KU LEUVEN

Coherence of Type Class Resolution

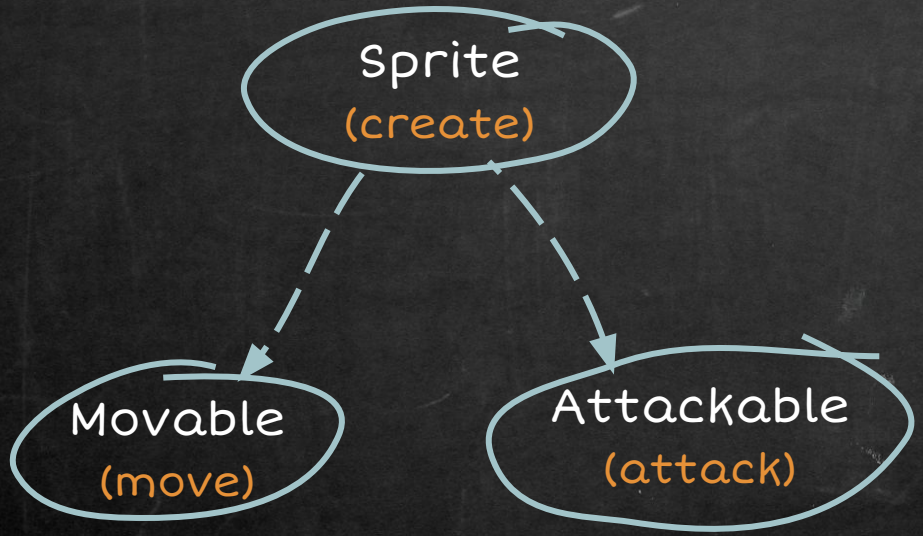
•••

G.J. Bottu ~ N. Xie ~ K. Marntirosian ~ T. Schrijvers

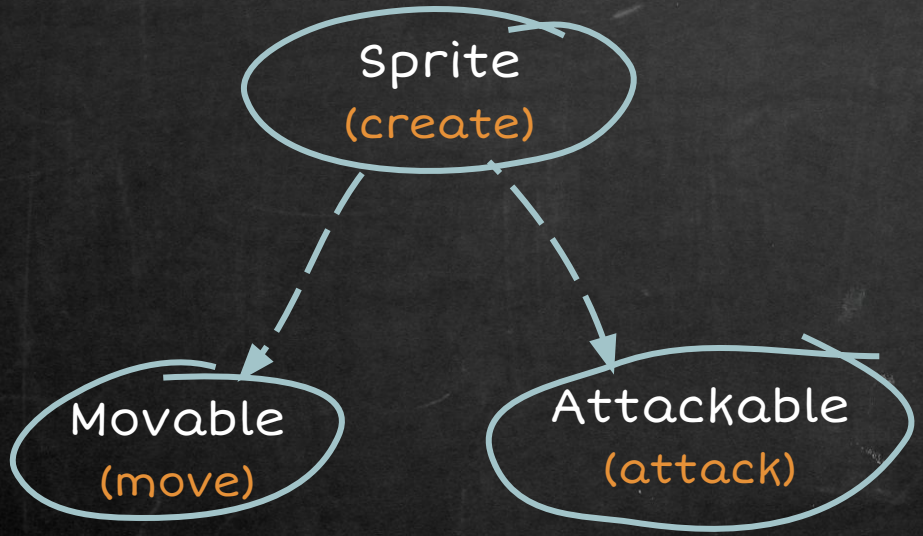
```
data Pacman = MkPacman Location
data Ghost  = MkGhost Location Color
data Candy  = MkCandy Location
```

```
data Pacman = MkPacman Location
data Ghost  = MkGhost Location Color
data Candy  = MkCandy Location


class Sprite a where
  create :: Disp -> a -> Disp

class Sprite a => Movable a where
  move :: Disp -> a -> Dir -> Disp

class Sprite a => Attackable a where
  attack :: Disp -> a -> Disp
```

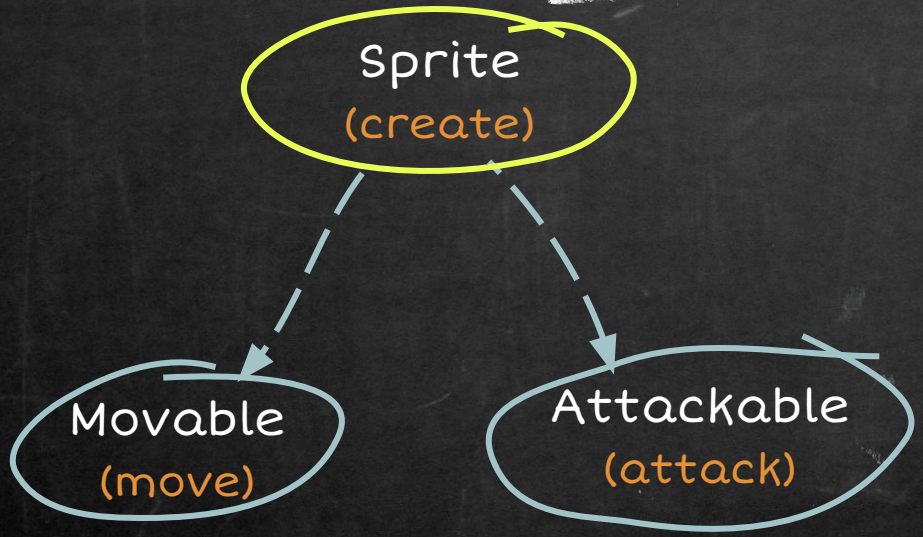
```
data Pacman = MkPacman Location
data Ghost  = MkGhost Location Color
data Candy  = MkCandy Location
```



