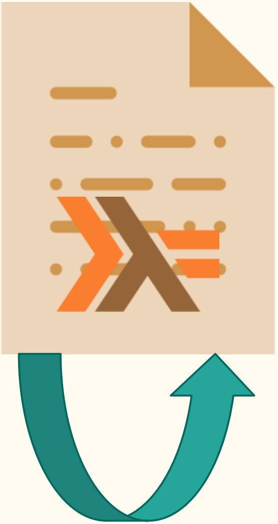
Proving Haskell Coherent

Gert-Jan Bottu, Ningning Xie,
Koar Marntirosian, Tom Schrijvers

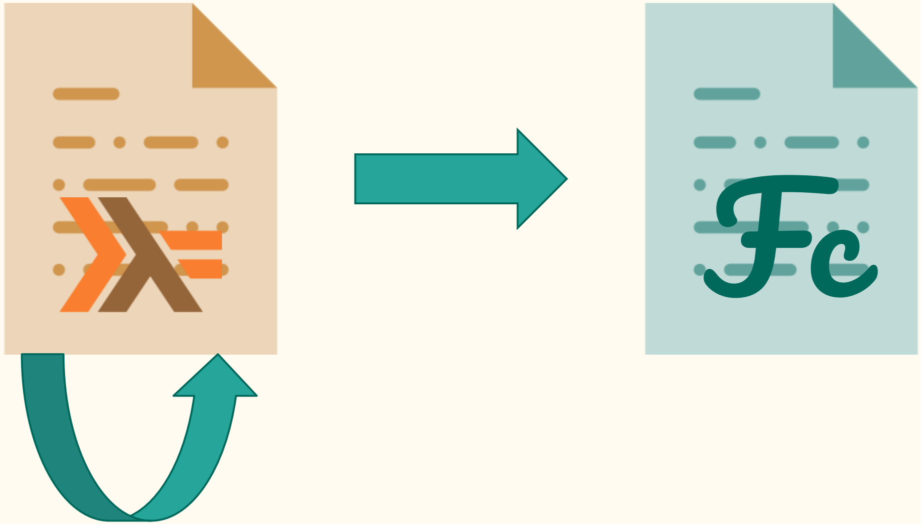
KU LEUVEN

Background

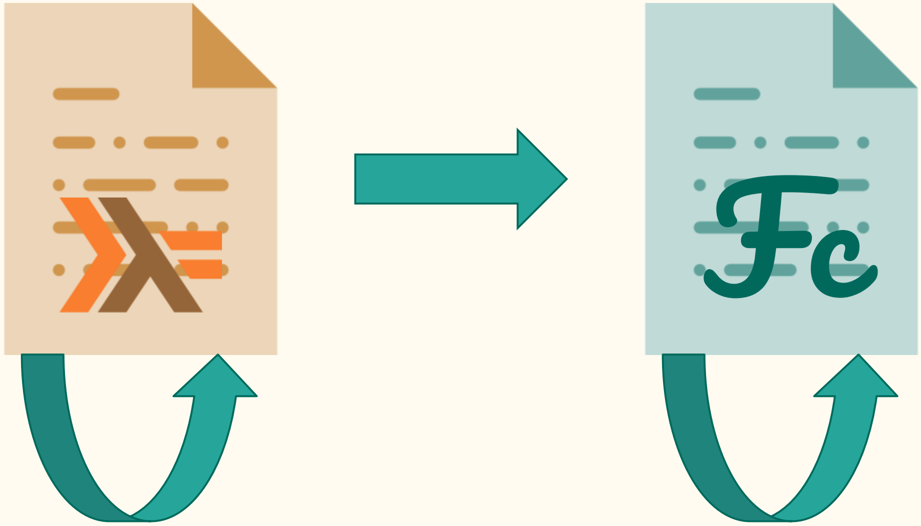

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?



