

In addition, from Lemma 137 applied on Equation C.964 and the second premise of rule rule iTm-arrI, we obtain $\Gamma \vdash_{ty} \sigma_1$.

By instantiating the premises of rule rule tTm-Abs with the above result and with Equation C.963, we get $\Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2$, where $\Gamma_C; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2$. The latter holds from Equation C.962, where x has been removed from the context (it does not affect the well-formedness of type σ_2). This, in combination with Equation C.965 in rule rule iTy-Arr, gives $\Gamma_C; \Gamma \vdash_{ty} \sigma_1 \to \sigma_2 \rightsquigarrow \sigma_2 \to \sigma_2$.

Case rule iTm-arrE

$$\textsc{iTm-arrE}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2}$$

The induction hypothesis from the first premise of rule rule iTm-arrE is

$$\text{There are unique } \Gamma \text{ and } \sigma \text{ such}$$

$$\text{that } \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.966}$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \to \sigma_2 \rightsquigarrow \sigma \tag{C.967}$$

$$\text{and } \Gamma \vdash_{tm} e_1 : \sigma \tag{C.968}$$

By inversion on Equation C.967, σ can only be of the form $\sigma_1 \rightarrow \sigma_2$ for the σ_1 and σ_2, uniquely determined by equations

$$\Gamma_C ; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{C.969}$$

$$\text{and } \Gamma_C ; \Gamma \vdash_{ty} \sigma_2 \rightsquigarrow \sigma_2$$

. The induction hypothesis from the second premise of rule rule iTm-arrE is

$$\text{There are unique } \Gamma' \text{ and } \sigma_1' \text{ such}$$

$$\text{that } \Gamma_C ; \Gamma \rightsquigarrow \Gamma' \tag{C.970}$$

$$\text{and } \Gamma_C ; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1' \tag{C.971}$$

$$\text{and } \Gamma \vdash_{tm} e_2 : \sigma_1' \tag{C.972}$$

By uniqueness (Lemma 134) on Equations C.966 and C.970, it must hold that $\Gamma' = \Gamma$. Also, uniqueness (Lemma 133) on Equations C.969 and C.971, gives $\sigma_1' = \sigma_1$.

Combining Equations C.968 and C.972, (rewritten with the equalities holding so far) in rule rule tTm-App, we get $\Gamma \vdash_{tm} e_1\, e_2 : \sigma_2$.

Case rule iTm-constrI

$$\frac{\begin{array}{c}\Sigma ; \Gamma_C ; \Gamma , \delta : C \vdash_{tm} e : \sigma \rightsquigarrow e \\ \Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma\end{array}}{\Sigma ; \Gamma_C ; \Gamma \vdash_{tm} \lambda \delta : C . e : C \Rightarrow \sigma \rightsquigarrow \lambda \delta : \sigma . e}\;\text{iTm-constrI}$$

The induction hypothesis from the first premise of rule rule iTm-constrI is

$$\text{There are unique } \Gamma' \text{ and } \sigma \text{ such}$$

$$\text{that } \Gamma_C ; \Gamma , \delta : C \rightsquigarrow \Gamma'$$

$$\text{and } \Gamma_C ; \Gamma , \delta : C \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.973}$$

$$\text{and } \Gamma' \vdash_{tm} e : \sigma \tag{C.974}$$

Similarly to the rule iTm-Let case, Γ' is of the form $\Gamma, \delta : \sigma_q$, where

$$\Gamma_C ; \Gamma \rightsquigarrow \Gamma \tag{C.975}$$

$$\text{and } \Gamma_C ; \Gamma \vdash_C C \rightsquigarrow \sigma_q \tag{C.976}$$

In addition, from Lemma 136 applied on Equation C.975 and the second premise of rule rule iTm-constrI, we obtain $\Gamma \vdash_{ty} \sigma_1$.

By instantiating the premises of rule rule tTm-Abs with the above result and with Equation C.974, we get $\Gamma \vdash_{tm} \lambda x : \sigma_q . e : \sigma_q \rightarrow \sigma$, where $\Gamma_C ; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$.

The latter holds from Equation C.973, where x has been removed from the context (it does not affect the well-formedness of type σ_2). This, in combination with Equation C.976 in rule rule iTy-Qual, gives $\Gamma_C;\Gamma \vdash_{ty} C \Rightarrow \sigma \rightsquigarrow \sigma_q \rightarrow \sigma$.

Case rule iTm-constrE

$$\text{iTm-constrE}\quad \frac{\begin{array}{c}\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : C \Rightarrow \sigma \rightsquigarrow e_1 \\ \Sigma;\Gamma_C;\Gamma \vdash_{d} d : C \rightsquigarrow e_2\end{array}}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1\, e_2}$$

The induction hypothesis from the first premise of rule rule iTm-constrE is

There are unique Γ and σ' such

$$\text{that } \Gamma_C;\Gamma \rightsquigarrow \Gamma \tag{C.977}$$

$$\text{and } \Gamma_C;\Gamma \vdash_{ty} C \Rightarrow \sigma \rightsquigarrow \sigma' \tag{C.978}$$

$$\text{and } \Gamma \vdash_{tm} e_1 : \sigma \tag{C.979}$$

By inversion on Equation C.978, σ' can only be of the form $\sigma_q \rightarrow \sigma$ for the σ_q and σ, uniquely determined by equations

$$\Gamma_C;\Gamma \vdash_{C} C \rightsquigarrow \sigma_q \tag{C.980}$$

$$\text{and } \Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$$

Theorem 65, applied to the second premise of rule rule iTm-constrE, is

There are unique Γ' and σ_0 such

$$\text{that } \Gamma_C;\Gamma \rightsquigarrow \Gamma' \tag{C.981}$$

$$\text{and } \Gamma_C;\Gamma \vdash_{C} C \rightsquigarrow \sigma_0 \tag{C.982}$$

$$\text{and } \Gamma' \vdash_{tm} e_2 : \sigma_0 \tag{C.983}$$

By uniqueness (Lemma 134) on Equations C.977 and C.981, it must hold that $\Gamma' = \Gamma$. Also, uniqueness (Lemma 132) on Equations C.980 and C.982, gives $\sigma_0 = \sigma_q$.

Combining Equations C.979 and C.983, (rewritten with the above-mentioned equalities) in rule rule tTm-App, we get $\Gamma \vdash_{tm} e_1\, e_2 : \sigma$.

Case rule iTm-forallI

$$\text{iTm-forallI}\quad \frac{\Sigma;\Gamma_C;\Gamma, a \vdash_{tm} e : \sigma \rightsquigarrow e}{\Sigma;\Gamma_C;\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma \rightsquigarrow \Lambda a.e}$$

The induction hypothesis for the premise of rule rule iTm-forallI is the following.

$$\text{There are unique } \Gamma' \text{ and } \sigma \text{ such}$$

$$\text{that } \Gamma_C; \Gamma, a \rightsquigarrow \Gamma' \tag{C.984}$$

$$\text{and } \Gamma_C; \Gamma, a \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.985}$$

$$\text{and } \Gamma' \vdash_{tm} e : \sigma \tag{C.986}$$

By inversion on Equation C.984, Γ' can only be of the form Γ, a, where Γ uniquely determined by

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma$$

Then, Equation C.986 becomes

$$\Gamma, a \vdash_{tm} e : \sigma$$

Using this to instantiate rule rule tTm-Tabs, we get

$$\Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma$$

where, by rule rule iTy-scheme on Equation C.985, it follows that $\Gamma_C; \Gamma \vdash_{ty} \forall a.\sigma \rightsquigarrow \forall a.\sigma$.

Case rule iTm-forallE

$$\dfrac{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \rightsquigarrow e \qquad \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\,\sigma : [\sigma/a]\sigma' \rightsquigarrow e\,\sigma}\ \text{iTm-forallE}$$

The induction hypothesis from the first premise of rule rule iTm-forallE is as follows.

$$\text{There are unique } \Gamma \text{ and } \sigma_0 \text{ such}$$

$$\text{that } \Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.987}$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \forall a.\sigma' \rightsquigarrow \sigma_0 \tag{C.988}$$

$$\text{and } \Gamma \vdash_{tm} e : \sigma_0 \tag{C.989}$$

By inversion on Equation C.988, σ_0 can only be of the form $\forall a.\sigma'$, where σ' is uniquely determined by equation

$$\Gamma_C; \Gamma, a \vdash_{ty} \sigma' \rightsquigarrow \sigma' \tag{C.990}$$

Lemma 82, applied on the second premise of rule rule iTm-forallE and on Equation C.990, results in $\Gamma_C; \Gamma \vdash_{ty} [\sigma/a]\sigma' \rightsquigarrow [\sigma/a]\sigma'$.

Also, Lemma 137 applied on the second premise of rule rule ITM-FORALLE and on Equation C.987 results in $\Gamma \vdash_{ty} \sigma$. We use the latter, together with Equation C.989, to instantiate rule rule TTM-TAPP, which concludes $\Gamma \vdash_{tm} e : [\sigma/a]\sigma'$. □

Theorem 65 (Dictionary Elaboration Soundness)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_d d : C \leadsto e$ (C.991)

then, there are unique Γ *and* σ *such that*

$$\Gamma_C; \Gamma \leadsto \Gamma \tag{C.992}$$

and $\Gamma_C; \Gamma \vdash_C C \leadsto \sigma$ (C.993)

and $\Gamma \vdash_{tm} e : \sigma$ (C.994)

Proof. This theorem is proved mutually with Theorem 64. The proof follows structural induction on the first hypothesis.

$$\textsc{D-var}\quad \frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C \leadsto \delta}$$

Case rule D-VAR

By Lemma 139, there exist Γ and σ such that $\Gamma_C; \Gamma \leadsto \Gamma$ and $\Gamma_C; \Gamma \vdash_C C \leadsto \sigma$ and $(\delta : \sigma) \in \Gamma$. This satisfies Equations C.992 and C.993 of the theorem.