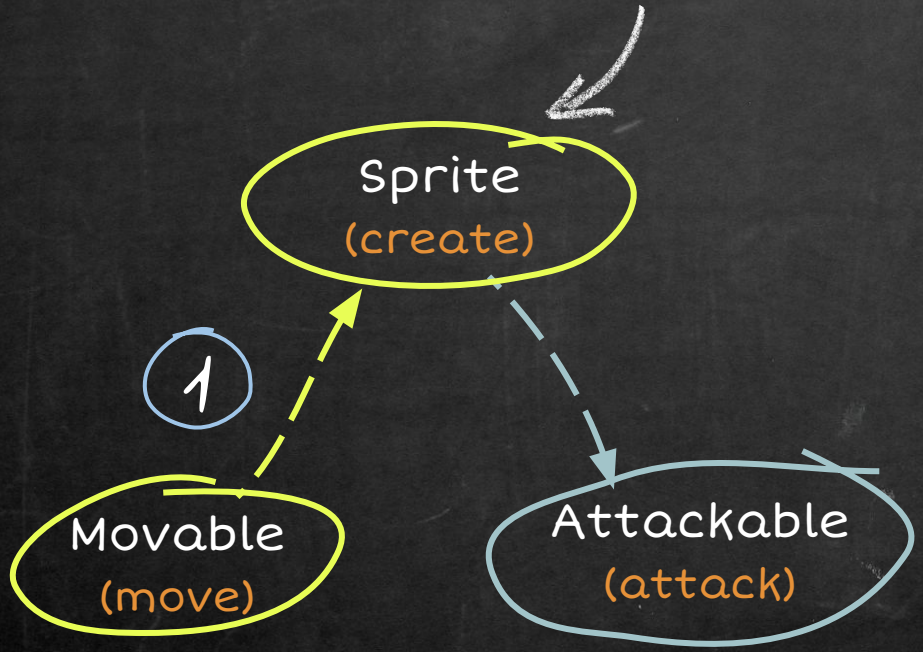
```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```



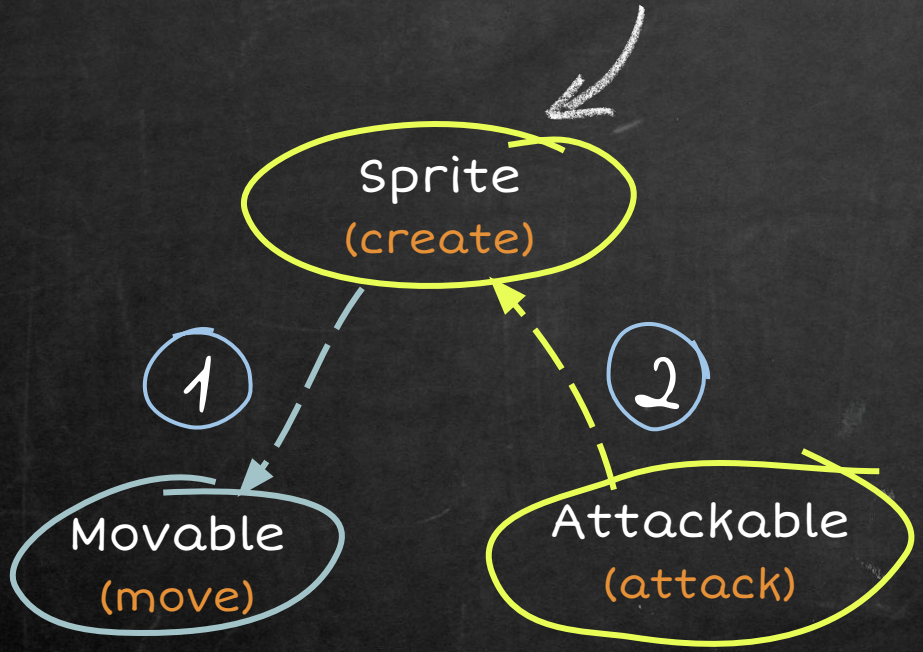
```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```



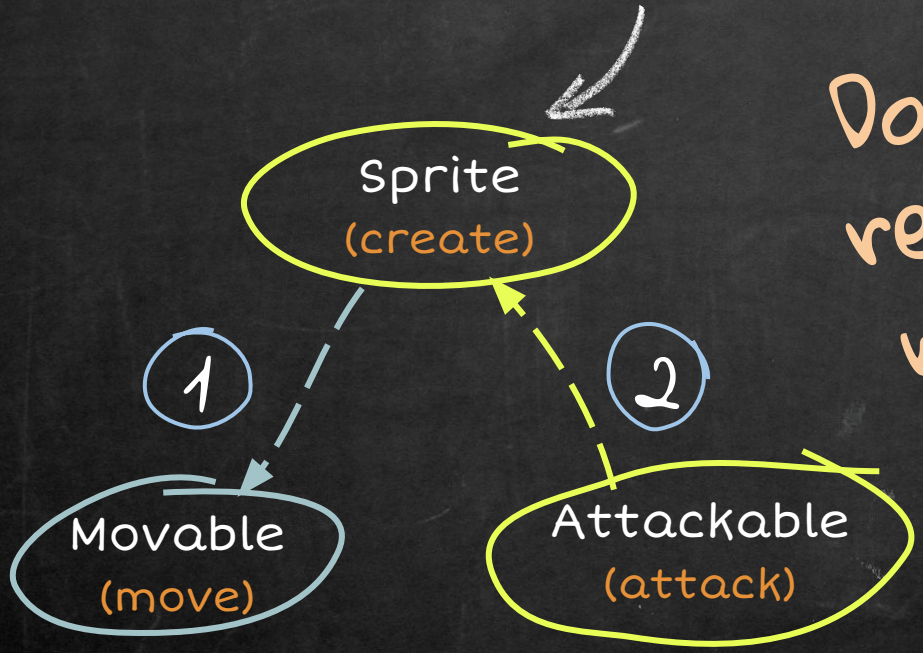
```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```



```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```

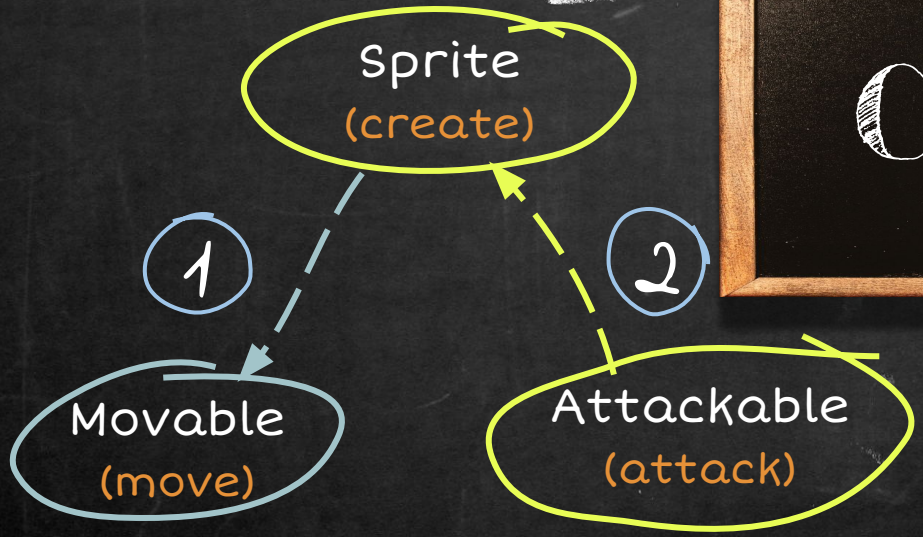


```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```



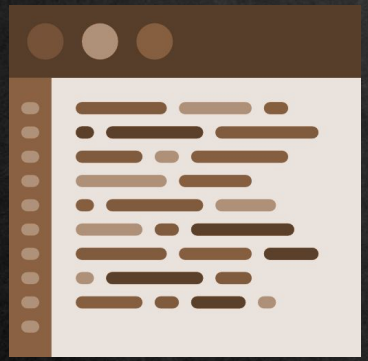
Does the resolution path matter?