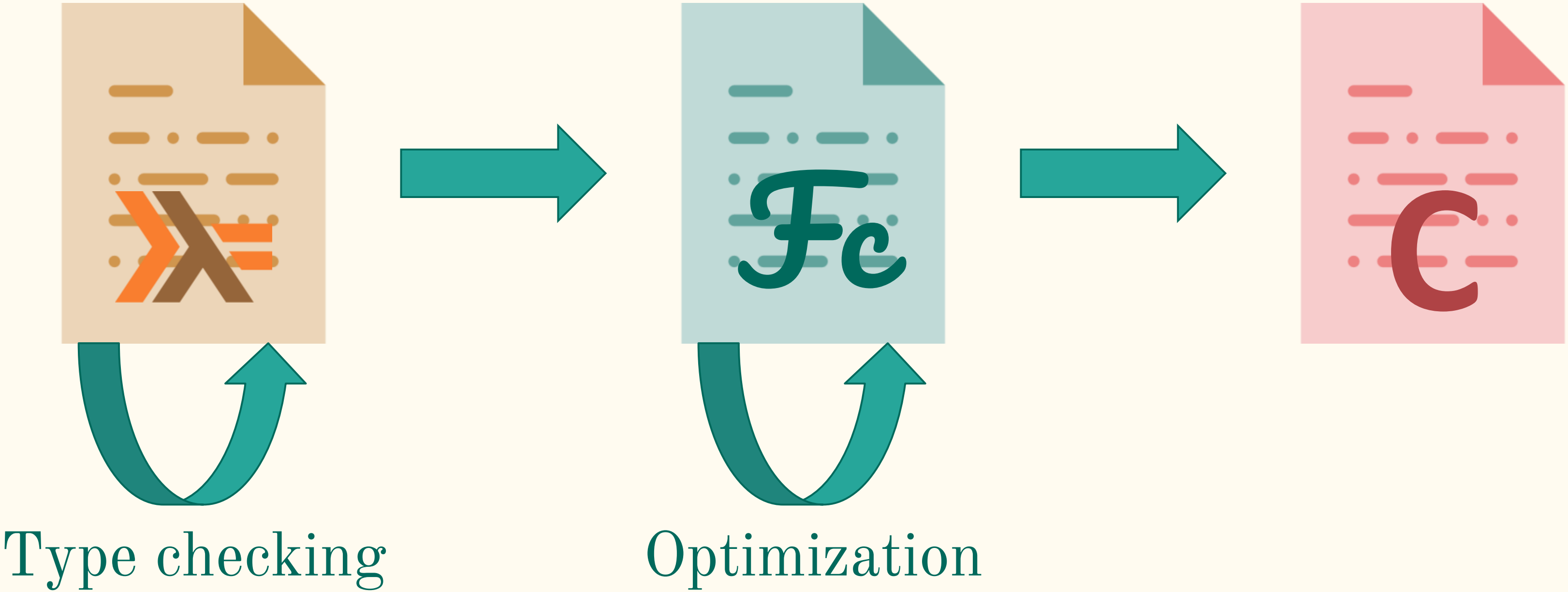
Type checking

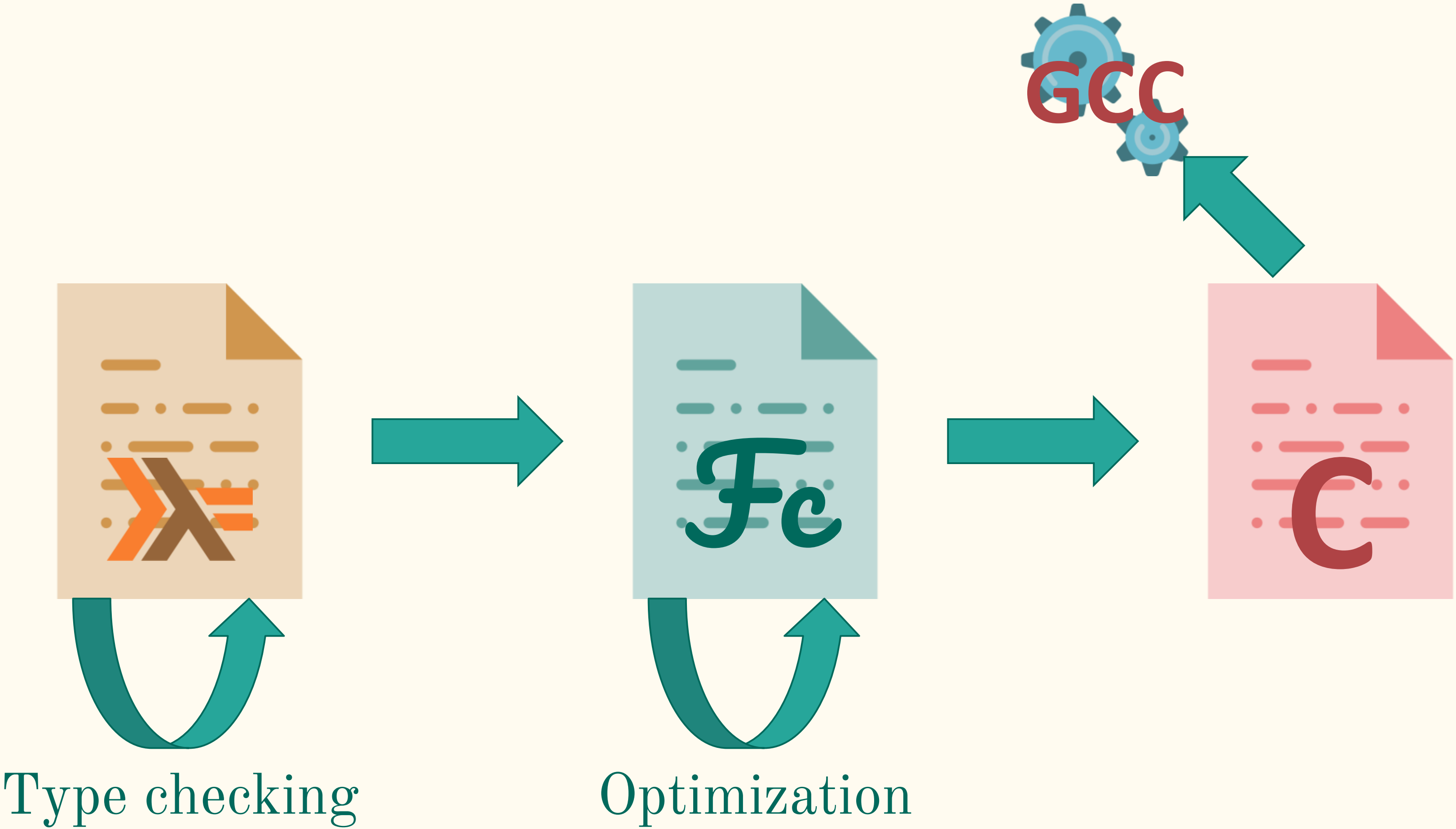


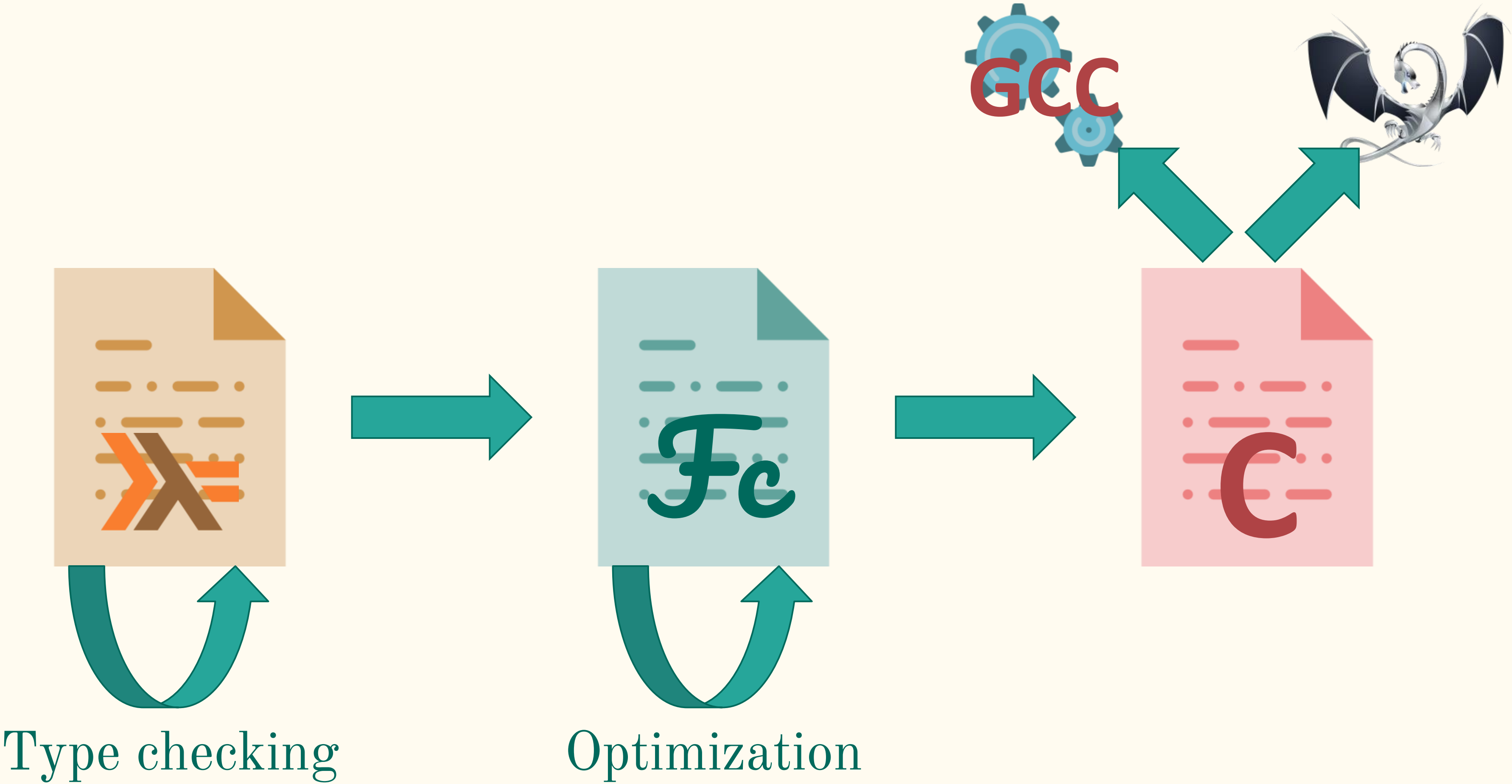
Type checking

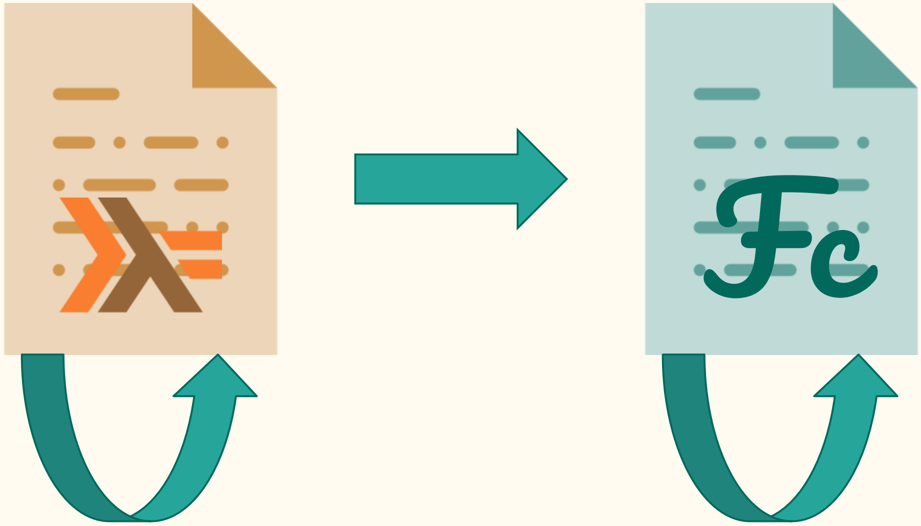


Type checking Optimization

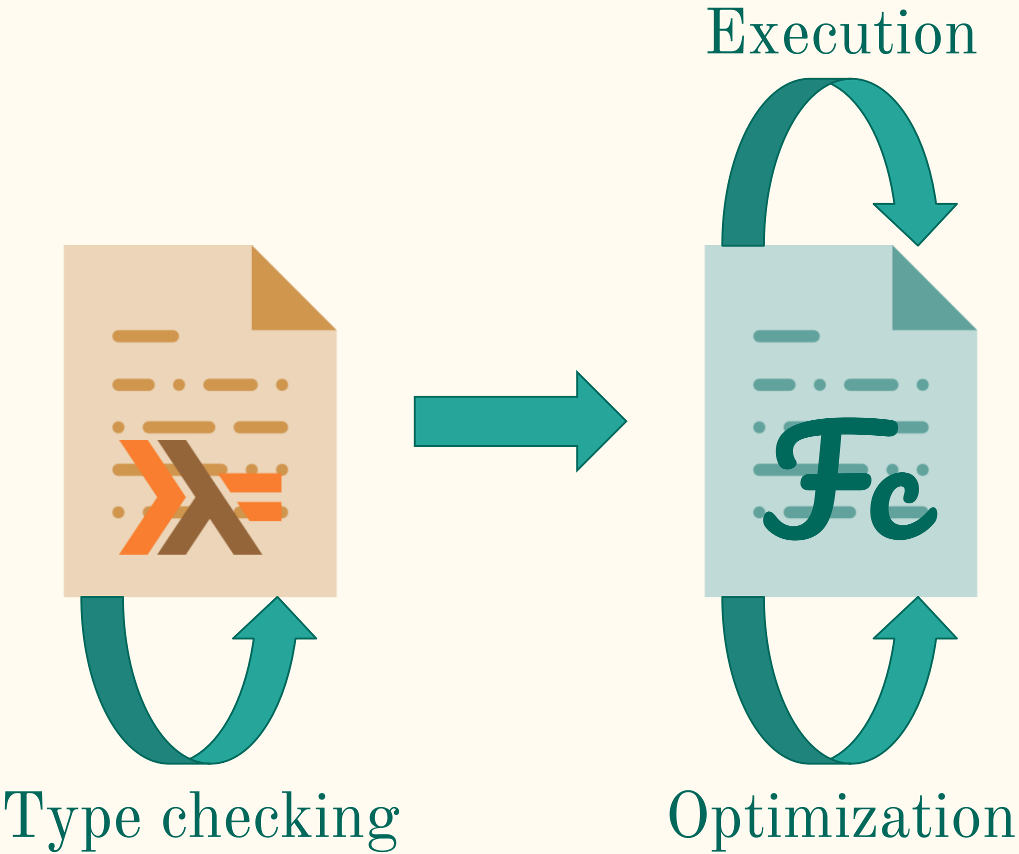

Fc
C
Type checking
Optimization


GCC
Fc
C
Type checking
Optimization


GCC
Fc
C
Type checking
Optimization



Type checking Optimization


Execution
Fc
Type checking
Optimization

```haskell
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

a = ?