By instantiating rule rule TTM-VAR on $(\delta : \sigma) \in \Gamma$, we get $\Gamma \vdash_{tm} \delta : \sigma$, which satisfies Equation C.994 of the theorem.

$$\textsc{D-con}\quad \frac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \leadsto \sigma'_i}^{\,i} \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \leadsto e\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \leadsto \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\}}$$

Case rule D-CON

We need to show that an Γ and σ exists such that

$$\Gamma_C; \Gamma \leadsto \Gamma \tag{C.995}$$

$$\Gamma_C; \Gamma \vdash_C \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \leadsto \sigma \tag{C.996}$$

$$\Gamma \vdash_{tm} \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\} : \sigma \tag{C.997}$$

Goal C.995 follows by applying Lemma 140 to the 2nd rule premise. Goal C.996 follows from the 1st rule premise, along with the well-formedness of the typing context (rule iCtx-MEnv). By case analysis on this result (rule iC-forall, rule iC-arrow and rule iC-classconstr) we know that

$$\sigma = \forall \overline{a}_j . \overline{\sigma}'_i \to \sigma'$$

where $\Gamma_C ; \Gamma \vdash_Q \mathit{TC}\,\sigma_q \rightsquigarrow \sigma'$. Again applying case analysis on this last result (rule iQ-TC) gives us

$$\Gamma_C ; \Gamma \vdash_{\mathit{ty}} \sigma_q \rightsquigarrow \sigma_q \tag{C.998}$$

$$\Gamma_C = \Gamma_{C1}, m : \mathit{TC}\,a : \sigma_m, \Gamma_{C2} \tag{C.999}$$

$$\Gamma_{C1} ; \bullet, a \vdash_{\mathit{ty}} \sigma_m \rightsquigarrow \sigma_m \tag{C.1000}$$

$$\sigma' = [\sigma_q / a]\{m : \sigma_m\} \tag{C.1001}$$

From these results, Goal C.997 reduces to

$$\Gamma \vdash_{\mathit{tm}} \Lambda \overline{a}_j . \lambda \overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\} : \forall \overline{a}_j . \overline{\sigma}'_i \to [\sigma_q / a]\{m : \sigma_m\}$$

By rule tTm-TAbs, rule tTm-Abs and rule tTm-Rec, this goal reduces to

$$\Gamma, \overline{a}_j, \overline{\delta_i : \sigma'_i}^{\,i} \vdash_{\mathit{tm}} e : [\sigma_q / a]\sigma_m \tag{C.1002}$$

Theorem 64, applied to the 5th premise of rule rule D-con, is:

There are Γ' and σ'' such that

$$\Gamma_C ; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \rightsquigarrow \Gamma'$$

$$\text{and } \Gamma_C ; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{\mathit{ty}} [\sigma_q / a]\sigma_m \rightsquigarrow \sigma'' \tag{C.1003}$$

$$\text{and } \Gamma' \vdash_{\mathit{tm}} e : \sigma''. \tag{C.1004}$$

It is easy to verify that $\Gamma' = \bullet, \overline{a}_j, \overline{\delta_i : \sigma'_i}^{\,i}$. From Lemma 97 on Equation C.1000, we get the weakened equation

$$\Gamma_C ; \bullet, a \vdash_{\mathit{ty}} \sigma_m \rightsquigarrow \sigma_m$$

where Γ_C is specified in Equation C.999. Using this result together with Equation C.998 in Lemma 82, we get $\Gamma_C ; \bullet, \overline{a}_j \vdash_{\mathit{ty}} [\sigma_q / a]\sigma_m \rightsquigarrow [\sigma_q / a]\sigma_m$, which we weaken as

$$\Gamma_C ; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{\mathit{ty}} [\sigma_q / a]\sigma_m \rightsquigarrow [\sigma_q / a]\sigma_m$$

Then, by uniqueness on the latter and on Equation C.1003, we have $\sigma'' = [\sigma_q/a]\sigma_m$, and Equation C.1004 becomes

$$\bullet, \overline{a}_j, \overline{\delta_i : \sigma'_i}^{\,i} \vdash_{tm} e : [\sigma_q/a]\sigma_m$$

Prefixing the typing environment of the above with Γ, Goal C.1002 is satisfied.

$$\boxed{\textbf{Case rule } \textsc{D-dabs}} \qquad \overset{\textsc{D-dabs}}{\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \lambda\delta : C_1.d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda\delta : \sigma_1.e}}$$

We need to show that an Γ and σ exists such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1005}$$

$$\Gamma_C; \Gamma \vdash_C C_1 \Rightarrow C_2 \rightsquigarrow \sigma \tag{C.1006}$$

$$\Gamma \vdash_{tm} \lambda\delta : \sigma_1.e : \sigma \tag{C.1007}$$

Applying the induction hypothesis to the 1st rule premise gives us

$$\Gamma_C; \Gamma, \delta : C_1 \rightsquigarrow \Gamma' \tag{C.1008}$$

$$\Gamma_C; \Gamma, \delta : C_1 \vdash_C C_2 \rightsquigarrow \sigma_2 \tag{C.1009}$$

$$\Gamma' \vdash_{tm} e : \sigma_2 \tag{C.1010}$$

Goal C.1005 follows by inversion on Equation C.1008 (rule Ctx-DVar) with $\Gamma' = \Gamma'', \delta : \sigma'_1$. Furthermore, by the uniqueness of typing context and constraint translation, we know that $\Gamma'' = \Gamma$ and $\sigma'_1 = \sigma_1$. Goal C.1006 follows by rule iC-Arrow, using Equation C.1009 (in combination with Lemma 106) and the 2nd rule premise, where $\sigma = \sigma_1 \rightarrow \sigma_2$. Finally, Goal C.1007 follows by rule tTm-Abs using Equation C.1010.

$$\boxed{\textbf{Case rule } \textsc{D-dapp}} \qquad \overset{\textsc{D-dapp}}{\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e_1\, e_2}}$$

We need to show that an Γ and σ_2 exists such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1011}$$

$$\Gamma_C; \Gamma \vdash_C C_2 \rightsquigarrow \sigma_2 \tag{C.1012}$$

$$\Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \tag{C.1013}$$

Applying the induction hypothesis on both rule premises gives us

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1014}$$

$$\Gamma_C; \Gamma \vdash_C C_1 \Rightarrow C_2 \rightsquigarrow \sigma' \tag{C.1015}$$

$$\Gamma \vdash_{tm} e_1 : \sigma' \tag{C.1016}$$

$$\Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1 \tag{C.1017}$$

$$\Gamma \vdash_{tm} e_2 : \sigma_1 \tag{C.1018}$$

Goal C.1011 thus follows directly by Equation C.1014. Goal C.1012 follows by inversion on Equation C.1015 (rule IC-ARROW). Furthermore, using the uniqueness of constraint translation, we know that $\sigma' = \sigma_1 \rightarrow \sigma_2$. Finally, Goal C.1013 follows using rule TTM-APP, in combination with Equations C.1016 and C.1018.

Case rule D-TYABS

$$\textsc{D-tyabs}\quad \frac{\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e}{\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e}$$

We need to show that an Γ and σ exists such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1019}$$

$$\Gamma_C; \Gamma \vdash_C \forall a.C \rightsquigarrow \sigma \tag{C.1020}$$

$$\Gamma \vdash_{tm} \Lambda a.e : \sigma \tag{C.1021}$$

Applying the induction hypothesis to the rule premise gives us

$$\Gamma_C; \Gamma, a \rightsquigarrow \Gamma' \tag{C.1022}$$

$$\Gamma_C; \Gamma, a \vdash_C C \rightsquigarrow \sigma' \tag{C.1023}$$

$$\Gamma' \vdash_{tm} e : \sigma' \tag{C.1024}$$

Goal C.1019 follows by inversion on Equation C.1022 (rule CTX-TVAR) with $\Gamma' = \Gamma'', a$. Furthermore, by the uniqueness of typing context translation, we know that $\Gamma'' = \Gamma$. Goal C.1020 follows by rule IC-FORALL, using Equation C.1023, where $\sigma = \forall a.\sigma'$. Finally, Goal C.1021 follows by rule TTM-TABS using Equation C.1024.

Case rule D-TYAPP

$$\textsc{D-tyapp}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d\,\sigma : [\sigma/a]C \rightsquigarrow e\,\sigma}$$

We need to show that an Γ and σ_1 exists such that

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1025}$$

$$\Gamma_C; \Gamma \vdash_C [\sigma/a]C \rightsquigarrow \sigma_1 \tag{C.1026}$$

$$\Gamma \vdash_{tm} e\,\sigma : \sigma_1 \tag{C.1027}$$

Applying the induction hypothesis on the 1st rule premise.

$$\Gamma_C; \Gamma \rightsquigarrow \Gamma \tag{C.1028}$$

$$\Gamma_C; \Gamma \vdash_C \forall a.C \rightsquigarrow \sigma_2 \tag{C.1029}$$

$$\Gamma \vdash_{tm} e : \sigma_2 \tag{C.1030}$$

Goal C.1025 thus follows directly by Equation C.1028. By inversion on Equation C.1029 (rule iC-forall) we know that $\Gamma_C; \Gamma, a \vdash_C C \rightsquigarrow \sigma_3$ where $\sigma_2 = \forall a.\sigma_3$. Goal C.1026 follows by applying Lemma 84 to this result, along with the 2nd rule premise, where $\sigma_1 = [\sigma/a]\sigma_3$. Applying Lemma 137 to the 2nd rule premise gives us

$$\Gamma \vdash_{ty} \sigma$$

Finally, Goal C.1027 follows using rule tTm-TApp, in combination with this result and Equation C.1030. □

C.8.3 Determinism

Theorem 66 (Deterministic Dictionary Elaboration)**.**
If $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e_1$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e_2$*, then* $e_1 = e_2$*.*

Proof. This theorem is proved mutually with Theorem 67. The proof follows structural induction on both hypotheses.

$$\boxed{\textbf{Case rule D-VAR}} \qquad \frac{\begin{array}{c}(\delta : C) \in \Gamma \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : C \rightsquigarrow \delta}\ \textsc{D-var}$$

The first and second hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \rightsquigarrow \delta_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \rightsquigarrow \delta_2$$

From the convention regarding namespace translations, explained in Section A.7.2, it follows directly that $\delta_1 = \delta_2 = \delta$.