```
addEnemies        :: (Movable a, Attackable a)
                  => Disp -> [a] -> Disp
addEnemies d es = foldr create d es
```

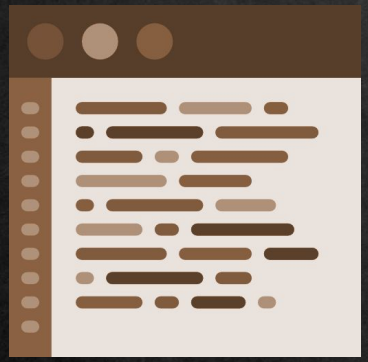


Coherence

Coherence



Coherence



Coherence for Haskell!

Coherence for ~~Haskell~~!
Type Class Resolution

Coherence for ~~Haskell~~!
Type Class Resolution

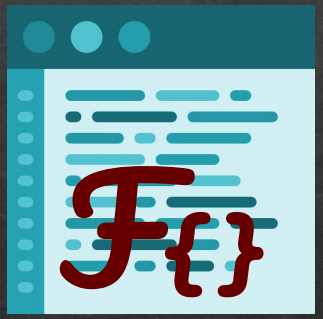
Haskell, Mercury, PureScript, etc.

+ Coherence of
  type class resolution
+ Proof technique

+ Coherence of
  type class resolution
+ Proof technique
+ ICFP paper?

Source

(Type classes)



Target

(Dictionaries)

```
class Show a where
  show :: a -> String
```

```
class Show a where
  show :: a -> String
```

```
instance Show Int where
  show = showInt

instance Show a => Show [a] where
  show = concatMap show
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"
```

```
shout :: {show :: a -> String}
      -> a -> String
shout ds x = ds.show x ++ "!"
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"


shout [1,2,3]
```

```
shout :: {show :: a -> String}
      -> a -> String
shout ds x = ds.show x ++ "!"
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"


shout [1,2,3]
```

```
shout :: {show :: a -> String}
      -> a -> String
shout ds x = ds.show x ++ "!"

shout ??? [1,2,3]
```

Wanted :

Show [Int]

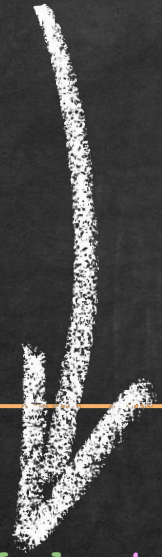
Solved :

```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]



```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

Solved :

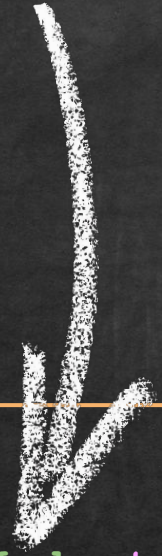
```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]

Solved :

{show = (\ d ->
  concatMap d.show) ??? }



```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]

Show Int

Solved :

```
{show = (\ d ->
  concatMap d.show) ??? }
```

```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]

Show Int

```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

Solved :

```
{show = (\ d ->
  concatMap d.show) ??? }
```

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]

Show Int

```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

Solved :

```
{show = (\ d ->
  concatMap d.show) ??? }
```

{show = showInt}

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

Wanted :

Show [Int]

Show Int

Solved :

{show = (\ d ->
  concatMap d.show) ??? }

{show = showInt}

```
inst Show Int where
  show = showInt
inst Show a => Show [a] where
  show = concatMap show
```

```
shout :: {show :: a -> String}
      -> a -> String

shout ??? [1,2,3]
```

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

```
instance Eq Int where
  (==) = eqInt

instance Ord Int where
  (<=) = leInt
```

```
f :: Ord a => a -> a -> Bool
f x y = ... (==) x y ...
```

```
f :: Ord a => a -> a -> Bool
f x y = ... (==) x y ...
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f do de x y = ... de.(==) x y ...
```

```
f :: Ord a => a -> a -> Bool
f x y = ... (==) x y ...


f 42 42
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f do de x y = ... de.(==) x y ...
```

```
f :: Ord a => a -> a -> Bool
f x y = ... (==) x y ...


f 42 42
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f do de x y = ... de.(==) x y ...

f ??? ???
42 42
```

Wanted :

Ord Int

Solved :

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f ??? ???
42 42
```

Wanted :

Ord Int

Solved :

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f ??? ???
42 42
```

Wanted :

Ord Int

Solved :

{(<=) = leInt}

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f ??? ???
42 42
```

Wanted :

Eq Int

Solved :

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f {(<=) = leInt} ???
42 42
```

Wanted :

Solved :

Eq Int

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f {(<=) = leInt} ???
42 42
```

Wanted :

Eq Int

Solved :

{(==) = eqInt}

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f {(<=) = leInt} ???
42 42
```

Wanted :

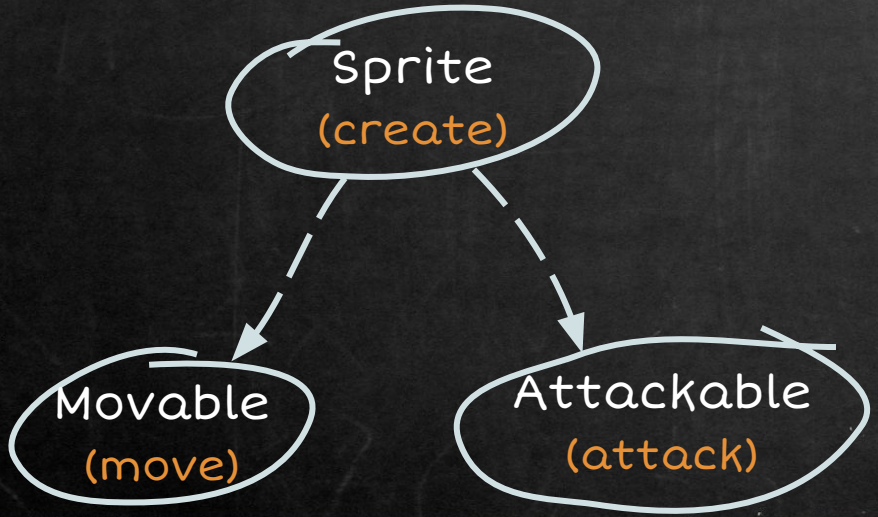
Solved :

```
inst Eq Int where
  (==) = eqInt

inst Ord Int where
  (<=) = leInt
```