```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

a = ?

```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

```
> foo "1"
```

a = ?

```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

1 (Int)

```
> foo "1"
```

a = ?

```haskell
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

1 (Int)

```
> foo "1"
```

1.0 (Float)

a = ?

```haskell
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

```
1 (Int)

> foo "1"        1.0 (Float)

True (Bool)
```

a = ?

```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

```
> foo "1"
```

1 (Int)

1.0 (Float)

True (Bool)

???

Ambiguity!

PICKERING 8
WHITBY 29½
LEEDS 41
NORWICH 189
LIVERPOOL 112
NEWCASTLE 98
HULL 38
TADCASTER 29
ALFRETON 88
SCARBOROUGH 23
223
LONDON 229
SHEFFIELD 69
BARROW IN FURNESS 134
BEDFORD 165
BIRMINGHAM
249

Coherence for qualified types

Mark P. Jones

Coherence for qualified types

Mark P. Jones*

All translations are equal, but some translations are more equal than others.
Misquoted, with apologies to George Orwell, from *Translation Farm*, 1945.

Research Report YALEU/DCS/RR-989, September 1993

Abstract

The meaning of programs in a language with implicit overloading can be described by translating them into a second language that makes the use of overloading explicit. A single program may have many distinct translations and it is important to show that any two translations are semantically equivalent to ensure that the meaning of the original program is well-defined. This property is commonly known as *coherence*.

This paper deals with an implicitly typed language that includes support for parametric polymorphism and overloading based on a system of *qualified types*. Typical applications include Haskell type classes, extensible records and subtyping. In the general case, it is possible to find examples for which the coherence property does not hold. Extending the development of a type inference algorithm for this language to include the calculation of translations, we give a simple syntactic condition on the principal type scheme of a term that is sufficient to guarantee coherence for a large class of programs.

One of the most interesting aspects of this work is the use of terms in the target language to provide a semantic interpretation for the ordering relation between types that is used to establish the existence of principal types.

On a practical level, our results explain the importance of *unambiguous* type schemes in Haskell.

Introduction

Consider the task of evaluating an expression of the form $x + y + z$. Depending on the way that it is parsed, this expression might be treated as either $(x+y)+z$ or $x+(y+z)$. Fortunately, it does not matter which of these we choose since the fact that $(+)$ is associative is both necessary and sufficient to guarantee that they are actually equivalent. We are therefore free to choose whichever is more convenient, retaining the same well-defined semantics in either case.

This paper deals with a similar problem that occurs with programs in OML, a simple implicitly typed language with overloading. The meaning of such programs can be described by translating them into OP, an extended language which uses additional constructs to make the use of overloading explicit. However, different typing derivations for a given OML program can lead to distinct translations and, just as in the example above, it is important to show that any two translations have the same meaning. In the terminology of [2], we need to show that 'the meaning of a term does not depend on the way that it was type checked', a property that they refer to as *coherence*.

The type system of OML is an extended form of the ML type system that includes support for qualified types [7]. The central idea is to allow the use of type expressions of the form $\pi \Rightarrow \sigma$ to represent all those instances of σ which satisfy π, a *predicate* on types. Applications of qualified types include Haskell type classes, extensible records and subtyping.

In previous work, we have described how the standard type inference algorithm for ML can be extended to calculate *principal type schemes* for terms in OML. In this paper, we extend these results to show how an arbitrary translation of an OML term can be written in terms of a particular *principal translation* determined by the type inference algorithm. Exploiting this relationship, we give conditions that can be used to guarantee that all of the translations for a given term are equivalent.

The remaining sections of this paper are as follows. Section 1 outlines the use of qualified types and defines the languages OML and OP and the translation between them that is used in this paper. A simple example in Section 2 shows that a single term may have semantically distinct translations and hence that we cannot hope to establish a general coherence result for arbitrary terms. Instead, we must look for conditions which can be used to ensure coherence for as wide a class of programs as possible.

As a first step, we need to specify exactly what it means for two translations to be equivalent. This is dealt with in Section 3 using a syntactic definition of (typed) equality between OP terms.

One of the most important tools in the development of a type inference algorithm is the ordering relation $(\leq)$ between type schemes. Indeed, without a notion of ordering, it would not even be possible to talk about principal or most general type schemes! Motivated by this, Section 4 gives a semantic interpretation for $(\leq)$ using OP terms which we call *conversions*.

Sections 5 and 6 extend the development of type inference

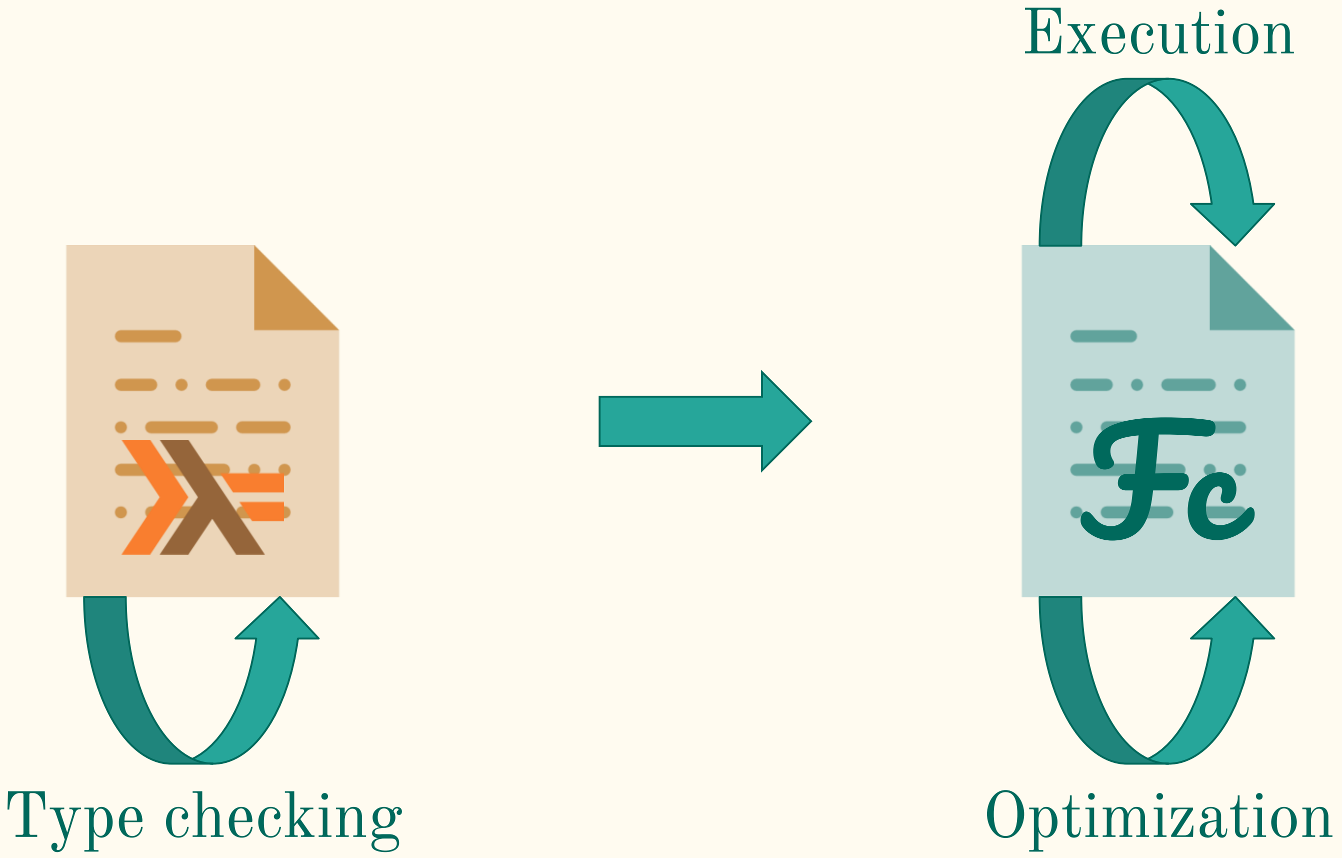
*This paper summarizes work carried out while the author was a member of the Programming Research Group, Oxford, supported by a SERC studentship [8]. Current address: Yale University, Department of Computer Science, P.O. Box 208285, New Haven, Connecticut 06520-8285, USA. Electronic mail jones-mark@cs.yale.edu. Supported in part by a grant from DARPA, contract number N00014-91-J-4043.

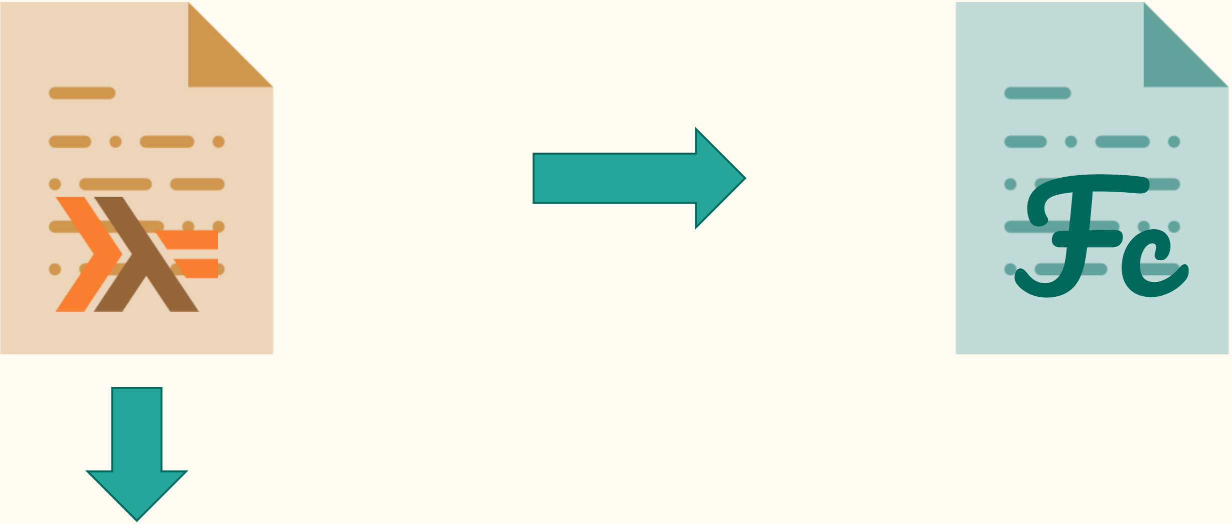
a = ?