Case rule D-CON

$$\textsc{D-con}\quad \dfrac{\begin{array}{c}\Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e, \Sigma_2 \\ (m : TC\,a : \sigma_m) \in \Gamma_C \\ \vdash_{ctx} \Sigma; \Gamma_C; \Gamma \\ \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma'_i}^{\,i} \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \rightsquigarrow e\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \rightsquigarrow \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\}}$$

The hypotheses of the theorem are

$$\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \rightsquigarrow \Lambda \overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\} \tag{C.1031}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q \rightsquigarrow \Lambda \overline{a}_j.\lambda\,\overline{\delta'_i : \sigma''_i}^{\,i}.\{m' = e'\} \tag{C.1032}$$

The 5th premise of the two instantiations of rule rule D-CON, are

$$\Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \rightsquigarrow e \tag{C.1033}$$

$$\text{and } \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}'_i : \overline{C}'_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \rightsquigarrow e' \tag{C.1034}$$

and the 1st premise of the two rule D-CON rules are

$$(D : \forall \overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda \overline{a}_j.\lambda \overline{\delta}_i : \overline{C}_i.e \in \Sigma$$

$$(D : \forall \overline{a}_j.\overline{C}'_i \Rightarrow TC\,\sigma_q).m' \mapsto \Lambda \overline{a}_j.\lambda \overline{\delta}'_i : \overline{C}'_i.e \in \Sigma$$

However, a valid type class instance environment, like Σ in this case, contains a unique entry for each constructor D. From the two above premises and this uniqueness property, we have that

$$\begin{gathered}\overline{C}'_i = \overline{C}_i \;\text{ and }\; \overline{\delta}'_i = \overline{\delta}_i, \;\text{ for all } i \\ \text{and } m' = m\end{gathered} \tag{C.1035}$$

By applying these equations to Equations C.1033 and C.1034, their typing environment becomes identical. We can, now, use Theorem 67 on these two equations, from which we get $e = e'$. Also, from the namespace-translation convention, we get $m' = m$ and $\overline{\delta}'_i = \overline{\delta}_i$. From Equations C.1035 and the 4th set of premises, we get

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma'_i}^{\,i}$$

$$\text{and } \overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma''_i}^{\,i}$$

By passing the two above in Lemma 132, we get $\sigma''_i = \sigma'_i$, for all i.

If we rewrite Equations C.1031 and C.1032 with the equations obtained so far, we have

$$\Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow T C\, \sigma_q \rightsquigarrow \Lambda \overline{a}_j . \lambda\, \overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_d D : \forall \overline{a}_j . \overline{C}_i \Rightarrow T C\, \sigma_q \rightsquigarrow \Lambda \overline{a}_j . \lambda\, \overline{\delta_i : \sigma'_i}^{\,i} . \{m = e\}$$

The goal thus follows directly from this result.

$$\boxed{\textbf{Case rule } \text{D-DABS}} \qquad \begin{array}{l}\text{D-DABS}\\ \dfrac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_C C_1 \rightsquigarrow \sigma_1\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d \lambda \delta : C_1 . d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda \delta : \sigma_1 . e}\end{array}$$

The hypotheses of the theorem are

$$\Sigma; \Gamma_C; \Gamma \vdash_d \lambda \delta : C_1 . d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda \delta : \sigma_1 . e \tag{C.1036}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d \lambda \delta : C_1 . d : C_1 \Rightarrow C_2 \rightsquigarrow \lambda \delta : \sigma'_1 . e' \tag{C.1037}$$

The 1st premise of the two instantiations of rule rule D-DABS, are

$$\Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e \tag{C.1038}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma, \delta : C_1 \vdash_d d : C_2 \rightsquigarrow e' \tag{C.1039}$$

Applying the induction hypothesis to Equations C.1038 and C.1039 teaches us that

$$e = e'$$

By passing the 2nd set of premises in Lemma 132, we get $\sigma_1 = \sigma'_1$. The goal follows from the equations obtained so far.

$$\boxed{\textbf{Case rule } \text{D-DAPP}} \qquad \begin{array}{l}\text{D-DAPP}\\ \dfrac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e_1\, e_2}\end{array}$$

The hypotheses of the theorem are

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e_1\, e_2 \tag{C.1040}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e'_1\, e'_2 \tag{C.1041}$$

By case analysis on these equations (rule D-DAPP) we get

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \tag{C.1042}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1' \tag{C.1043}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2 \tag{C.1044}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d_2 : C_1 \rightsquigarrow e_2' \tag{C.1045}$$

Applying the induction hypothesis to both sets of equations teaches us that

$$e_1 = e_1' \text{ and } e_2 = e_2'$$

The goal follows directly from this result.

Case rule D-TYABS

$$\frac{\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e}{\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e}\ \textsc{D-tyabs}$$

The hypotheses of the theorem are

$$\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e \tag{C.1046}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e' \tag{C.1047}$$

By case analysis on these equations (rule D-TABS) we get

$$\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e \tag{C.1048}$$

$$\Sigma; \Gamma_C; \Gamma, a \vdash_d d : C \rightsquigarrow e' \tag{C.1049}$$

Applying the induction hypothesis on these equations teaches us that

$$e = e'$$

The goal follows directly from this result.

Case rule D-TYAPP

$$\frac{\begin{array}{c}\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d d\,\sigma : [\sigma/a]C \rightsquigarrow e\,\sigma}\ \textsc{D-tyapp}$$

The hypotheses of the theorem are

$$\Sigma; \Gamma_C; \Gamma \vdash_d d\,\sigma : [\sigma/a]C \rightsquigarrow e\,\sigma \tag{C.1050}$$

$$\Sigma; \Gamma_C; \Gamma \vdash_d d\,\sigma : [\sigma/a]C \rightsquigarrow e'\,\sigma' \tag{C.1051}$$

The 1st premise of the two instantiations of rule rule D-DABS, are

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow e \tag{C.1052}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_d d : \forall a.C \rightsquigarrow e' \tag{C.1053}$$

Applying the induction hypothesis to Equations C.1052 and C.1053 teaches us that

$$e = e'$$

By passing the 2nd set of premises in Lemma 133, we get $\sigma = \sigma'$. The goal follows from the equations obtained so far. □

Theorem 67 (Deterministic Term Elaboration).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_1$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_2$*, then* $e_1 = e_2$*.*

Proof. This theorem is proved mutually with Theorem 66. The proof follows structural induction on both hypotheses.

$\boxed{\textbf{Case rule } \textsc{iTm-true}}$
The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{True} : \mathit{Bool} \rightsquigarrow e_2$$

From rule rule iTm-true, it must hold that $e_1 = e_2 = \mathit{True}$.
$\boxed{\textbf{Case rule } \textsc{iTm-false}}$
The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \mathit{False} : \mathit{Bool} \rightsquigarrow e_2$$

From rule rule iTm-false, it must hold that $e_1 = e_2 = \mathit{False}$.
$\boxed{\textbf{Case rule } \textsc{iTm-var}}$
The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma \rightsquigarrow x_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} x : \sigma \rightsquigarrow x_2$$

From rule rule iTm-var, it must hold that $x_1 = x_2 = x$, where x is a target-term-variable with the same identifier as x.
$\boxed{\textbf{Case rule } \textsc{iTm-let}}$
The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } \ x : \sigma_1 = e_1 \textbf{ in } \ e_2 : \sigma_2 \rightsquigarrow \textbf{let } \ x_0 : \sigma_1 = e_1 \textbf{ in } \ e_2 \tag{C.1054}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \textbf{let } \ x : \sigma_1 = e_1 \textbf{ in } \ e_2 : \sigma_2 \rightsquigarrow \textbf{let } \ x_0' : \sigma_1' = e_1' \textbf{ in } \ e_2' \tag{C.1055}$$

From our convention regarding translation of identifiers, we have

$$x_0 = x'_0 = x \tag{C.1056}$$

where x is a target-term variable with the same identifier as x.

The first premise of the two rule iTm-let instantiations above, are

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e'_1$$

By induction hypothesis, we get

$$e_1 = e'_1 \tag{C.1057}$$

The 3rd premise of the two instantiations in Equations C.1054 and C.1055 of rule iTm-let are:

$$\Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma'_1$$

By uniqueness (Lemma 133), we get

$$\sigma_1 = \sigma'_1 \tag{C.1058}$$

The 2nd premise of the two instantiations in Equations C.1054 and C.1055 of rule iTm-let are:

$$\Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e_2$$

$$\text{and } \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \sigma_2 \rightsquigarrow e'_2$$

By induction hypothesis, we get

$$e_2 = e'_2 \tag{C.1059}$$

From Equations C.1056, C.1057, C.1058 and C.1059, we obtain

$$\mathbf{let}\ \ x_0 : \sigma_1 = e_1\ \mathbf{in}\ \ e_2 = \mathbf{let}\ \ x'_0 : \sigma'_1 = e'_1\ \mathbf{in}\ \ e'_2$$

Case rule iTm-method

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma_q/a]\sigma_m \rightsquigarrow e.m_0 \tag{C.1060}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} d.m : [\sigma_q/a]\sigma_m \rightsquigarrow e'.m'_0 \tag{C.1061}$$

By our convention for dictionary labels, we have

$$m_0 = m'_0 = m$$

where m is a record field with the same identifier as class method m.