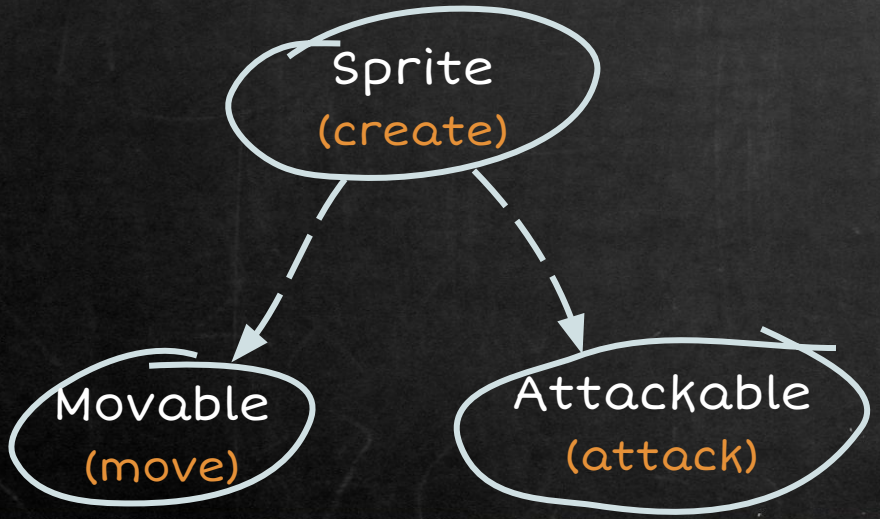
```
f :: {(<=) :: a -> a -> Bool} ->
     {(==) :: a -> a -> Bool} ->
     a -> a -> Bool
f {(<=) = leInt} {(==) = eqInt}  42
42
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

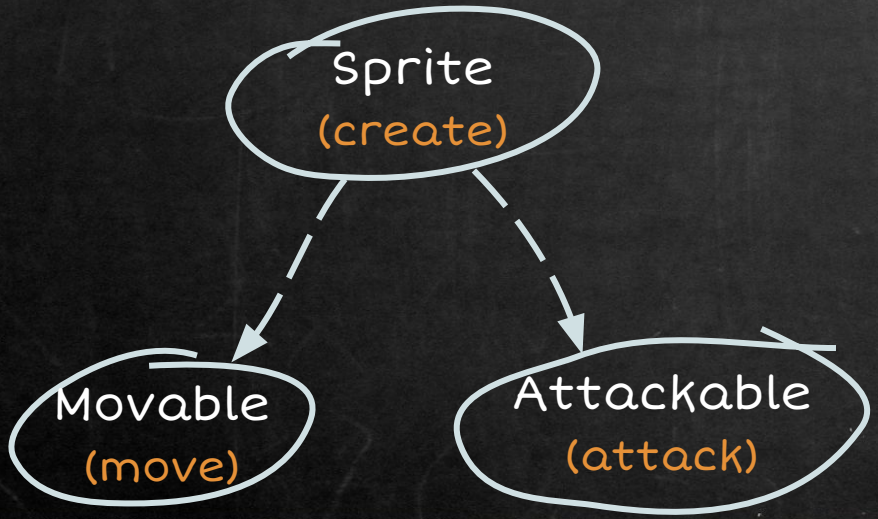


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```



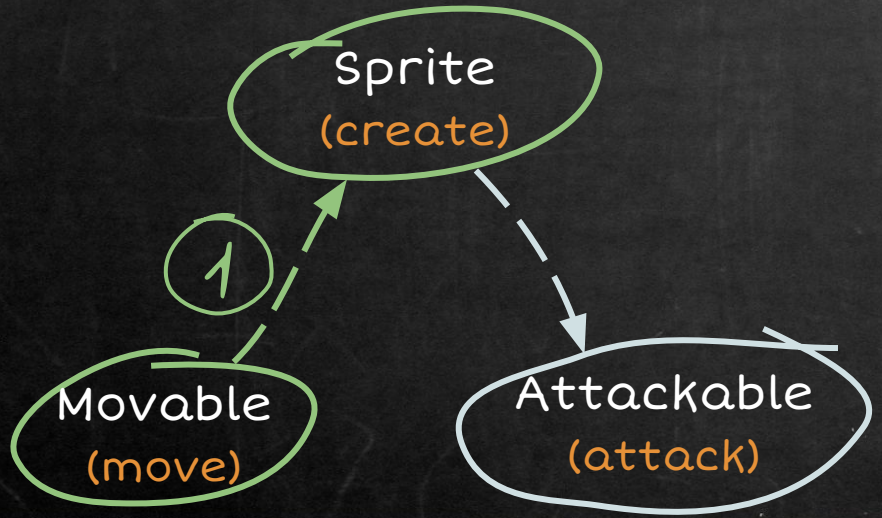
```
addEnemies ::
  {move :: ... } ->
  {create :: ... } ->
  {attack :: ... } ->
  {create :: ... } ->
  Disp -> [a] -> Disp
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```



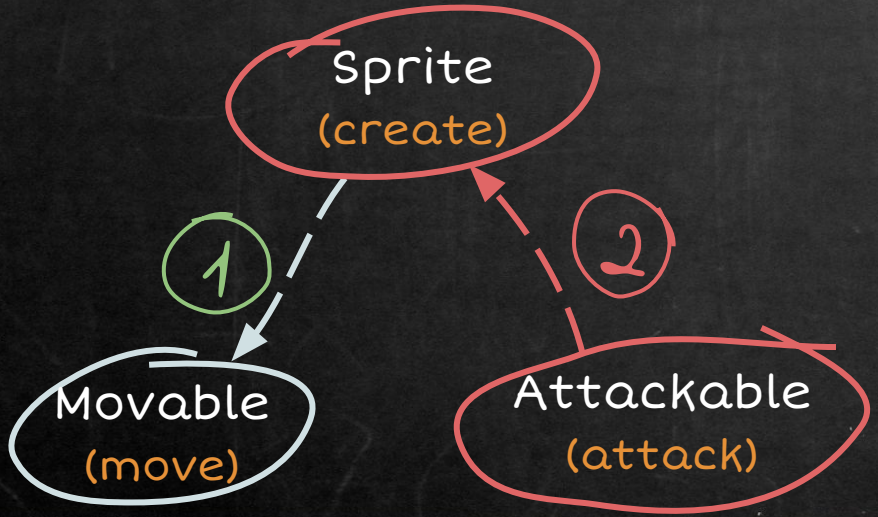
```
addEnemies ::
  {move :: ... } ->
  {create :: ... } ->
  {attack :: ... } ->
  {create :: ... } ->
  Disp -> [a] -> Disp

addEnemies
  recM recS recA recS' d es
  = foldr ???.create d es
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```



```
addEnemies ::
  {move :: ... } ->
  {create :: ... } ->
  {attack :: ... } ->
  {create :: ... } ->
  Disp -> [a] -> Disp

addEnemies
  recM recS recA recS' d es
  = foldr recS.create d es
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```



```
addEnemies ::
  {move :: ... } ->
  {create :: ... } ->
  {attack :: ... } ->
  {create :: ... } ->
  Disp -> [a] -> Disp

addEnemies
  recM recS recA recS' d es
  = foldr recS'.create d es
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