```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

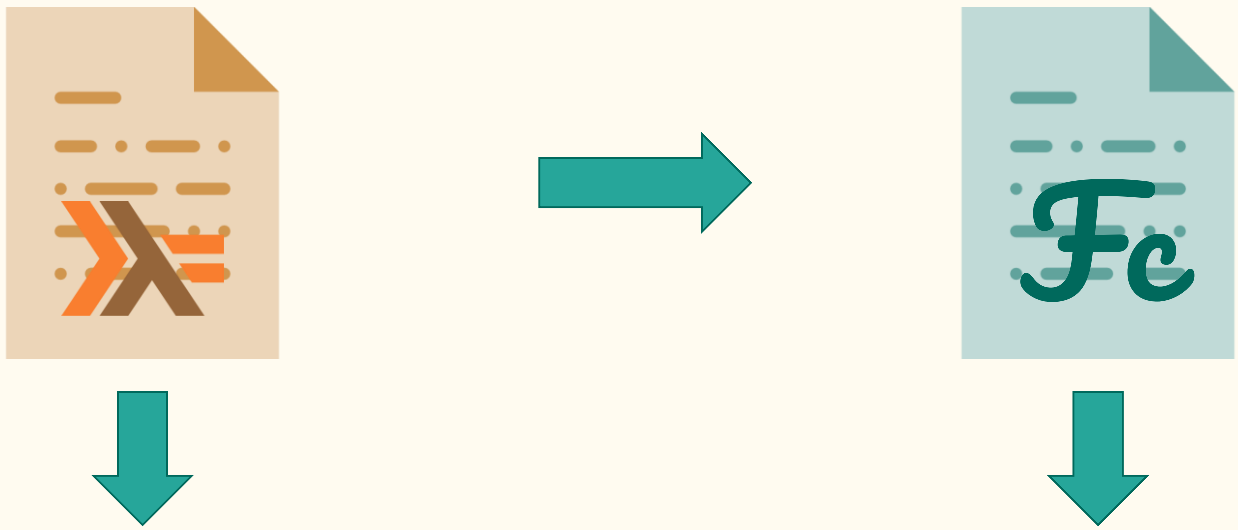
a = ?

```
foo :: (Show a , Read          tring -> String
foo s = show (read
```


Execution
Type checking
Optimization

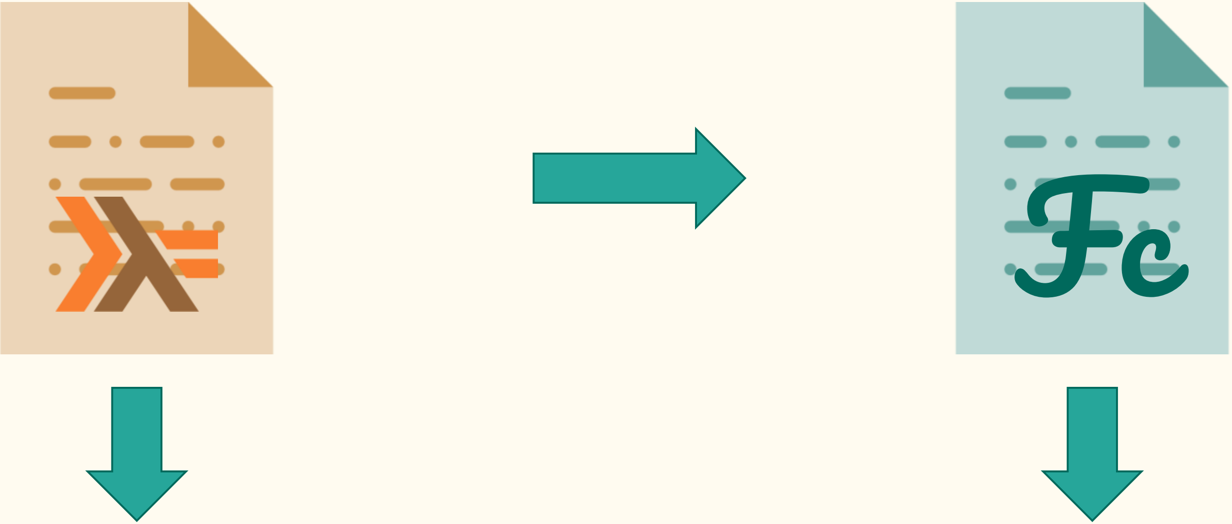


Type classes!



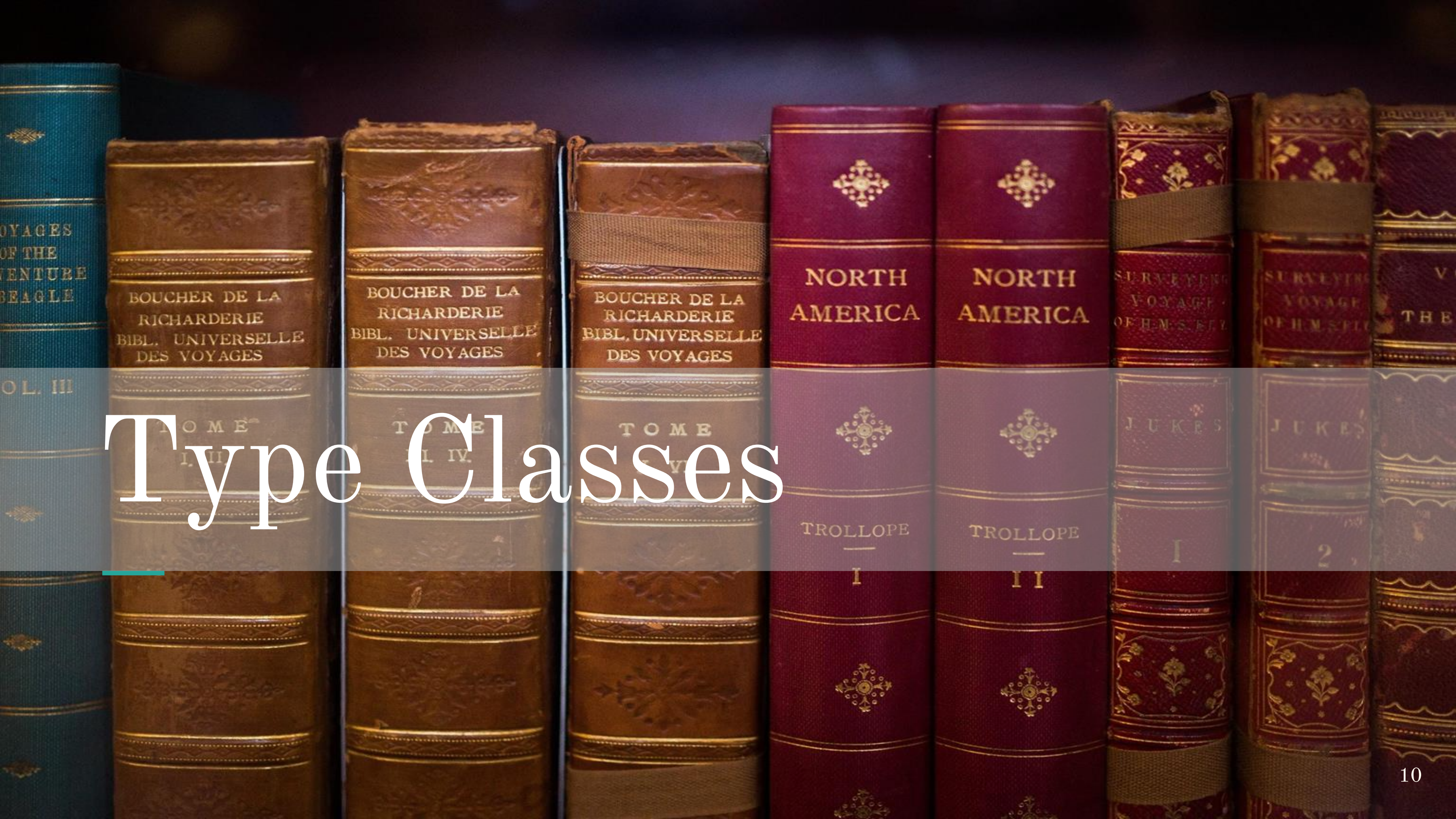
Type classes!

No type classes?



Type classes!

No type classes?
Dictionaries!