The first premise of the two rule iTm-method instantiations in Equations C.1060 and C.1061 are:

$$\Sigma; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma_q \rightsquigarrow e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_d d : TC\,\sigma_q \rightsquigarrow e'$$

By Theorem 66, we have

$$e = e'$$

Then, $e.m_0 = e'.m'_0$.

Case rule iTm-arrI

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x_0 : \sigma_1.e \tag{C.1062}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x'_0 : \sigma'_1.e' \tag{C.1063}$$

From our identifiers' translation, it is implied that

$$x_0 = x'_0 = x \tag{C.1064}$$

where x is the target-term variable with the same identifier as x.

The 2nd premise of the two rule iTm-arrI instantiations above are:

$$\Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1$$

$$\text{and } \Gamma_C; \Gamma \vdash_{ty} \sigma_1 \rightsquigarrow \sigma'_1$$

By uniqueness (Lemma 133), we have

$$\sigma_1 = \sigma'_1 \tag{C.1065}$$

The first premise of the two rule iTm-arrI instantiations in Equations C.1062 and C.1063 are:

$$\Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \rightsquigarrow e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma, x : \sigma_1 \vdash_{tm} e : \sigma_2 \rightsquigarrow e'$$

By the induction hypothesis, we get

$$e = e' \tag{C.1066}$$

From Equations C.1064, C.1065 and C.1066, we obtain

$$\lambda x_0 : \sigma_1.e = \lambda x_0' : \sigma_1'.e'$$

Case rule iTm-arrE

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2 \tag{C.1067}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1'\, e_2' \tag{C.1068}$$

The first premise of the above two instantiated rule iTm-arrE rules are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e_1 : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow e_1'$$

By induction hypothesis, we have

$$e_1 = e_1' \tag{C.1069}$$

Similarly, from the second premise of the two rule iTm-arrE rules, and the induction hypothesis, we get

$$e_2 = e_2' \tag{C.1070}$$

Then, we obtain

$$e_1\, e_2 = e_1'\, e_2'$$

by Equations C.1069 and C.1070.

Case rule iTm-constrI

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma \rightsquigarrow \lambda\delta_0 : \sigma.e \tag{C.1071}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma \rightsquigarrow \lambda\delta_0' : \sigma'.e' \tag{C.1072}$$

From our identifiers' translation, it is implied that

$$\delta_0 = \delta_0' = \delta \tag{C.1073}$$

where δ is the target-term variable with the same identifier as the dictionary variable δ.

The 2^{nd} premise of the two rule iTm-constrI instantiations above are:

$$\Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma$$

$$\text{and } \Gamma_C; \Gamma \vdash_C C \rightsquigarrow \sigma'$$

By uniqueness (Lemma 132), we have

$$\sigma = \sigma' \tag{C.1074}$$

The first premise of the two rule iTm-constrI instantiations in Equations C.1071 and C.1072 are:

$$\Sigma; \Gamma_C; \Gamma, \delta : C \vdash_{tm} e : \sigma \rightsquigarrow e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma, \delta : C \vdash_{tm} e : \sigma \rightsquigarrow e'$$

By the induction hypothesis, we get

$$e = e' \tag{C.1075}$$

From Equations C.1073, C.1074 and C.1075, we obtain

$$\lambda\delta_0 : \sigma.e = \lambda\delta_0' : \sigma'.e'$$

Case rule iTm-constrE

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1\, e_2 \tag{C.1076}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, d : \sigma \rightsquigarrow e_1'\, e_2' \tag{C.1077}$$

The first premise of the above two instantiated rule iTm-constrE rules are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : C \Rightarrow \sigma \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e : C \Rightarrow \sigma \rightsquigarrow e_1'$$

By induction hypothesis, we have

$$e_1 = e_1' \tag{C.1078}$$

Similarly, from the second premise of the two rule iTm-constrE rules, and the induction hypothesis, we get

$$e_2 = e_2' \tag{C.1079}$$

Then, we obtain

$$e_1\, e_2 = e_1'\, e_2'$$

by Equations C.1078 and C.1079.

Case rule ITM-FORALLI

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma \rightsquigarrow \Lambda a_0.e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} \Lambda a.e : \forall a.\sigma \rightsquigarrow \Lambda a_0'.e'$$

It is implied by our identifiers' translation convention, that

$$a_0 = a_0' = a \tag{C.1080}$$

where a is the target-type variable with the same identifier as the a.

The premise of the two rule ITM-FORALLI instantiations above are:

$$\Sigma; \Gamma_C; \Gamma, a \vdash_{tm} e : \sigma \rightsquigarrow e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma, a \vdash_{tm} e : \sigma \rightsquigarrow e'$$

By the induction hypothesis, we have

$$e = e' \tag{C.1081}$$

Equations C.1080 and C.1081 result in

$$\Lambda a_0.e = \Lambda a_0'.e'$$

Case rule ITM-FORALLE

The two hypotheses of the theorem are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, \sigma : [\sigma/a]\sigma' \rightsquigarrow e\, \sigma \tag{C.1082}$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e\, \sigma : [\sigma/a]\sigma' \rightsquigarrow e'\, \sigma' \tag{C.1083}$$

The first premise of the above two instantiations of rule rule ITM-FORALLE are:

$$\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \rightsquigarrow e$$

$$\text{and } \Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \forall a.\sigma' \rightsquigarrow e'$$

From the induction hypothesis, we get

$$e = e' \tag{C.1084}$$

The second premise of the two instantiations of rule rule ITM-FORALLE in Equations C.1082 and C.1083 are:

$$\Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma \text{ and } \Gamma_C; \Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma'$$

Applying Lemma 133 on these equations gives

$$\sigma = \sigma' \tag{C.1085}$$

From Equations C.1084 and C.1085 we have

$$e\,\sigma = e'\,\sigma'$$

□

Theorem 68 (Deterministic Context Elaboration)**.**
If $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M_1$
and $M : (\Sigma; \Gamma_C; \Gamma \Rightarrow \sigma) \mapsto (\Sigma; \Gamma_C; \Gamma' \Rightarrow \sigma') \rightsquigarrow M_2$*, then* $M_1 = M_2$*.*

Proof. This proof proceeds by straightforward structural induction on both hypotheses, in combination with Lemmas 132 and 133 and Theorems 66 and 67. □

C.8.4 Semantic Preservation

Theorem 69 (Dictionary Semantic Preservation)**.**
If $d \longrightarrow d'$

$$\textit{and } \Sigma; \Gamma_C; \bullet \vdash_d d : C \rightsquigarrow e \tag{C.1086}$$

$$\textit{and } \Sigma; \Gamma_C; \bullet \vdash_d d' : C \rightsquigarrow e' \tag{C.1087}$$

for some C*,* e *and* e'*,*

then, there is an e_y *such that*

$e \longrightarrow^* e_y$ *and* $e' \longrightarrow^* e_y$*.*

Proof. This proof proceeds by induction on the first hypothesis.

$$\boxed{\textbf{Case rule } \textsc{iDictEval-app}} \qquad \frac{d_1 \longrightarrow d_1'}{d_1\, d_2 \longrightarrow d_1'\, d_2}\ \textsc{iDictEval-app}$$

By inversion on Hypotheses C.1086 and C.1087 (rule D-dApp), we get

$$\Sigma; \Gamma_C; \bullet \vdash_d d_1\, d_2 : C_2 \rightsquigarrow e_1\, e_2$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d_1'\, d_2 : C_2 \rightsquigarrow e_1'\, e_2$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d_1 : C_1 \Rightarrow C_2 \rightsquigarrow e_1 \tag{C.1088}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d_1' : C_1 \Rightarrow C_2 \rightsquigarrow e_1' \tag{C.1089}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d_2 : C_1 \rightsquigarrow e_2$$

for some C_1, C_2, e_1, e_1' and e_2. Applying the induction hypothesis to Equations C.1088 and C.1089, together with the premise of rule iDictEval-app, gives us

$$e_1 \longrightarrow^* e_k$$

$$e_1' \longrightarrow^* e_k$$

for some e_k. By Lemma 144, in combination with Theorem 65, we have that

$$e_1\, e_2 \longrightarrow^* e_k\, e_2$$

$$e_1'\, e_2 \longrightarrow^* e_k\, e_2$$

The goal thus follows from this result, by taking $e_y = e_k\, e_2$.

$$\boxed{\textbf{Case rule } \textsc{iDictEval-appAbs}} \qquad \frac{}{(\lambda\delta : C.d_1)\, d_2 \longrightarrow [d_2/\delta]d_1}\ \textsc{iDictEval-appAbs}$$

By repeated inversion on Hypotheses C.1086 and C.1087 (rule D-dApp and rule D-dAbs), we get

$$\Sigma; \Gamma_C; \bullet \vdash_d (\lambda\delta : C_1.d_1)\, d_2 : C_2 \rightsquigarrow (\lambda\delta : \sigma_1.e_1)\, e_2$$

$$\Sigma; \Gamma_C; \bullet \vdash_d [d_2/\delta]d_1 : C_2 \rightsquigarrow e' \tag{C.1090}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d \lambda\delta : C_1.d_1 : C_1 \Rightarrow C_2 \rightsquigarrow \lambda\delta : \sigma_1.e_1$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d_2 : C_1 \rightsquigarrow e_2 \tag{C.1091}$$

$$\Sigma; \Gamma_C; \bullet, \delta : C_1 \vdash_d d_1 : C_2 \rightsquigarrow e_1 \tag{C.1092}$$

$$\Gamma_C; \bullet \vdash_C C_1 \rightsquigarrow \sigma_1$$

for some C_2, σ_1, e_1 and e_2. We thus need to show that there exists and e_y such that

$$(\lambda\delta : \sigma_1.e_1)\, e_2 \longrightarrow^* e_y \tag{C.1093}$$

$$e' \longrightarrow^* e_y \tag{C.1094}$$

Applying Lemma 91 to Equations C.1091 and C.1092 gives us

$$\Sigma; \Gamma_C; \bullet \vdash_d [d_2/\delta]d_1 : C_2 \rightsquigarrow [e_2/\delta]e_1$$

By applying Theorem 66 to this result, in combination with Equation C.1090, we know that $e' = [e_2/\delta]e_1$.