```
addEnemies ::
                    -> 
                  } -> 
                  } -> 
                  } -> 
                    -> Disp
                    s
recM recS recA recS' d es
= foldr recS'.create d es
```

recS ≟ recS'

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy
instance Sprite Ghost
instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy
instance Sprite Ghost
instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

Non-overlapping Instances

Coherence for ~~Haskell~~!
Type Class Resolution + Non-overlapping Instances

Haskell, Mercury, PureScript, etc.

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

How to express this in a proof?

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

Type	Method	Impl
Sprite Pacman	create	...
Sprite Ghost	create	...
Sprite Candy	create	...
Movable Pacman	move	...
Movable Ghost	move	...
Attackable Pacman	attack	...
Attackable Ghost	attack	...

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

	Type	Method	Impl
D1	Sprite Pacman	create	...
D2	Sprite Ghost	create	...
D3	Sprite Candy	create	...
D4	Movable Pacman	move	...
D5	Movable Ghost	move	...
D6	Attackable Pacman	attack	...
D7	Attackable Ghost	attack	...

SOURCE

(Type classes)

TARGET

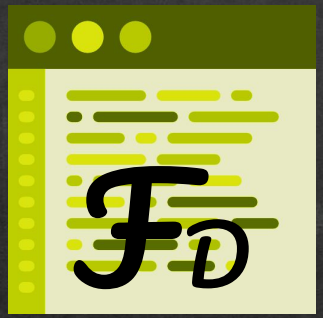
(Dictionaries)

Source

(Type classes)

Target

(Dictionaries)



Alternative

(Implementation table)

```
class Show a where
  show :: a -> String
```

```
inst Show Int where
  show = showInt

inst Show a => Show [a] where
  show = concatMap show
```

```
class Show a where
  show :: a -> String
```

```
inst Show Int where
  show = showInt

inst Show a => Show [a] where
  show = concatMap show
```

	Type	Method	Impl
D1	Show Int	show	...
D2	Show a => Show [a]	show	...

```
shout :: Show a => a -> String
shout x = show x ++ "!"

shout [1,2,3]
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"

shout [1,2,3]
```

```
shout :: Show a -> a -> String
shout ds x = ds ↦ show x ++ "!"
```

```
shout :: Show a => a -> String
shout x = show x ++ "!"

shout [1,2,3]
```

```
shout :: Show a -> a -> String
shout ds x = ds ↦ show x ++ "!"

shout ??? [1,2,3]
```

	Type	Method	Impl
D1	Show Int	show	...
D2	Show a => Show [a]	show	...

```
shout :: Show a -> a -> String
shout ds x = ds↦show x ++ "!"

shout ??? [1,2,3]
```

	Type	Method	Impl
D1	Show Int	show	...
D2	Show a => Show [a]	show	...

```
shout :: Show a -> a -> String
shout ds x = ds↦show x ++ "!"

shout (D2 D1) [1,2,3]
```

	Type	Method	Impl
D1	Show Int	show	...
D2	Show a => Show [a]	show	...

```
shout :: Show a -> a -> String
shout ds x = ds↦show x ++ "!"

shout (D2 D1) [1,2,3]
```

	Type	Method	Impl
D1	Show Int	show	...
D2	Show a => Show [a]	show	...

```
shout :: Show a -> a -> String
shout ds x = ds↦show x ++ "!"

shout (D2 D1) [1,2,3]
```

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

```
instance Sprite Pacman
instance Sprite Ghost
instance Sprite Candy

instance Movable Pacman
instance Movable Ghost

instance Attackable Pacman
instance Attackable Ghost
```

	Type	Method	Impl
D1	Sprite Pacman	create	...
D2	Sprite Ghost	create	...
D3	Sprite Candy	create	...
D4	Movable Pacman	move	...
D5	Movable Ghost	move	...
D6	Attackable Pacman	attack	...
D7	Attackable Ghost	attack	...


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```


```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```


```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr ???↦create d es
```

```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS↦create d es
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```


```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS'↦create d es
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

```
addEnemies disp [gho1, gho2]
```


```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS'↦create d es
```


```
addEnemies ::
  (Movable a, Attackable a) =>
  Disp -> [a] -> Disp
addEnemies d es
  = foldr create d es
```

```
addEnemies disp [ghol, gho2]
```

```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS'↦create d es
```

```
addEnemies ??? ??? ??? ???
  disp [ghol, gho2]
```

	Type	Method	Impl
D1	Sprite Pacman	create	...
D2	Sprite Ghost	create	...
D3	Sprite Candy	create	...
D4	Movable Pacman	move	...
D5	Movable Ghost	move	...
D6	Attackable Pacman	attack	...
D7	Attackable Ghost	attack	...

```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS'↦create d es

addEnemies ??? ??? ??? ???
  disp [ghol, gho2]
```

	Type	Method	Impl
D1	Sprite Pacman	create	...
D2	Sprite Ghost	create	...
D3	Sprite Candy	create	...
D4	Movable Pacman	move	...
D5	Movable Ghost	move	...
D6	Attackable Pacman	attack	...
D7	Attackable Ghost	attack	...

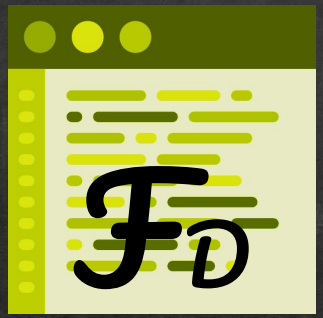
```
addEnemies ::
  Movable a ->
  Sprite a ->
  Attackable a ->
  Sprite a ->
  Disp -> [a] -> Disp

addEnemies
  pM pS pA pS' d es
  = foldr pS'↦create d es

addEnemies D5 D2 D7 D2
  disp [ghol, gho2]
```

Source

(Type classes)

Target

(Dictionaries)



Alternative

(Implementation table)

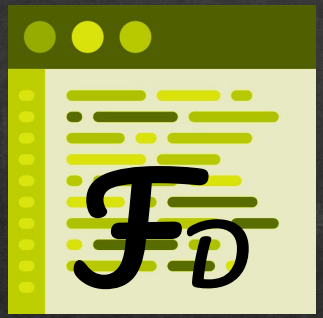
Coherence

Source

(Type classes)

Target

(Dictionaries)



Alternative

(Implementation table)

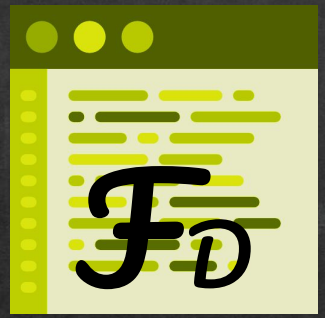
Coherence

Source

(Type classes)

Target

(Dictionaries)



Intermediate

(Implementation table)

Coherence

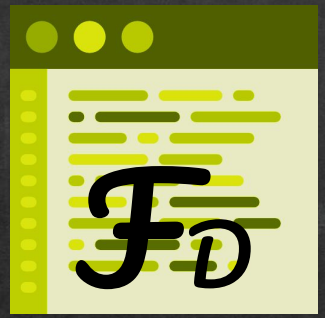
Source

(Type classes)

Target

(Dictionaries)

Coherence



Intermediate

(Implementation table)