Type Classes

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

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

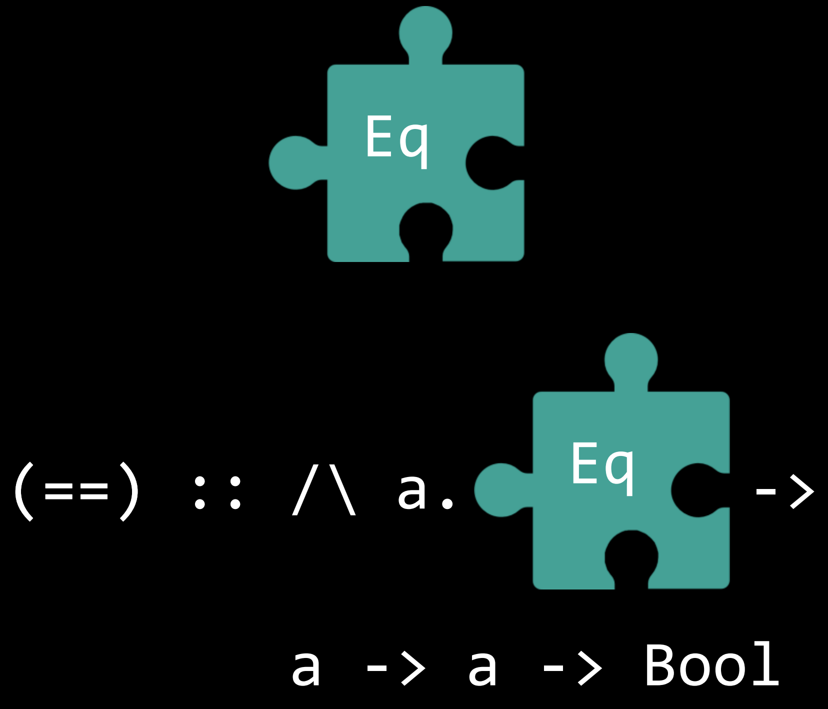
```
class Eq a where
  (==) :: a -> a  -> Bool

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```


```
(==) ::
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```



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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

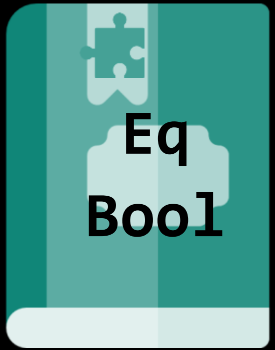
(==) :: /\ a. 

-> a -> a -> Bool

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a.  ->

a -> a -> Bool

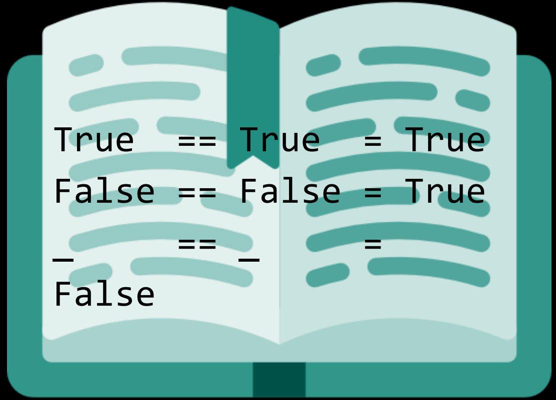


```
class Eq a where
  (==) :: a -> a  -> Bool

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a.

->

a -> a -> Bool



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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

```
(==) :: /\ a.
```


```
->
   a -> a -> Bool
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

```
(==) :: /\ a.
```

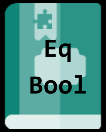

```
->

        a -> a -> Bool

foo =
```

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

: .

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

: 

.

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

: 

.

\x : Bool.

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

: 

.

\x : Bool.

(==)

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

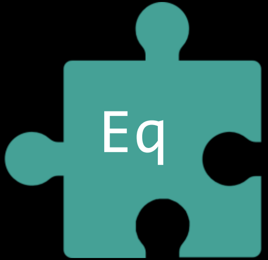
: 

.

\x : Bool.

(==) Bool

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

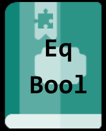
a -> a -> Bool

foo = \ 

: 

.

\x : Bool.

(==) Bool 

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

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

foo :: Bool -> Bool
foo x = x == x
```

(==) :: /\ a. 

->

a -> a -> Bool

foo = \ 

: 

.

\x : Bool.

(==) Bool 

x x

Superclasses

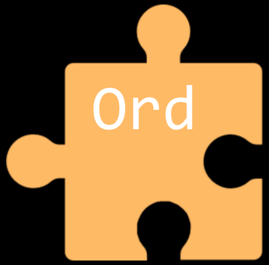
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False
```

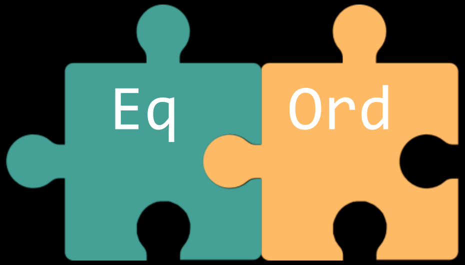
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

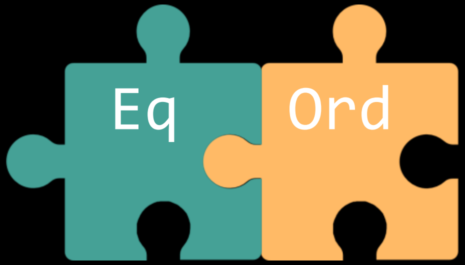
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

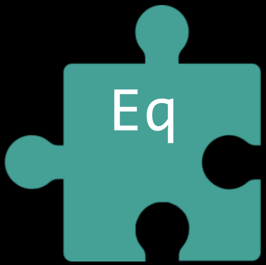


```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



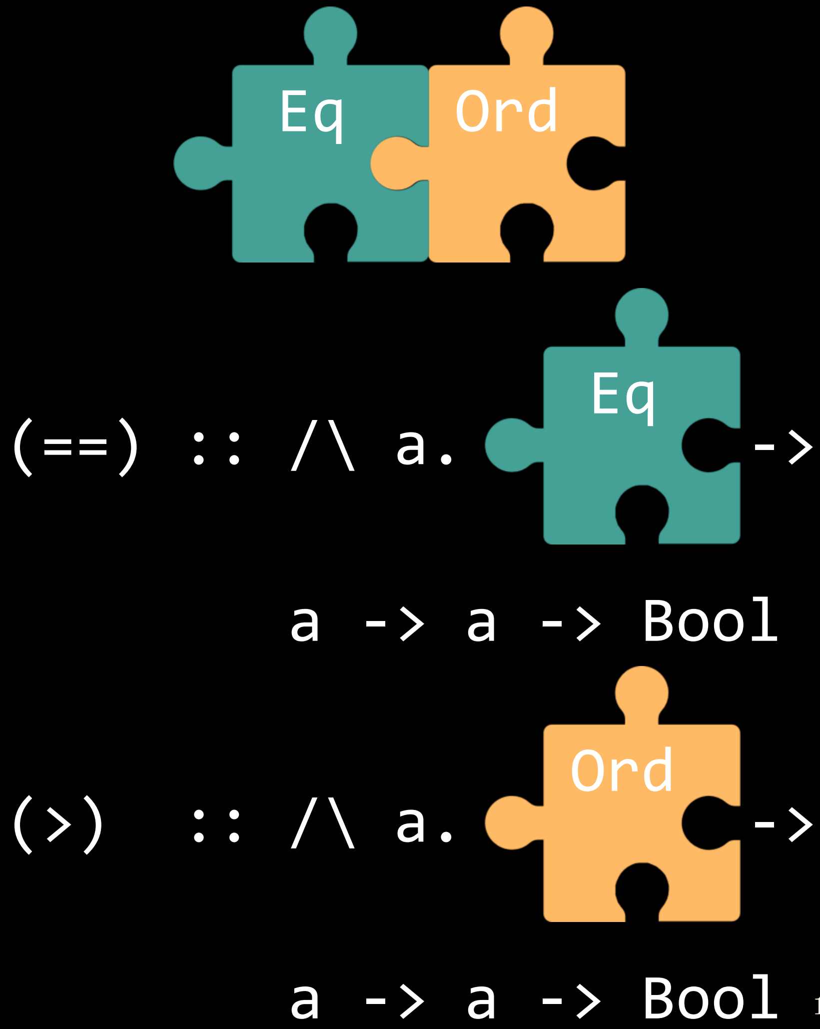
```haskell
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



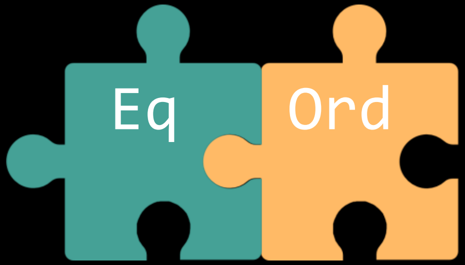
(==) :: /\ a. 