We thus take $e_y = [e_2/\delta]e_1$. Goal C.1093 follows by rule tEval-AppAbs, and Goal C.1094 follows directly by rule tReduce-stop.

$$\boxed{\textbf{Case rule } \text{iDictEval-tyApp}} \quad \frac{d \longrightarrow d'}{d\,\sigma \longrightarrow d'\,\sigma}\ \text{iDictEval-tyApp}$$

By inversion on Hypotheses C.1086 and C.1087 (rule D-tyApp), we get

$$\Sigma; \Gamma_C; \bullet \vdash_d d\,\sigma : [\sigma/a]C \rightsquigarrow e\,\sigma$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d'\,\sigma : [\sigma/a]C \rightsquigarrow e'\,\sigma$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d : \forall a.C \rightsquigarrow e \tag{C.1095}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d' : \forall a.C \rightsquigarrow e' \tag{C.1096}$$

$$\Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma$$

for some C, e, and σ. Applying the induction hypothesis to Equations C.1095 and C.1096, together with the premise of rule iDictEval-tyApp, gives us

$$e \longrightarrow^* e_k$$

$$e' \longrightarrow^* e_k$$

for some e_k. By Lemma 145, in combination with Theorem 65, we have that

$$e\,\sigma \longrightarrow^* e_k\,\sigma$$

$$e'\,\sigma \longrightarrow^* e_k\,\sigma$$

The goal thus follows from this result, by taking $e_y = e_k\,\sigma$.

Case rule $\textsc{iDictEval-tyAppAbs}$ $$\frac{\textsc{iDictEval-tyAppAbs}}{(\Lambda a.d)\,\sigma \longrightarrow [\sigma/a]d}$$

By repeated inversion on Hypotheses C.1086 and C.1087 (rule D-TYAPP and rule D-TYABS), we get

$$\Sigma; \Gamma_C; \bullet \vdash_d (\Lambda a.d)\,\sigma : [\sigma/a]C \rightsquigarrow (\Lambda a.e)\,\sigma$$

$$\Sigma; \Gamma_C; \bullet \vdash_d [\sigma/a]d : [\sigma/a]C \rightsquigarrow e' \tag{C.1097}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d \Lambda a.d : \forall a.C \rightsquigarrow \Lambda a.e$$

$$\Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma$$

$$\Sigma; \Gamma_C; \bullet, a \vdash_d d : C \rightsquigarrow e \tag{C.1098}$$

$$\Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma \tag{C.1099}$$

for some C, e and σ. We thus need to show that there exists and e_y such that

$$(\Lambda a.e)\,\sigma \longrightarrow^* e_y \tag{C.1100}$$

$$e' \longrightarrow^* e_y \tag{C.1101}$$

Applying Lemma 93 to Equations C.1098 and C.1099 gives us

$$\Sigma; \Gamma_C; \bullet \vdash_d [\sigma/a]d : [\sigma/a]C \rightsquigarrow [\sigma/a]e$$

By applying Theorem 66 to this result, in combination with Equation C.1097, we know that $e' = [\sigma/a]e$.

We thus take $e_y = [\sigma/a]e$. Goal C.1100 follows by rule tEval-TappTabs , and Goal C.1101 follows directly by rule tReduce-stop. □

Theorem 70 (Semantic Preservation)**.**
If $\Sigma \vdash e \longrightarrow e'$

and $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma \rightsquigarrow e$ (C.1102)

and $\Sigma; \Gamma_C; \bullet \vdash_{tm} e' : \sigma \rightsquigarrow e'$ (C.1103)

for some σ, e *and* e',

then, there is an e_y *such that*

$e \longrightarrow^* e_y$ *and* $e' \longrightarrow^* e_y$.

Proof. This proof proceeds by induction on the first hypothesis.

$$\boxed{\textbf{Case rule } \textsc{iEval-app}} \qquad \begin{array}{c}\textsc{iEval-app}\\ \dfrac{\Sigma \vdash e_1 \longrightarrow e_1'}{\Sigma \vdash e_1\, e_2 \longrightarrow e_1'\, e_2}\end{array}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1'\, e_2 : \sigma_2 \rightsquigarrow e_1'\, e_2$$

for some σ_2, e_1, e_1' and e_2.

The last rule of these derivations must be instances of rule iTm-arrE. For Hypothesis C.1102, we have:

$$\begin{array}{c}\textsc{rule iTm-arrE}\\ \dfrac{\begin{array}{c}\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : \sigma_1 \to \sigma_2 \rightsquigarrow e_1\\ \Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : \sigma_1 \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1\, e_2 : \sigma_2 \rightsquigarrow e_1\, e_2}\end{array} \tag{C.1104}$$

and for Hypothesis C.1103, we have:

$$\begin{array}{c}\textsc{rule iTm-arrE}\\ \dfrac{\begin{array}{c}\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1' : \sigma_1' \to \sigma_2 \rightsquigarrow e_1'\\ \Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : \sigma_1' \rightsquigarrow e_2\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1'\, e_2 : \sigma_2 \rightsquigarrow e_1'\, e_2}\end{array} \tag{C.1105}$$

From the second premise of the two above rules and by uniqueness (Lemma 133), we get $\sigma_1 = \sigma_1'$.

The induction hypothesis is:

$$\text{If } \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : \sigma_y \rightsquigarrow e_p$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1' : \sigma_y \rightsquigarrow e_q$$

$$\text{for some } \sigma_y,\ e_p \text{ and } e_q,$$

$$\text{then, there is an } e' \text{ such that}$$

$$e_p \longrightarrow^* e' \text{ and } e_q \longrightarrow e'$$

Then, an appropriate choice for e_y is $e'\, e_2$, since from Lemma 144 (in combination with Theorem 64), we have $e_1\, e_2 \longrightarrow^* e'\, e_2$ and $e_1'\, e_2 \longrightarrow^* e'\, e_2$.

$$\boxed{\textbf{Case rule } \textsc{iEval-appAbs}} \qquad \begin{array}{c}\textsc{iEval-appAbs}\\ \dfrac{}{\Sigma \vdash (\lambda x : \sigma . e_1)\, e_2 \longrightarrow [e_2/x]e_1}\end{array}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} (\lambda x : \sigma . e_1)\, e_2 : \sigma' \rightsquigarrow e_0 \tag{C.1106}$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} [e_2/x]e_1 : \sigma' \rightsquigarrow e_0' \tag{C.1107}$$

for some σ', e_0 and e_0'. We need to show that there exists an e_y such that $e_0 \longrightarrow^* e_y$ and $e_0' \longrightarrow^* e_y$. We do this by showing that $e_0 \longrightarrow^* e_0'$.

By inversion, the last part of Derivation C.1106 must be an instance of rule iTm-arrI directly followed by rule iTm-arrE, as shown below.

$$\cfrac{\text{RULE iTM-ARRE} \quad \cfrac{\text{RULE iTM-ARRI} \quad \begin{array}{c}\Sigma; \Gamma_C; \bullet, x : \sigma \vdash_{tm} e_1 : \sigma' \rightsquigarrow e_1 \\ \Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma . e_1 : \sigma \to \sigma' \rightsquigarrow \lambda x : \sigma . e_1} \quad \Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2}{\Sigma; \Gamma_C; \bullet \vdash_{tm} (\lambda x : \sigma . e_1)\, e_2 : \sigma' \rightsquigarrow (\lambda x : \sigma . e_1)\, e_2}$$

where $e_0 = (\lambda x : \sigma . e_1)\, e_2$. From the above equation, we can use premises

$$\Sigma; \Gamma_C; \bullet, x : \sigma \vdash_{tm} e_1 : \sigma' \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2$$

in Lemma 85, to obtain

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} [e_2/x]e_1 : \sigma' \rightsquigarrow [e_1/x]e_2$$

Then, by uniqueness (Theorem 67 on the latter and on Equation C.1107), we have $e_0' = [e_1/x]e_2$.

We set $e_y = [e_1/x]e_2$, since $(\lambda x : \sigma . e_1)\, e_2 \longrightarrow^* [e_1/x]e_2$, by evaluation rule rule tEval-AppAbs, and $[e_1/x]e_2 \longrightarrow^* [e_1/x]e_2$, by reflexivity of $\longrightarrow^*$.

Case rule iEval-tyApp

$$\text{iEval-tyApp} \quad \frac{\Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\, \sigma \longrightarrow e'\, \sigma}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} e\, \sigma : [\sigma/a]\sigma_p \rightsquigarrow e\, \sigma$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e'\, \sigma : [\sigma/a']\sigma_q \rightsquigarrow e'\, \sigma$$

for some a, a', σ_p, σ_q, e, e' and σ.

The last rule of both derivations above must be instances of rule iTm-forallE. For the first, we have:

$$\text{rule iTm-forallE}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \forall a . \sigma_p \rightsquigarrow e \\ \Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} e\, \sigma : [\sigma / a] \sigma_p \rightsquigarrow e\, \sigma} \tag{C.1108}$$

and for the second:

$$\text{rule iTm-forallE}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \bullet \vdash_{tm} e' : \forall a' . \sigma_q \rightsquigarrow e' \\ \Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} e\, \sigma : [\sigma / a'] \sigma_q \rightsquigarrow e'\, \sigma} \tag{C.1109}$$

By applying Theorem 30 on the first premise of Equation C.1108 and on the premise of rule rule iEval-tyApp, we have that $\forall a . \sigma_p = \forall a' . \sigma_q$.