Coherence

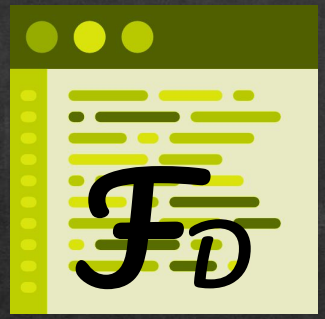
SOURCE

(Type classes)

TARGET

(Dictionaries)

Coherence



Determinism

INTERMEDIATE

(Implementation table)

Coherence

SOURCE

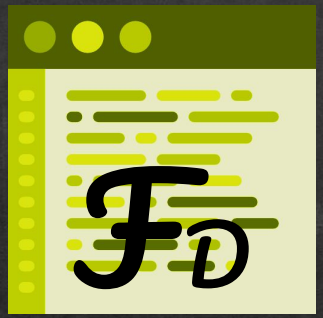
(Type classes)

Equivalence

TARGET

(Dictionaries)

Coherence

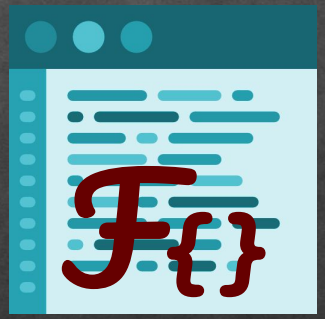


Determinism

INTERMEDIATE

(Implementation table)

Coherence



Source

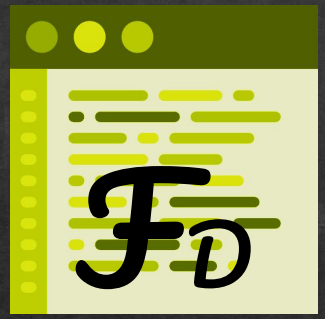
(Type classes)

Equivalence

Target

(Dictionaries)

Coherence



Determinism

Intermediate

(Implementation table)

Coherence

Source

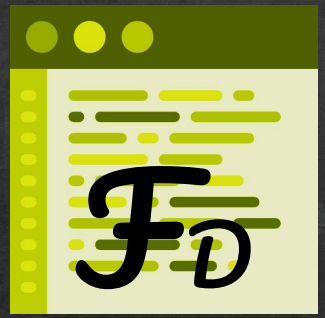
(Type classes)

Equivalence

Target

(Dictionaries)

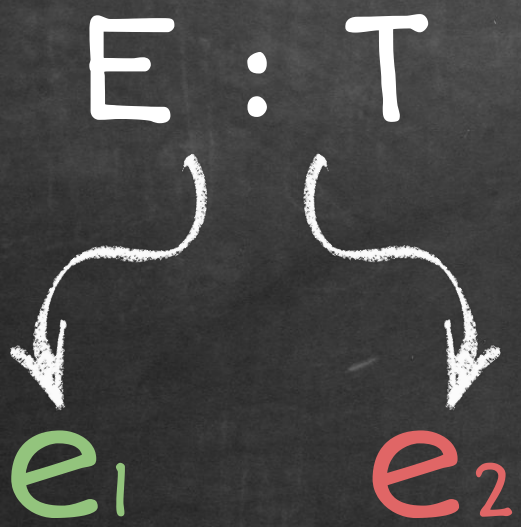
Coherence

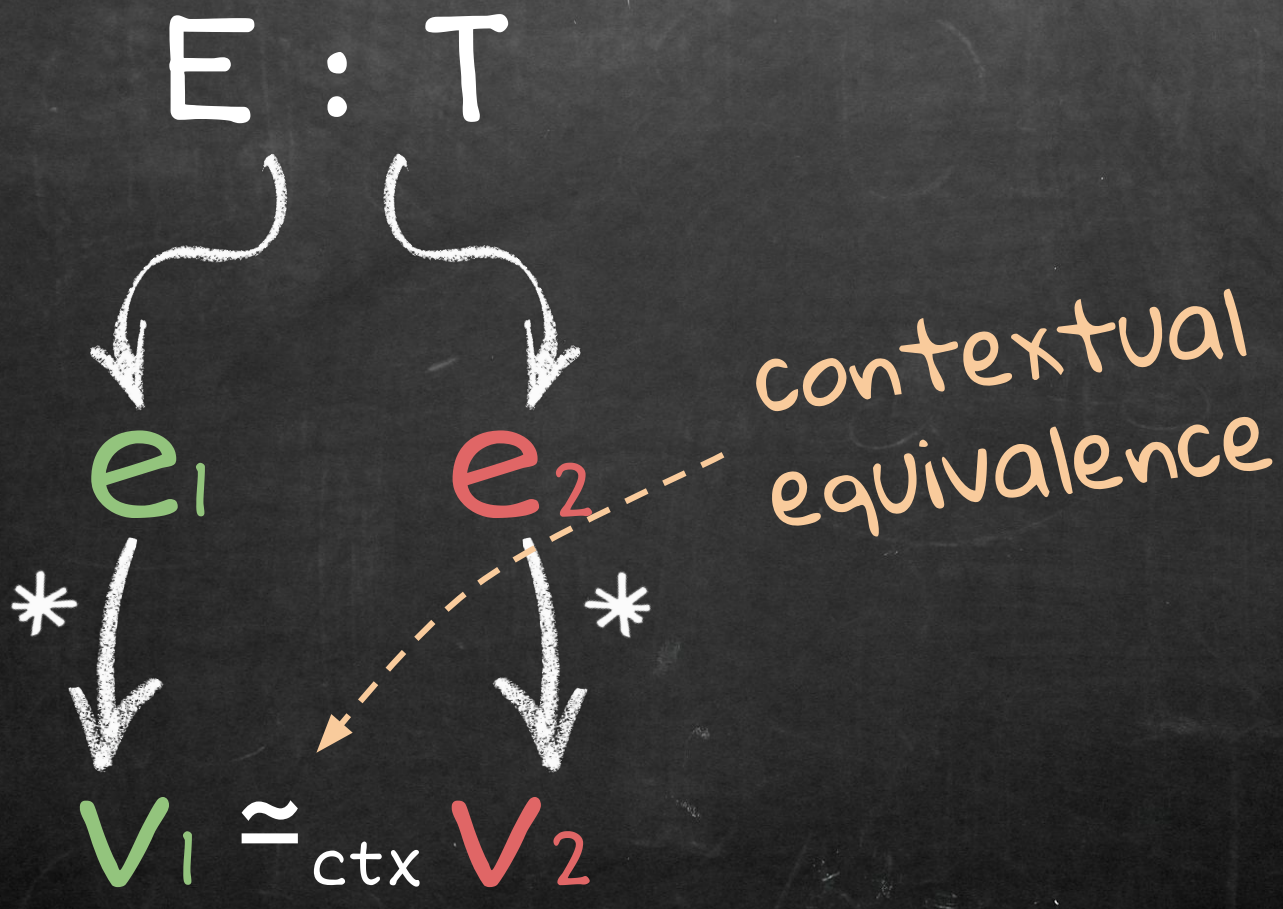


Determinism

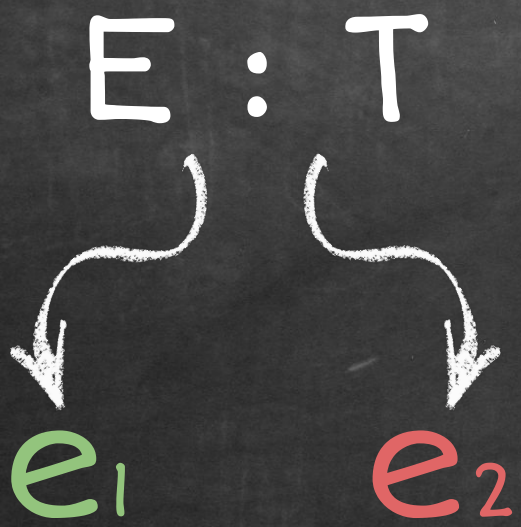
Intermediate

(Implementation table)