-> a -> a -> Bool

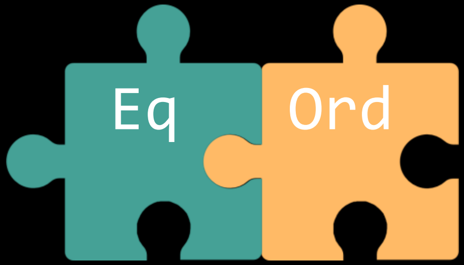
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



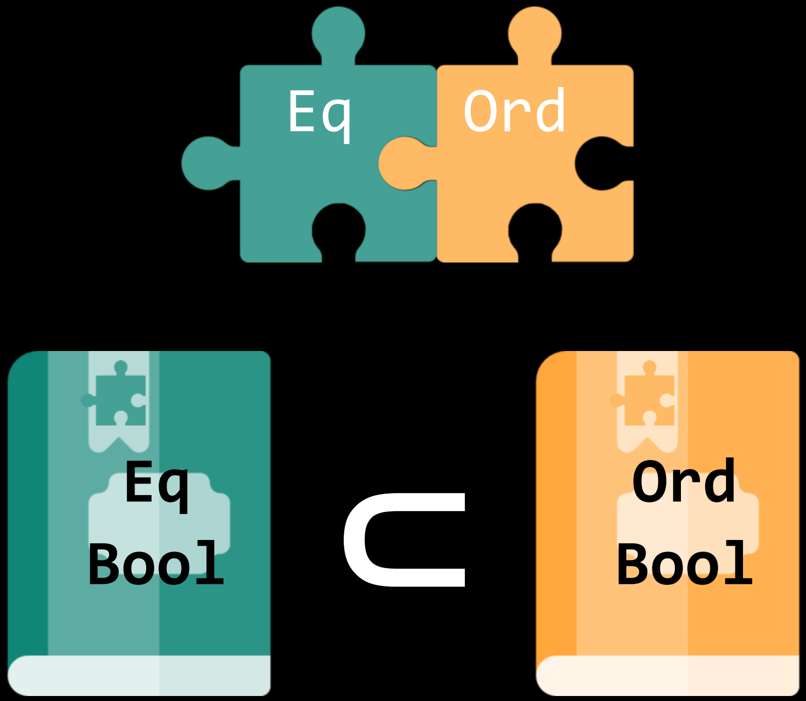
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```



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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

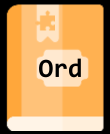
```
foo =
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

```
foo = /\ a .
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

```
foo = /\ a .
      \
```

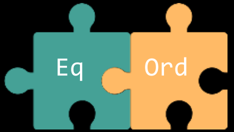



```
:          .
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

```
foo = /\ a .
```

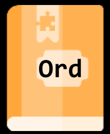
\ 

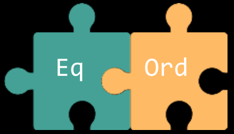
: 

.

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

foo = /\ a .

\ 

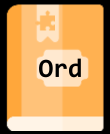
: 

.

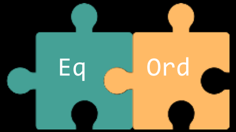
\x : a .

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

```
foo = /\ a .
      \
```

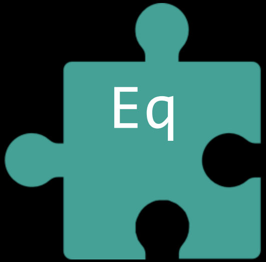



```
:
```




```
.
      \x : a .
      (==)
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

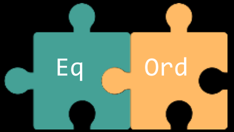
(==) :: /\ a. 

->

a -> a -> Bool

foo = /\ a .

\ 

: 

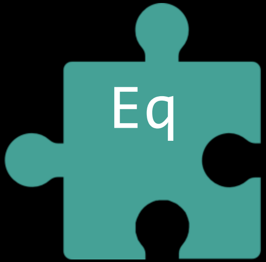
.

\x : a .

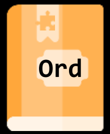
(==)

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

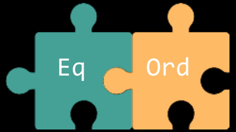
```
(==) :: /\ a.
```




```
->
     a -> a -> Bool

foo = /\ a .
      \
```

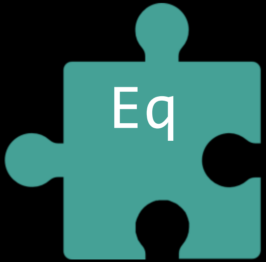



```
:
```




```
.
      \x : a .
      (==) a
```

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

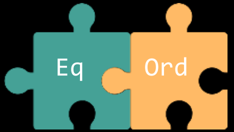
(==) :: /\ a. 

->

a -> a -> Bool

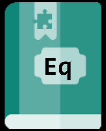
foo = /\ a .

\ 

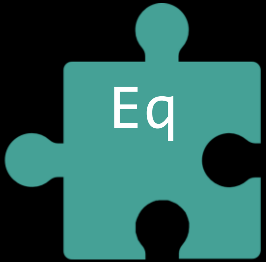
: 

.

\x : a .

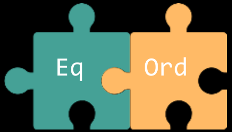
(==) a 

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

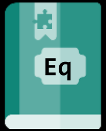
(==) :: /\ a.  ->

a -> a -> Bool

foo = /\ a .

\  :  .

\x : a .

(==) a  x x

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

inst Ord Bool where
  True > False = True
  _    > _     = False

foo :: Ord a => a -> Bool
foo x = x == x
```

```
(==) :: /\ a. Eq ->
          a -> a -> Bool

foo = /\ a .
Eq    \ Ord : Eq Ord .
      \x : a .
      (==) a Eq x x
```

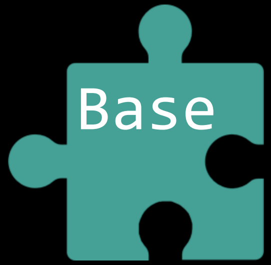
```
class Base a where
  base :: a -> Bool
```

```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a
```

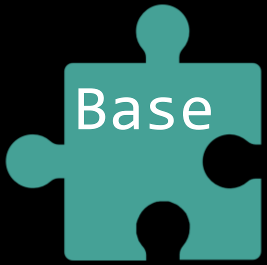
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```

```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```

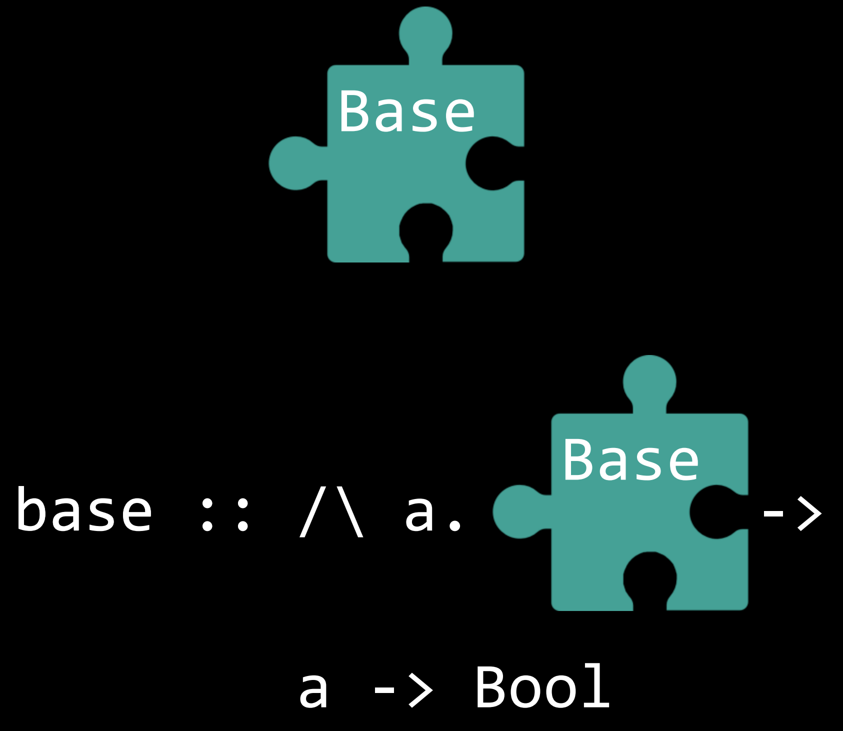


```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```



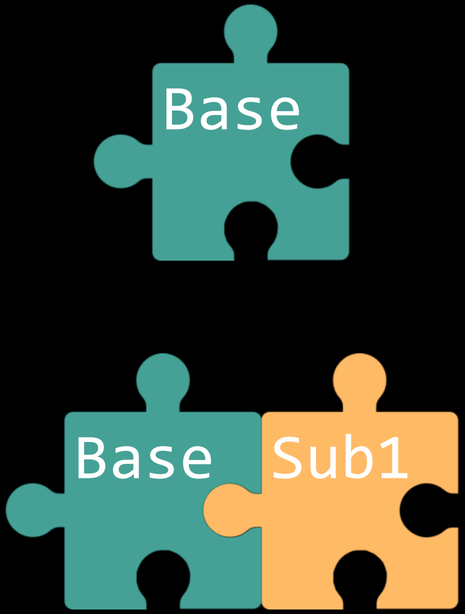