The induction hypothesis is:

$$\text{If } \Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : \sigma_y \rightsquigarrow e_p$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e'_1 : \sigma_y \rightsquigarrow e_q$$

$$\text{for some } \sigma_y,\ e_p \text{ and } e_q,$$

$$\text{then, there is an } e'_y \text{ such that}$$

$$e_p \longrightarrow^* e'_y \text{ and } e_q \longrightarrow e'_y$$

For $\sigma_y = \forall a . \sigma_p$, $e_p = e$ and $e_q = e'$, the two conditions are fulfilled by the first premise of Equations C.1108 and C.1109.

We choose $e_y = e'_y\, \sigma$, because from Lemma 145, we have $e\, \sigma \longrightarrow^* e'_y\, \sigma$ and $e'\, \sigma \longrightarrow^* e'_y\, \sigma$.

Case rule iEval-tyAppAbs $\quad \text{iEval-tyAppAbs}\ \dfrac{}{\Sigma \vdash (\Lambda a . e)\, \sigma \longrightarrow [\sigma / a] e}$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} (\Lambda a . e)\, \sigma : \sigma_0 \rightsquigarrow e_0 \tag{C.1110}$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} [\sigma / a] e : \sigma_0 \rightsquigarrow e'_0 \tag{C.1111}$$

for some σ_0, e_0 and e'_0. We need to show that there exists an e_y such that $e_0 \longrightarrow^* e_y$ and $e'_0 \longrightarrow^* e_y$. We do this by showing that $e_0 \longrightarrow^* e'_0$.

By inversion, the last part of Derivation C.1110 must be an instance of rule iTm-forallI directly followed by rule iTm-forallE, as shown below.

$$\dfrac{\textsc{rule iTm-forallE}\quad \dfrac{\textsc{rule iTm-forallI}\quad \Sigma;\Gamma_C;\bullet,a \vdash_{tm} e : \sigma' \rightsquigarrow e}{\Sigma;\Gamma_C;\bullet \vdash_{tm} \Lambda a.e : \forall a.\sigma' \rightsquigarrow \Lambda a.e} \qquad \Gamma_C;\bullet \vdash_{ty} \sigma \rightsquigarrow \sigma}{\Sigma;\Gamma_C;\bullet \vdash_{tm} (\Lambda a.e)\,\sigma : [\sigma/a]\sigma' \rightsquigarrow (\Lambda a.e)\,\sigma}$$

where $\sigma_0 = [\sigma/a]\sigma'$ and $e_0 = (\Lambda a.e)\,\sigma$. From the above equation, we can use premises

$$\Sigma;\Gamma_C;\bullet,a \vdash_{tm} e : \sigma' \rightsquigarrow e$$

$$\text{and } \Gamma_C;\bullet \vdash_{ty} \sigma \rightsquigarrow \sigma$$

in Lemma 89, to obtain

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} [\sigma/a]e : [\sigma/a]\sigma' \rightsquigarrow [\sigma/a]e$$

Then, by uniqueness (Theorem 67 on the latter and on Equation C.1111), we have $e_0' = [\sigma/a]e$.

We set $e_y = [\sigma/a]e$, since $(\Lambda a.e)\,\sigma \longrightarrow^* [\sigma/a]e$, by evaluation rule rule tEval-TAppAbs, and $[\sigma/a]e \longrightarrow^* [\sigma/a]e$, by reflexivity of $\longrightarrow^*$.

$$\boxed{\textbf{Case rule } \textsc{iEval-DApp}} \qquad \dfrac{\textsc{iEval-DApp}\quad \Sigma \vdash e \longrightarrow e'}{\Sigma \vdash e\,d \longrightarrow e'\,d}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} e\,d : \sigma \rightsquigarrow e_1\,e_2$$

$$\text{and } \Sigma;\Gamma_C;\bullet \vdash_{tm} e'\,d : \sigma \rightsquigarrow e_1'\,e_2$$

for some e_1, e_1' and e_2.

The last rule of these derivations must be instances of rule iTm-constrE. For Hypothesis C.1102, we have:

$$\dfrac{\textsc{rule iTm-constrE}\quad \Sigma;\Gamma_C;\bullet \vdash_{tm} e : C \Rightarrow \sigma \rightsquigarrow e_1 \quad \Sigma;\Gamma_C;\bullet \vdash_d d : C \rightsquigarrow e_2}{\Sigma;\Gamma_C;\bullet \vdash_{tm} e\,d : \sigma \rightsquigarrow e_1\,e_2} \tag{C.1112}$$

and for Hypothesis C.1103, we have:

$$\overset{\textsc{rule iTm-constrE}}{\dfrac{\begin{array}{c}\Sigma;\Gamma_C;\bullet \vdash_{tm} e' : C' \Rightarrow \sigma \leadsto e_1' \\ \Sigma;\Gamma_C;\bullet \vdash_{d} d : C' \leadsto e_2\end{array}}{\Sigma;\Gamma_C;\bullet \vdash_{tm} e_1'\, d : \sigma \leadsto e_1'\, e_2}} \tag{C.1113}$$

From the second premise of the two above rules and by uniqueness, we get $C = C'$.

The induction hypothesis is:

$$\begin{array}{c}\text{If } \Sigma;\Gamma_C;\bullet \vdash_{tm} e : \sigma_y \leadsto e_p \\ \text{and } \Sigma;\Gamma_C;\bullet \vdash_{tm} e' : \sigma_y \leadsto e_q \\ \text{for some } \sigma_y,\ e_p \text{ and } e_q, \\ \text{then, there is an } e_y' \text{ such that} \\ e_p \longrightarrow^* e_y' \text{ and } e_q \longrightarrow e_y'\end{array}$$

Then, the conditions of the above hold from the first premise of Derivations C.1112 and C.1113, where $\sigma_y = C \Rightarrow \sigma$, $e_p = e_1$ and $e_q = e_1'$.

Then, an appropriate choice for e_y is $e_y'\, e_2$, since from Lemma 144, we have $e_1\, e_2 \longrightarrow^* e_y'\, e_2$ and $e_1'\, e_2 \longrightarrow^* e_y'\, e_2$.

Case rule iEval-DAppAbs

$$\overset{\textsc{iEval-DAppAbs}}{\dfrac{}{\Sigma \vdash (\lambda\delta : C.e)\, d \longrightarrow [d/\delta]e}}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma;\Gamma_C;\bullet \vdash_{tm} (\lambda\delta : C.e)\, d : \sigma \leadsto e_0 \tag{C.1114}$$

$$\text{and } \Sigma;\Gamma_C;\bullet \vdash_{tm} [d/\delta]e : \sigma \leadsto e_0' \tag{C.1115}$$

for some σ', e_0 and e_0'. We need to show that there exists an e_y such that $e_0 \longrightarrow^* e_y$ and $e_0' \longrightarrow^* e_y$. We do this by showing that $e_0 \longrightarrow^* e_0'$.

By inversion, the last part of Derivation C.1114 must be an instance of rule iTm-constrI directly followed by rule iTm-constrE, as shown below.

$$\overset{\textsc{rule iTm-constrE}}{\dfrac{\begin{array}{c}\overset{\textsc{rule iTm-constrI}}{\dfrac{\begin{array}{c}\Sigma;\Gamma_C;\bullet,\delta : C \vdash_{tm} e : \sigma \leadsto e_1 \\ \Gamma_C;\bullet \vdash_{ty} \sigma \leadsto \sigma\end{array}}{\Sigma;\Gamma_C;\bullet \vdash_{tm} \lambda\delta : C.e : C \Rightarrow \sigma \leadsto \lambda\delta : \sigma.e_1}} \\ \Sigma;\Gamma_C;\bullet \vdash_{d} d : C \leadsto e_2\end{array}}{\Sigma;\Gamma_C;\bullet \vdash_{tm} (\lambda\delta : C.e)\, d : \sigma \leadsto (\lambda\delta : \sigma.e_1)\, e_2}}$$

where $e_0 = (\lambda\delta : \sigma.e_1)\, e_2$. From the above equation, we can use premises

$$\Sigma; \Gamma_C; \bullet, \delta : C \vdash_{tm} e : \sigma \rightsquigarrow e_1$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_d d : C \ \rightsquigarrow e_2$$

in Lemma 87, to obtain

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} [d/\delta]e : \sigma \rightsquigarrow [e_1/\delta]e_2$$

Then, by uniqueness (Theorem 66 on the latter and on Equation C.1115), we have $e_0' = [e_1/\delta]e_2$.

We set $e_y = [e_1/\delta]e_2$, since $(\lambda\delta : \sigma.e_1)\, e_2 \ \longrightarrow^* \ [e_1/\delta]e_2$, by evaluation rule rule tEval-AppAbs, and $[e_1/\delta]e_2 \longrightarrow^* [e_1/\delta]e_2$, by reflexivity of $\longrightarrow^*$.