E : T
e1
e2

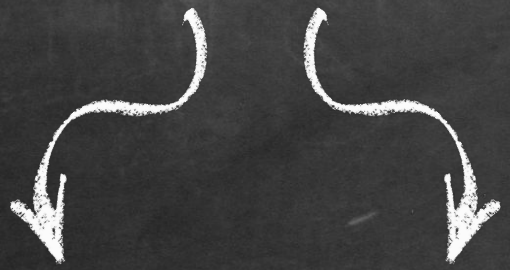

E : T
e1
e2
*
*
contextual
equivalence
V1 ≃ctx V2

Step 1



Step 1

$$E : T$$



$$(\ e_1\ ,\ e_2\ ) \in \mathscr{R}[T]$$

Step 2

$$(e_1, e_2) \in \mathcal{R}[T]$$

Step 2

$$(e_1, e_2) \in \mathcal{R}[T]$$

$e_1 \downarrow^* v_1$ $\quad$ $e_2 \downarrow^* v_2$

$$v_1 \simeq_{ctx} v_2$$

Coherence

SOURCE

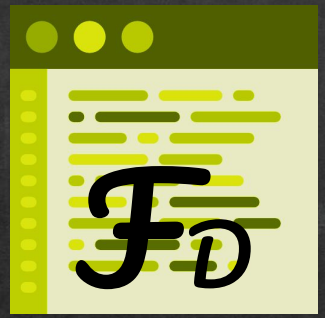
(Type classes)

Equivalence

TARGET

(Dictionaries)

Coherence

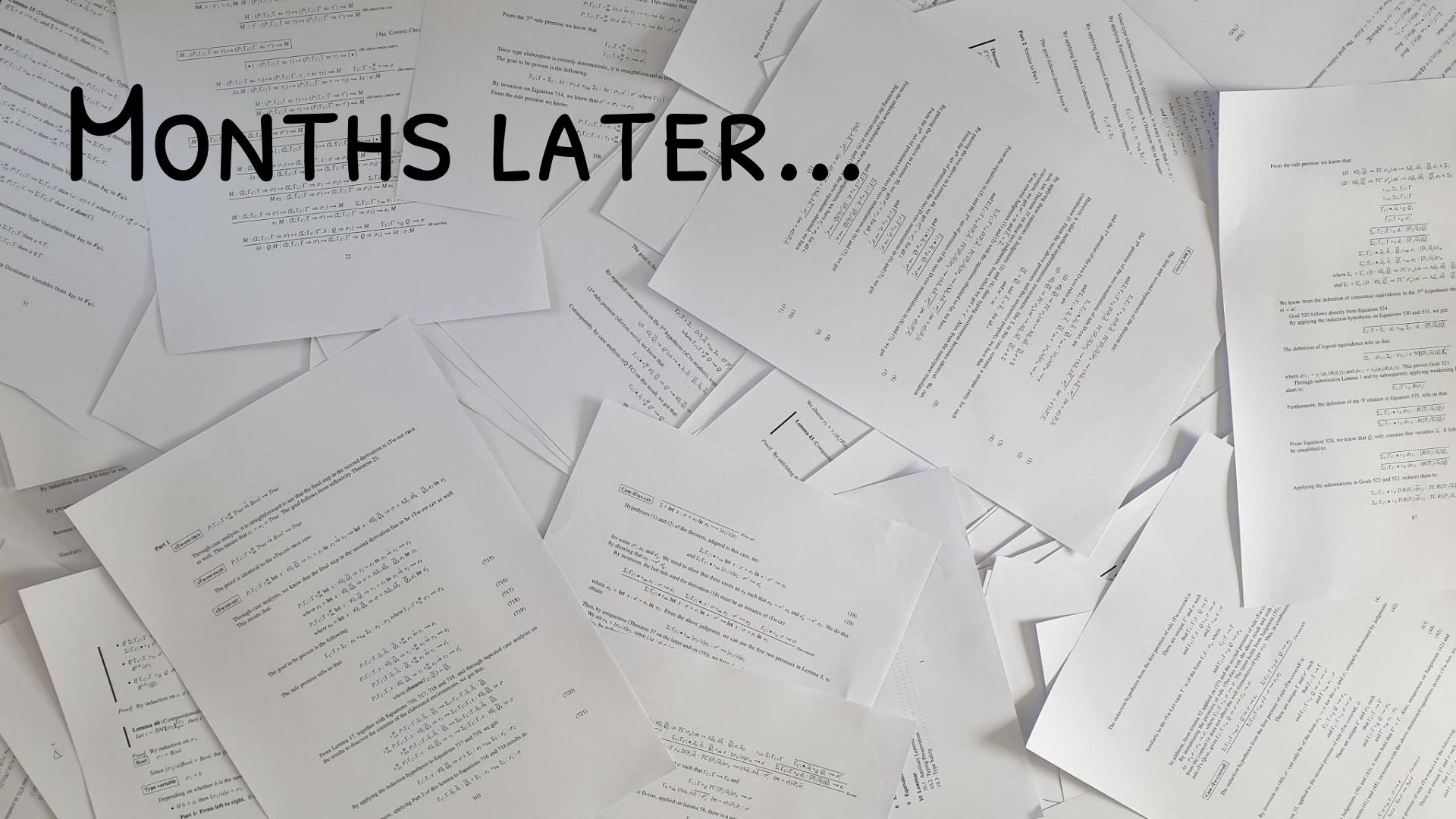


Determinism

INTERMEDIATE

(Implementation table)

Months later...



Coherence for ~~Haskell~~!
Type Class Resolution
+ Non-overlapping Instances

G.J. Bottu ~ N. Xie ~ K. Marntirosian ~ T. Schrijvers

Appendix
Coherence of Type Class Resolution

Gert-Jan Bottu[1], Ningning Xie[2], Koar Marntirosian[1], Tom Schrijvers[1]

[1] KU Leuven [2] The University of Hong Kong

Contents

Coherence of Type Class Resolution

GERT-JAN BOTTU, KU Leuven, Belgium
NINGNING XIE, The University of Hong Kong, China
KOAR MARNTIROSIAN, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium

Elaboration-based type class resolution, as found in languages like Haskell, Mercury and PureScript, is generally nondeterministic: there can be multiple ways to satisfy a wanted constraint in terms of global instances and locally given constraints. Coherence is the key property that keeps this sane; it guarantees that, despite the nondeterminism, programs still behave predictably. Even though elaboration-based resolution is generally assumed coherent, as far as we know, there is no formal proof of this property in the presence of sources of nondeterminism, like superclasses and flexible contexts.