```
base ::
```

```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

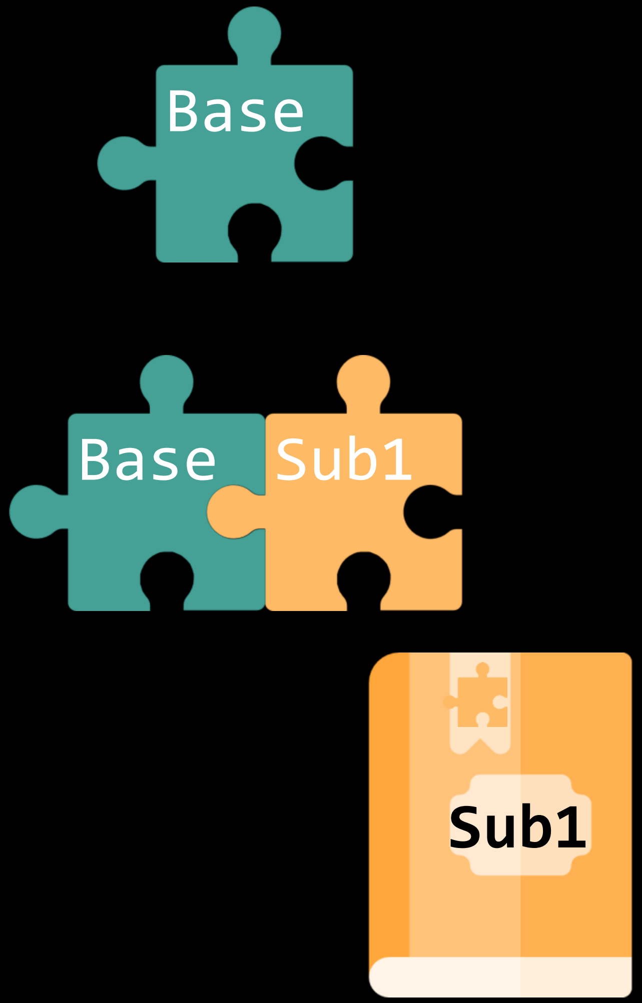


```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
    => a -> Bool
foo x = base x
```

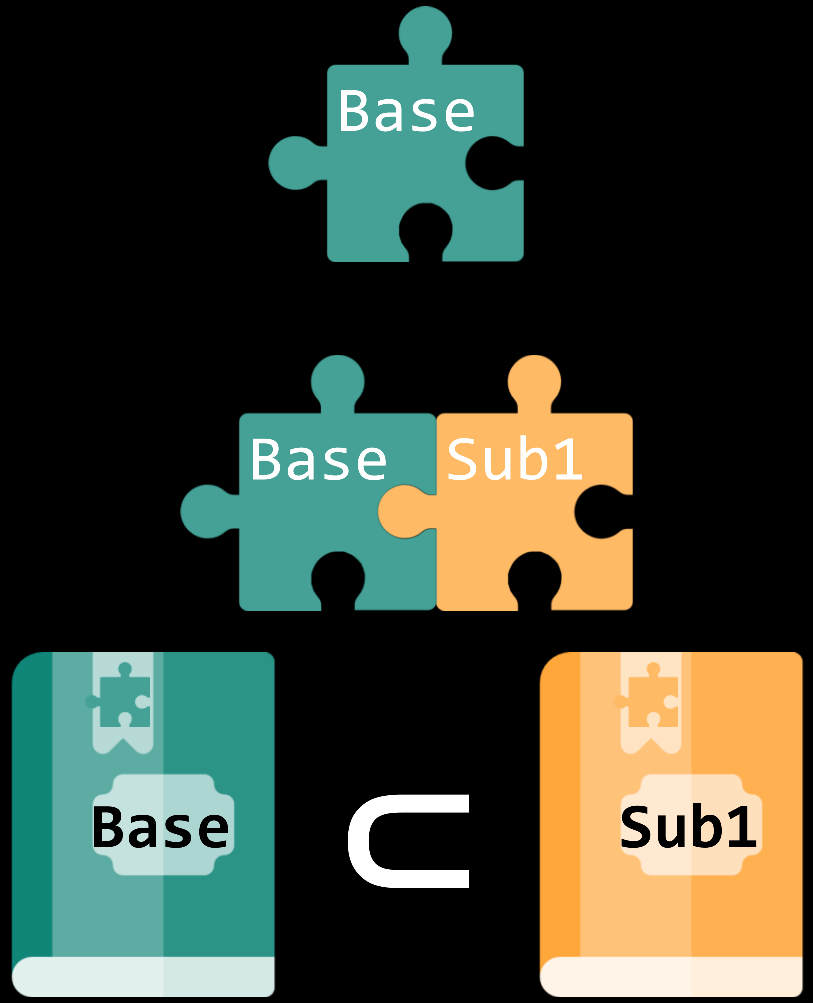
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```



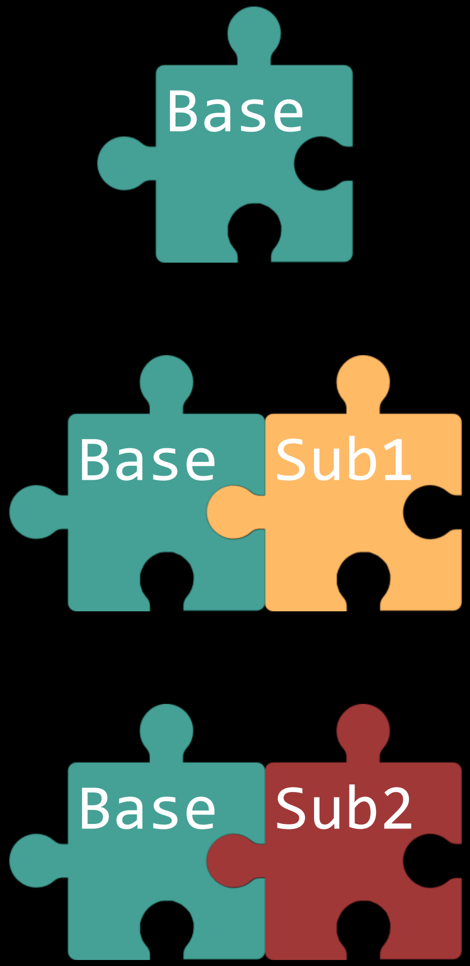
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
    => a -> Bool
foo x = base x
```



```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
    => a -> Bool
foo x = base x
```



```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```



```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
         => a -> Bool
foo x = base x
```

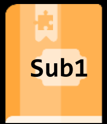
```
foo =
```

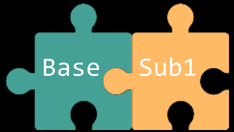
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

```
foo = /\ a .
```

```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```

```
foo = /\ a .
```

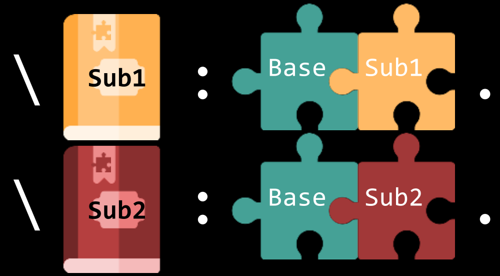
\ 

: 

.

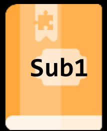
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

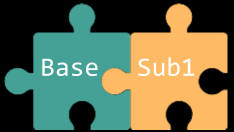
```
foo = /\ a .
```



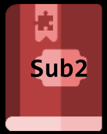
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
         => a -> Bool
foo x = base x
```

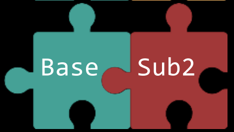
foo = /\ a .

\ 

: 

.

\ 

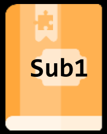
: 

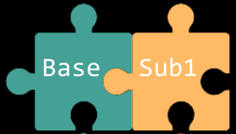
.

\x : a .

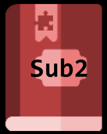
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```

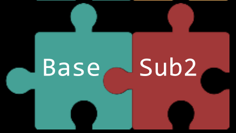
```
foo = /\ a .
```

\ 

: 

.

\ 

: 

.

```
\x : a .
base
```

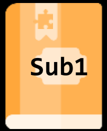
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

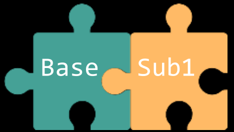
base :: /\ a. 

->

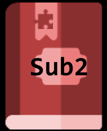
a -> Bool

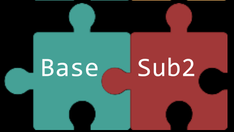
foo = /\ a .

\ 

: 

.

\ 

: 

.

\x : a .

base

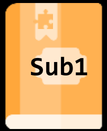
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
       => a -> Bool
foo x = base x
```

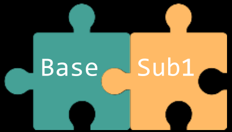
base :: /\ a. 

->

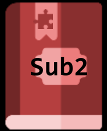
a -> Bool

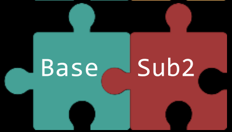
foo = /\ a .

\ 

: 

.

\ 

: 

.

\x : a .

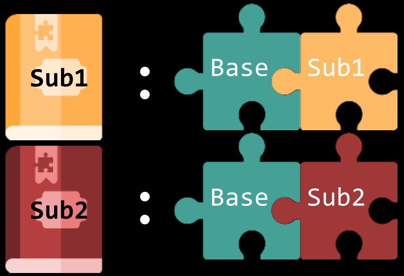
base a

```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

base :: /\ a.  ->

a -> Bool

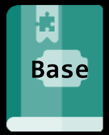
foo = /\ a .



\ : .

\ : .

\x : a .

base a 

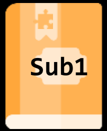
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

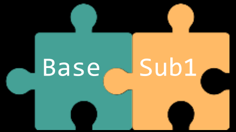
base :: /\ a. 

->

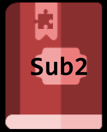
a -> Bool

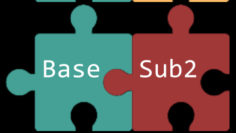
foo = /\ a .

\ 

: 

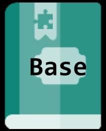
.