Case rule iEval-method

$$\text{iEval-method}\quad \frac{d \longrightarrow d'}{\Sigma \vdash d.m \longrightarrow d'.m}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} d.m : \sigma \rightsquigarrow e \tag{C.1116}$$

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} d'.m : \sigma \rightsquigarrow e' \tag{C.1117}$$

BY inversion on these equations (rule iTm-method) we get

$$\Sigma; \Gamma_C; \bullet \vdash_d d : TC\, \sigma_q \ \rightsquigarrow e_1 \tag{C.1118}$$

$$\Sigma; \Gamma_C; \bullet \vdash_d d' : TC\, \sigma_q \ \rightsquigarrow e_1' \tag{C.1119}$$

$$(m : TC\, a : \sigma_m) \in \Gamma_C$$

$$\sigma = [\sigma_q/a]\sigma_m$$

$$e = e_1.m$$

$$e' = e_1'.m$$

Applying Theorem 65 to Equations C.1118 and C.1119 gives us

$$\Gamma_C; \Gamma \vdash_C TC\, \sigma_q \ \rightsquigarrow \sigma \tag{C.1120}$$

$$\Gamma \vdash_{tm} e_1 : \sigma \tag{C.1121}$$

$$\Gamma \vdash_{tm} e_1' : \sigma \tag{C.1122}$$

Inversion on Equation C.1120 teaches us that $\sigma = \{m : \sigma'\}$ for some σ'.

Applying Theorem 69 to Equations C.1118 and C.1119, in combination with the premise of rule iEval-method, gives us

$$e_1 \longrightarrow^* e_k$$

$$e'_1 \longrightarrow^* e_k$$

for some e_k.

By applying Lemma 146 to these results, we get that

$$e_1.m \longrightarrow^* e_k.m$$

$$e'_1.m \longrightarrow^* e_k.m$$

The goal thus follows from this result by taking $e_y = e_k.m$.

$$\boxed{\textbf{Case rule } \textsc{iEval-methodVal}} \qquad \overset{\textsc{iEval-methodVal}}{\dfrac{(D : C).m \mapsto e \in \Sigma}{\Sigma \vdash (D\,\overline{\sigma}_m\,\overline{d}_n).m \longrightarrow e\,\overline{\sigma}_m\,\overline{d}_n}}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} (D\,\overline{\sigma}_j\,\overline{d}_i).m : \sigma_0 \rightsquigarrow e_0 \tag{C.1123}$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} e\,\overline{\sigma}_j\,\overline{d}_i : \sigma_0 \rightsquigarrow e'_0 \tag{C.1124}$$

for some σ_0, e_0 and e'_0. We need to show that there is a e_y such that $e_0 \longrightarrow^* e_y$ and $e'_0 \longrightarrow^* e_y$.

By inversion on Equation C.1123, we have $e_0 = ((\Lambda\overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e_m\})\,\overline{\sigma}_j\,\overline{e}_i).m$ and $\sigma_0 = [[\overline{\sigma}_j/\overline{a}_j]\sigma_q/a]\sigma_m$, for some σ_m, e_m, a, σ'_i, e_m, σ_j and e_i, such that

$$(m : TC\,a : \sigma_m) \in \Gamma_C$$

$$\Gamma_C; \bullet, a \vdash_{ty} \sigma_m \rightsquigarrow \sigma_m$$

$$(D : \forall\overline{a}_j.\overline{C}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e_m \in \Sigma$$

$$\Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{C}_i \vdash_{tm} e_m : [\sigma_q/a]\sigma_m \rightsquigarrow e_m$$

$$\overline{\Gamma_C; \bullet, \overline{a}_j \vdash_C C_i \rightsquigarrow \sigma'_i}^{\,i}$$

$$\overline{\Gamma_C; \bullet \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j}^{\,j}$$

$$\overline{\Sigma; \Gamma_C; \bullet \vdash_d d_i : [\overline{\sigma}_j/\overline{a}_j]C_i \rightsquigarrow e_i}^{\,i}$$

Because Σ contains a unique method implementation per class instance, we also have

$$e = \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e_m \tag{C.1125}$$

Then, term $e\,\overline{\sigma}_j\,\overline{d}_i$, equal to $(\Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{C}_i.e_m)\,\overline{\sigma}_j\,\overline{d}_i$, is deterministcally elaborated to $(\Lambda\overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.e_m)\,\overline{\sigma}_j\,\overline{e}_i$.

Indeed, $e_y = [\overline{e_i/x_i}^{\,i}][\overline{\sigma_j/a_j}^{\,j}]e_m$ is an appropriate choice, since

$$\begin{aligned}
e_0 &= ((\Lambda\overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e_m\})\,\overline{\sigma}_j\,\overline{e}_i).m \\
&\rightarrow (\lambda\,\overline{\delta_i : [\overline{\sigma_j/a_j}^{\,j}]\sigma'_i}^{\,i}.\{m = [\overline{\sigma_j/a_j}^{\,j}]e_m\}).m \\
&\rightarrow \{m = [\overline{e_i/\delta_i}^{\,i}][\overline{\sigma_j/a_j}^{\,j}]e_m\}.m \\
&\rightarrow [\overline{e_i/\delta_i}^{\,i}][\overline{\sigma_j/a_j}^{\,j}]e_m
\end{aligned}$$

and

$$\begin{aligned}
e'_0 &= (\Lambda\overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.e_m)\,\overline{\sigma}_j\,\overline{e}_i \\
&\rightarrow \lambda\,\overline{\delta_i : [\overline{\sigma_j/a_j}^{\,j}]\sigma'_i}^{\,i}.[\overline{\sigma_j/a_j}^{\,j}]e_m \\
&\rightarrow [\overline{e_i/\delta_i}^{\,i}][\overline{\sigma_j/a_j}^{\,j}]e_m
\end{aligned}$$

$\boxed{\textbf{Case rule } \textsc{iEval-let}}$

$$\textsc{iEval-let}\quad \frac{}{\Sigma \vdash \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e_2 \longrightarrow [e_1/x]e_2}$$

Hypotheses C.1102 and C.1103 of the theorem, adapted to this case, are:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e : \sigma' \rightsquigarrow e_0 \tag{C.1126}$$

$$\text{and } \Sigma; \Gamma_C; \bullet \vdash_{tm} [e_1/x]e_2 : \sigma' \rightsquigarrow e'_0 \tag{C.1127}$$

for some σ', e_0 and e'_0. We need to show that there exists an e_y such that $e_0 \longrightarrow^* e_y$ and $e'_0 \longrightarrow^* e_y$. We do this by showing that $e_0 \longrightarrow^* e'_0$.

By inversion, the last rule used for Derivation C.1126 must be an instance of rule iTm-let.

$$\textsc{rule iTm-constrI}\quad \frac{\begin{array}{c}\Sigma; \Gamma_C; \bullet \vdash_{tm} e_1 : \sigma \rightsquigarrow e_1 \\ \Sigma; \Gamma_C; \bullet, x : \sigma \vdash_{tm} e_2 : \sigma' \rightsquigarrow e_2 \\ \Gamma_C; \bullet \vdash_{ty} \sigma \rightsquigarrow \sigma\end{array}}{\Sigma; \Gamma_C; \bullet \vdash_{tm} \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e : \sigma' \rightsquigarrow \mathbf{let}\ \ x : \sigma = e_1\ \mathbf{in}\ \ e_2}$$

where $e_0 = \textbf{let}\ \ x : \sigma = e_1\ \textbf{in}\ \ e_2$. From the above equation, we can use the first two premises in Lemma 85, to obtain

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} [e_1/x]e_2 : \sigma' \rightsquigarrow [e_1/x]e_2$$

Then, by uniqueness (Theorem 67 on the latter and on Equation C.1127), we have $e_0' = [e_1/x]e_2$.

We set $e_y = [e_1/x]e_2$, since $(\lambda x : \sigma . e_1)\, e_2 \longrightarrow^* [e_1/x]e_2$, by evaluation rule rule TEVAL-APPABS, and $[e_1/x]e_2 \longrightarrow^* [e_1/x]e_2$, by reflexivity of $\longrightarrow^*$.

□

Lemma 147 ($F_{\{\}}$ Preservation of Values)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} v : \sigma \rightsquigarrow e$ *then* e *is a value.*

Proof. By straightforward case analysis on the typing derivation.

Theorem 71 (Value Semantic Preservation)**.**
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Sigma \vdash e \longrightarrow^* v$ *then* $\Sigma; \Gamma_C; \bullet \vdash_{tm} v : \sigma \rightsquigarrow v$ *and* $e \simeq v$*.*

Proof. From the Preservation Theorem 30 and Progress Theorem 32, in combination with the hypothesis, we know that:

$$\Sigma; \Gamma_C; \bullet \vdash_{tm} v : \sigma \rightsquigarrow e'$$

Lemma 147 teaches us that e' is some value v.

The goal follows by repeatedly applying Theorem 70, in combination with the fact that evaluation in both $F_{\mathbf{D}}$ and $F_{\{\}}$ is deterministic (Lemmas 110 and 135).

□

Bibliography

[1] M. Abadi. *Protection in Programming-Language Translations*, pages 19–34. Springer, 1999. ISBN 978-3-540-48749-4.

[2] J. Abella and F. Cazorla. Chapter 9 - harsh computing in the space domain. In A. Vega, P. Bose, and A. Buyuktosunoglu, editors, *Rugged Embedded Systems*, pages 267–293. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-802459-1. doi: https://doi.org/10.1016/B978-0-12-802459-1.00009-9. URL `https://www.sciencedirect.com/science/article/pii/B9780128024591000099`.

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.

[4] A. Ahmed. Logical relations, 2015. `https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html`.

[5] J.-m. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.

[6] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(2):107–152, 2012.

[7] X. Bi, B. C. d. S. Oliveira, and T. Schrijvers. The essence of nested composition. In *ECOOP*, 2018.

[8] X. Bi, N. Xie, B. C. d. S. Oliveira, and T. Schrijvers. Distributive disjoint polymorphism for compositional programming. 2019.

[9] D. Biernacki and P. Polesiuk. Logical relations for coherence of effect subtyping. In *LIPIcs*, 2015.

[10] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *MPC '98*, pages 52–67. Springer, 1998. ISBN 3-540-64591-8.