This paper provides a formal proof to remedy the situation. 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.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Correctness**; **Functional languages**.

Additional Key Words and Phrases: type class resolution, coherence, logical relations

ACM Reference Format:
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. https://doi.org/10.1145/3341695

As a guide to the reader, we present the source language λ_{TC} in blue, the intermediate language F_D in green and the target language $F_{\{\}}$ in red. We thus encourage the reader to view / print this paper in color.

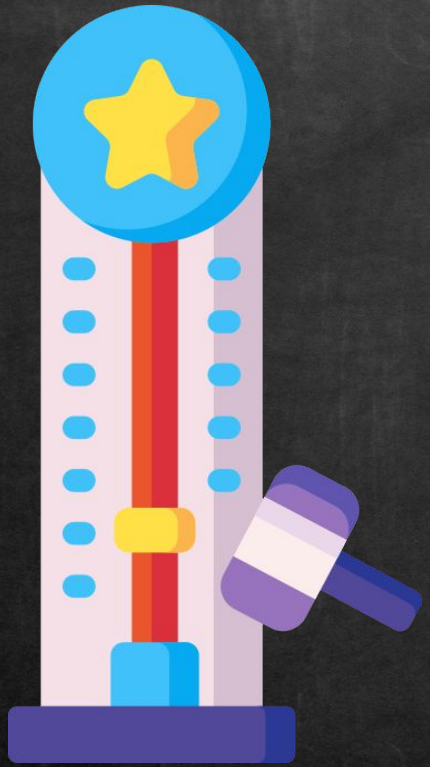
1 INTRODUCTION

Type classes were initially introduced in Haskell [Peyton Jones 2003] by Wadler and Blott [Wadler and Blott 1989] 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 [Henderson et al. 1996], Coq [Sozeau and Oury 2008], PureScript [Freeman 2017], Lean [de Moura et al. 2015]), and they have inspired various alternative language features (e.g., Scala's implicits [Martin Odersky and Venners 2008];

Authors' addresses: Gert-Jan Bottu, Department of Computer Science, KU Leuven, Belgium, gertjan.bottu@kuleuven.be; Ningning Xie, Department of Computer Science, The University of Hong Kong, China, nnxie@cs.hku.hk; Koar Marntirosian, Department of Computer Science, KU Leuven, Belgium, koar.marntirosian@kuleuven.be; Tom Schrijvers, Department of Computer Science, KU Leuven, Belgium, tom.schrijvers@kuleuven.be.

2475-1421/2019/8-ART91
https://doi.org/10.1145/3341695

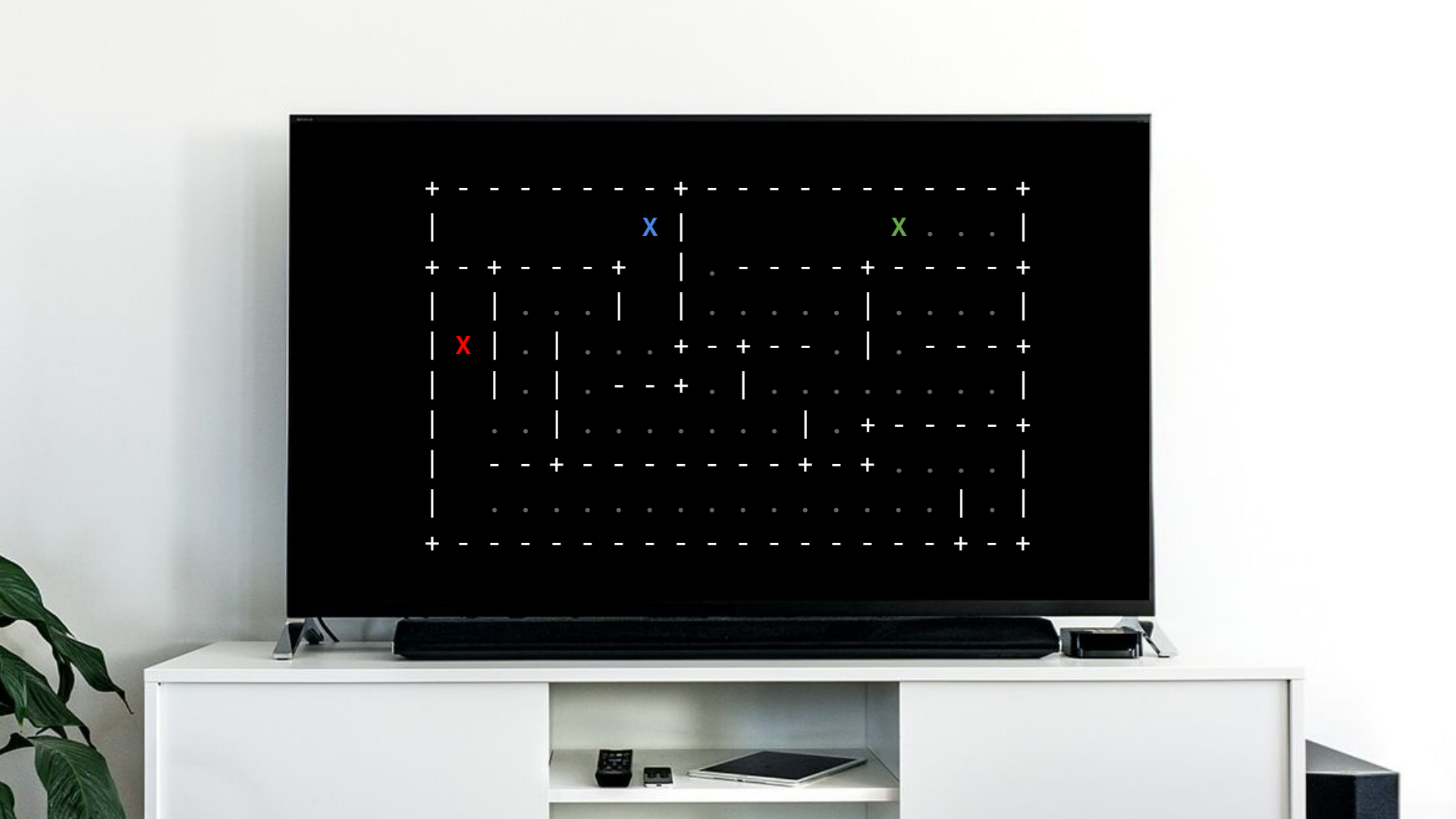
Mechaniz-o-meter



27%


C
X
X


C
X
X


X
X
X


GAME OVER