[11] G.-J. Bottu, G. Karachalias, T. Schrijvers, B. C. d. S. Oliveira, and P. Wadler. Quantified class constraints. In *Haskell 2017*, pages 148–161. ACM, 2017. ISBN 978-1-4503-5182-9.

[12] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[13] J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for haskell. *Journal of Functional Programming*, 26:e15, 2016. doi: 10.1017/S0956796816000150.

[14] I. Cervesato and F. Pfenning. A linear spine calculus. Technical report, Journal of Logic and Computation, 2003.

[15] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, 2005. ISSN 0362-1340.

[16] S. Chauhan, P. P. Kurur, and B. A. Yorgey. How to twist pointers without breaking them. In *Haskell 2016*, pages 51–61. ACM, 2016. ISBN 978-1-4503-4434-0.

[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82*, pages 207–212. ACM, 1982. ISBN 0-89791-065-6.

[18] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. The lean theorem prover. 2015.

[19] D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in agda. In *ICFP '11*, pages 143–155. ACM, 2011. ISBN 978-1-4503-0865-6.

[20] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *POPL '07*, pages 63–70. ACM, 2007. ISBN 1-59593-575-4.

[21] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[22] R. A. Eisenberg. Binding type variables in lambda-expressions. GHC Proposal #155, 2018. URL `https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0155-type-lambda.rst`.

[23] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[24] R. A. Eisenberg, S. Weirich, and H. G. Ahmed. Visible type application. In P. Thiemann, editor, *Programming Languages and Systems*, pages 229–254, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49498-1.

[25] R. A. Eisenberg, J. Breitner, and S. Peyton Jones. Type variables in patterns. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 94–105, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358354. doi: 10.1145/3242744.3242753. URL `https://doi.org/10.1145/3242744.3242753`.

[26] P. Freeman. *PureScript by Example.* Leanpub, 2017. `https://leanpub.com/purescript`.

[27] P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond. Proof relevant corecursive resolution. In *FLOPS 2016*, pages 126–143. Springer, 2016.

[28] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, pages 1–28, Jan. 2019. doi: 10.1145/329031610.1145/3290316. URL `https://hal.inria.fr/hal-01859964`.

[29] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur (Ph.D. thesis).* Université Paris 7, 1972.

[30] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[31] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, 2006. ISSN 0362-1340.

[32] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. ISSN 0164-0925.

[33] R. Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2nd edition, 2016.

[34] R. Harrop. On disjunctions and existential statements in intuitionistic systems of logic. *Mathematische Annalen*, 132(4):347–361, 1956.

[35] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. The mercury language reference manual. Technical report, 1996.

[36] R. Hinze. Perfect trees and bit-reversal permutations. *JFP*, 10(3):305–317, 2000.

[37] R. Hinze. Adjoint folds and unfolds: Or: Scything through the thicket of morphisms. In *MPC'10*, pages 195–228. Springer, 2010. ISBN 3-642-13320-7, 978-3-642-13320-6.

[38] R. Hinze and S. Peyton Jones. Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, pages 227–236. Elsevier Science, 2000.

[39] M. Jaskelioff. Monatron: an extensible monad transformer library. In *IFL'08*, pages 233–248, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24451-3.

[40] M. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Dept. of Computer Science, 1993.

[41] M. P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *ESOP '92*, volume 582 of *LNCS*, pages 287–306. Springer Berlin Heidelberg, 1992.

[42] M. P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

[43] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95*, pages 160–169. ACM, 1995.

[44] M. P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1995. ISBN 0-521-47253-9.

[45] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136. Springer, 1995. ISBN 3-540-59451-5.

[46] M. P. Jones. Type classes with functional dependencies. In *Programming Languages and Systems*, volume 1782 of *LNCS*, pages 230–244. Springer, 2000.

[47] S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proceedings of the 1997 Haskell Workshop.* ACM, 1997.

[48] W. Kahl and J. Scheffczyk. Named instances for haskell type classes. In R. Hinze, editor, *Proc. Haskell Workshop 2001*, volume 59, 2001.

[49] E. A. Kmett. The constraint package, 2017. `https://hackage.haskell.org/package/constraints-0.9.1`.

[50] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, 2003. ISSN 0362-1340.

[51] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. *SIGPLAN Not.*, 40(9):204–215, 2005. ISSN 0362-1340.

[52] L. Lampropoulos and B. C. Pierce. *QuickChick: Property-Based Testing in Coq*, volume 4 of *Software Foundations*. 1st edition, 2018.

[53] D. Le Botlan and D. Rémy. Recasting mlf. *Information and Computation*, 207(6):726–785, 2009. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2008.12.006. URL `https://www.sciencedirect.com/science/article/pii/S0890540109000145`.

[54] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009. ISSN 0304-3975.

[55] S. Marlow (editor). Haskell 2010 language report, 2010.

[56] L. S. Martin Odersky and B. Venners. Implicit conversions and parameters. In *Programming in Scala*, chapter 21. 2008.

[57] T. Mens. On the complexity of software systems. *Computer*, 45(8):79–81, 2012. doi: 10.1109/MC.2012.273. URL `https://doi.org/10.1109/MC.2012.273`.

[58] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. Technical report, 1989.

[59] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(78)90014-4. URL `https://www.sciencedirect.com/science/article/pii/0022000078900144`.

[60] J. G. Morris. A simple semantics for haskell overloading. In W. Swierstra, editor, *Haskell 2014*, pages 107–118. ACM, 2014. ISBN 978-1-4503-3041-1.

[61] J. H. Morris Jr. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[62] T. Mozilla Research. *The Rust Programming Language.* 2017. `https://www.rust-lang.org/en-US/`.

[63] D. Musser and A. Stepanov. Generic programming. 358, 11 1994. doi: 10.1007/3-540-51084-2_2.

[64] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 54–67, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.237729. URL `https://doi.org/10.1145/237721.237729`.

[65] M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, and S. Stucki. Simplicitly: Foundations and Applications of Implicit Function Types. In *POPL '18*, 2017.

[66] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, 2010. ISSN 0362-1340.

[67] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: A new foundation for generic programming. *SIGPLAN Not.*, 47 (6):35–44, 2012. ISSN 0362-1340.

[68] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, pages 56–71, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12251-4.

[69] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming. Cambridge University Press, 2003.

[70] S. Peyton Jones. Simplify subsumption. GHC Proposal #287, 2019. URL `https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0287-simplify-subsumption.rst`.

[71] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340.

[72] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159811. URL `http://doi.acm.org/10.1145/1159803.1159811`.

[73] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *JFP*, 17(1), 2007.

[74] F. Pfenning. Lecture notes on focusing, 2010. `https://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf`.

[75] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

[76] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. ISSN 0164-0925.

[77] G. Plotkin. *Lambda-definability and logical relations.* Edinburgh University, 1973.

[78] B. Ray, D. Posnett, P. Devanbu, and V. Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60 (10):91–100, Sept. 2017. ISSN 0001-0782. doi: 10.1145/3126905. URL `http://doi.acm.org/10.1145/3126905`.

[79] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, page 408–423, Berlin, Heidelberg, 1974. Springer-Verlag. ISBN 3540068597.

[80] J. C. Reynolds. The coherence of languages with intersection types. In *TACS '91*, pages 675–700. Springer-Verlag, 1991. ISBN 3-540-54415-1.

[81] Y.-J. Ringard. Mustard watches: An integrated approach to time and food.

[82] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: Virtualizing the monad stack. *SIGPLAN Not.*, 46(9):32–44, 2011. ISSN 0362-1340.

[83] T. Schrijvers, B. C. Oliveira, P. Wadler, and K. Marntirosian. Cochis: Stable and coherent implicits. *Journal of Functional Programming*, 29:e3, 2019. doi: 10.1017/S0956796818000242.

[84] A. Serrano, J. Hage, D. Vytiniotis, and S. Peyton Jones. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 783–796, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192389. URL `https://doi.org/10.1145/3192366.3192389`.

[85] A. Serrano, J. Hage, S. Peyton Jones, and D. Vytiniotis. A quick look at impredicativity. In *International Conference on Functional Programming (ICFP'20).* ACM, ACM, August 2020. URL `https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/`.

[86] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008. ISBN 978-3-540-71065-3.

[87] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs '08*, pages 278–293. Springer-Verlag, 2008. ISBN 978-3-540-71065-3.

[88] M. Spivey. Faster coroutine pipelines. In *ICFP 2017*, 2017. accepted.

[89] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2-3):85–97, 1985.

[90] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 1967.

[91] M. Sulzmann, G. J. Duck, S. Peyton-Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *JFP*, 17(1):83–129, 2007.

[92] W. W. Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(2):198–212, 1967.

[93] V. Trifonov. Simulating quantified class constraints. In *Haskell '03*, pages 98–102. ACM, 2003. ISBN 1-58113-758-3.

[94] D. Vytiniotis, S. Peyton Jones, and T. Schrijvers. Let should not be generalized. In *TLDI '10*, pages 39–50. ACM, 2010. ISBN 978-1-60558-891-9.

[95] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*. ACM, 1989.

[96] S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[97] L. White, F. Bour, and J. Yallop. Modular implicits. In *ML/OCaml 2014.*, 2014.

[98] T. Winant and D. Devriese. Coherent explicit dictionary application for haskell. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 81–93, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5835-4.

List of publications

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017). Association for Computing Machinery, New York, NY, USA, 148–161. DOI:https://doi.org/10.1145/3122955.3122967

Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of type class resolution. Proc. ACM Program. Lang. 3, ICFP, Article 91 (August 2019), 28 pages. DOI:https://doi.org/10.1145/3341695

Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking stability by being lazy and shallow: lazy and shallow instantiation is user friendly. In Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021). Association for Computing Machinery, New York, NY, USA, 85–97. DOI:https://doi.org/10.1145/3471874.3472985

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
gertjan@bottu.dev
http://www.bottu.dev