\ 

: 

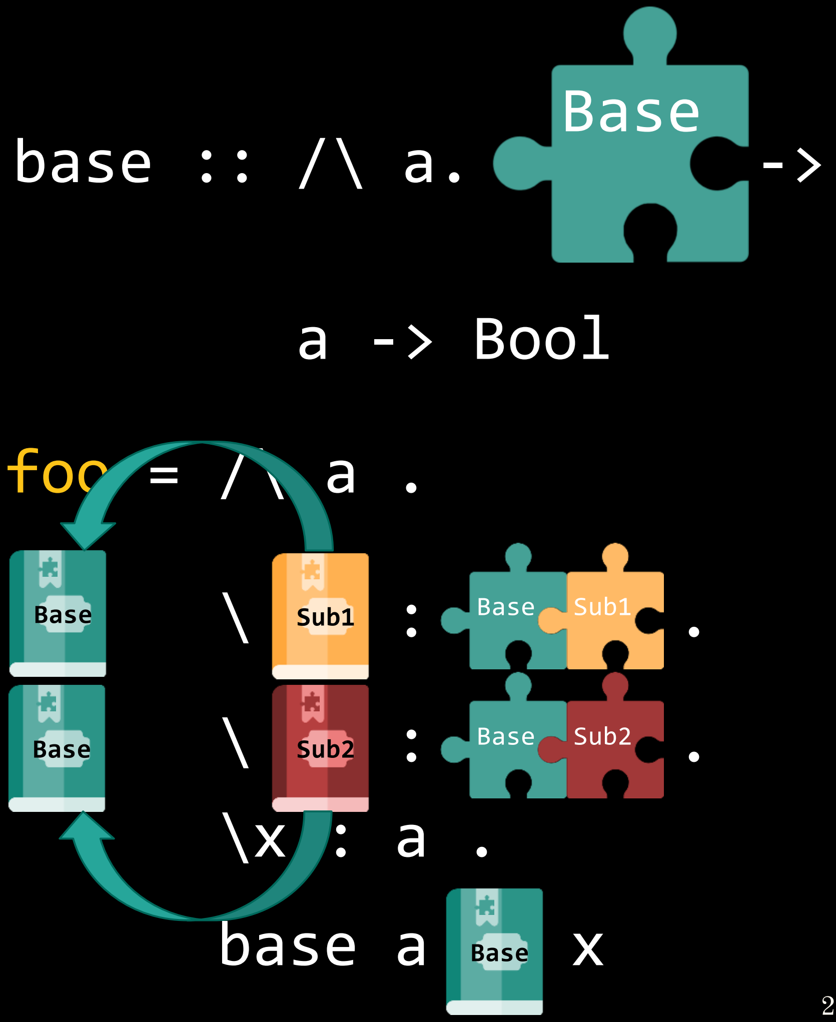
.

\x : a .

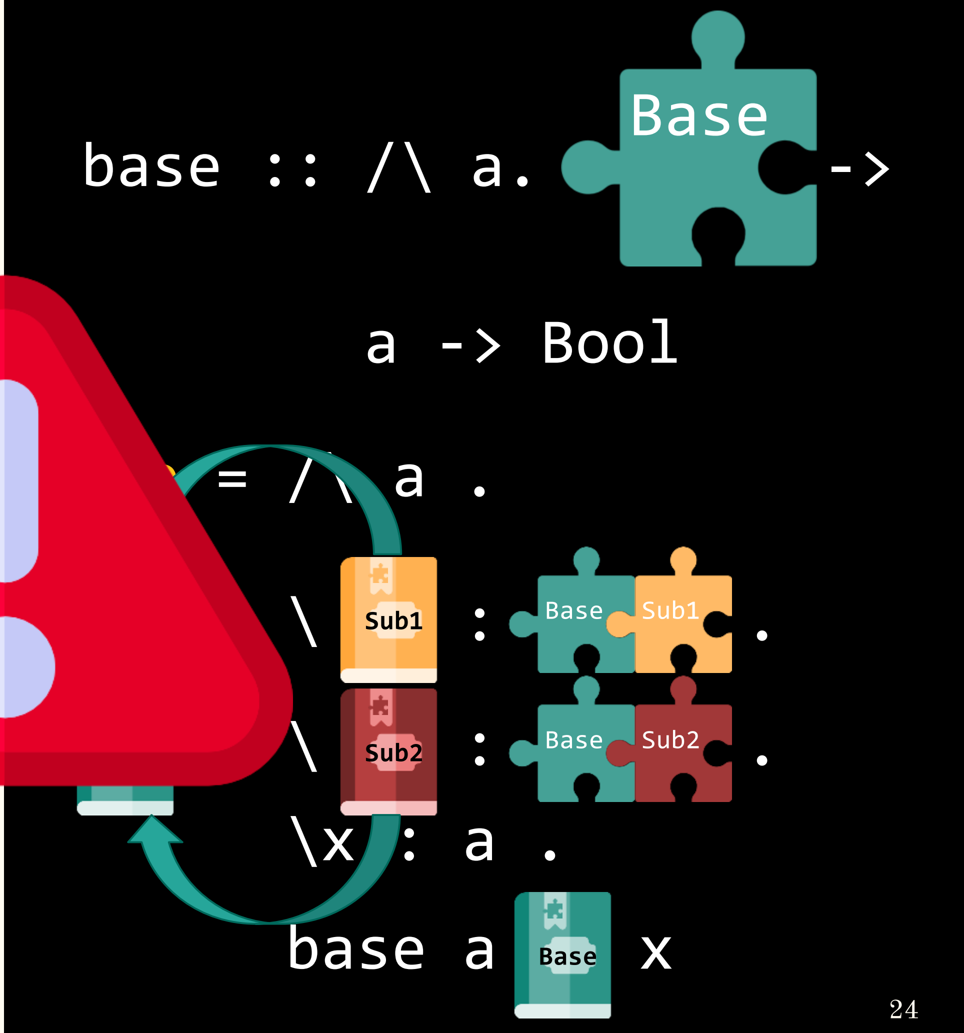
base a 

x

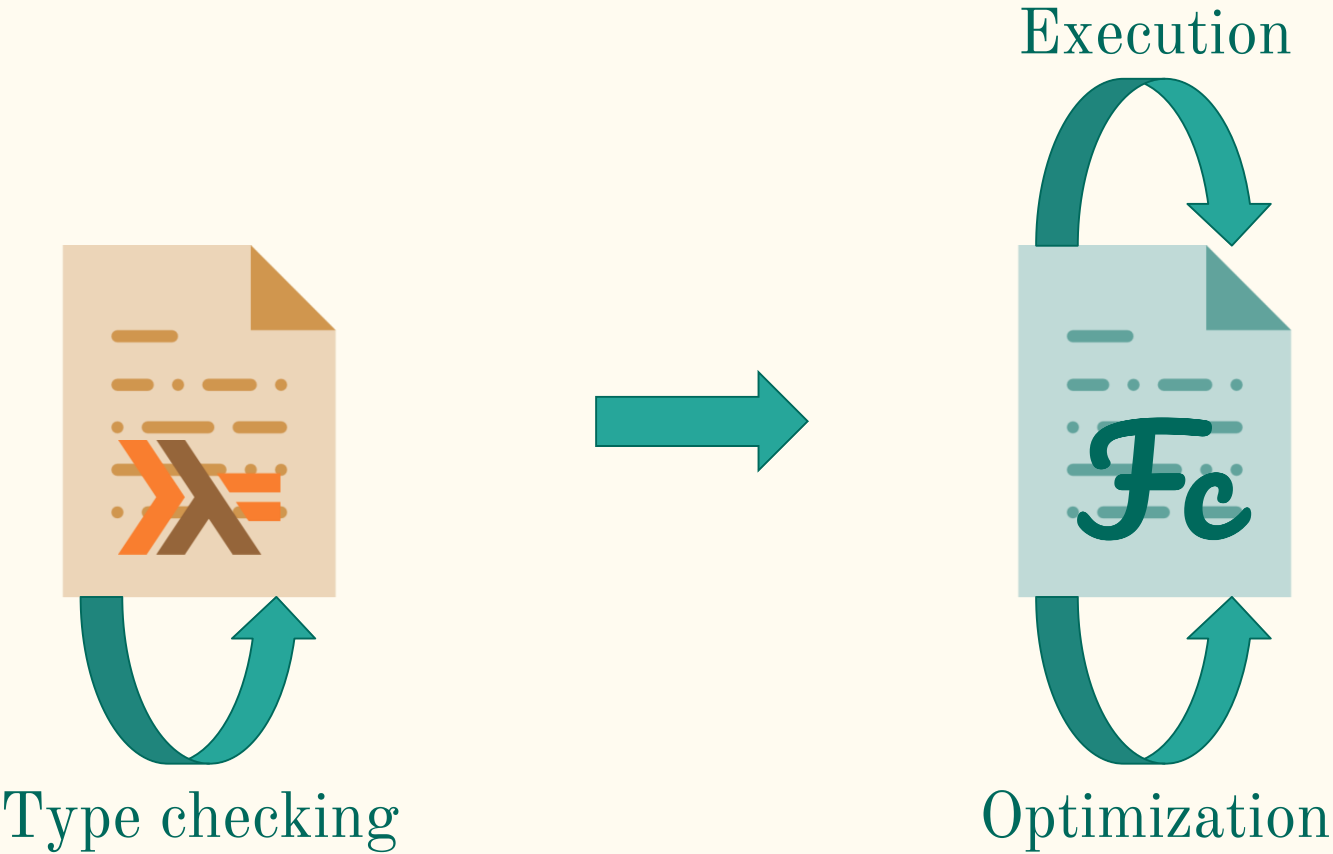
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2 a

foo :: (Sub1 a, Sub2 a)
        => a -> Bool
foo x = base x
```

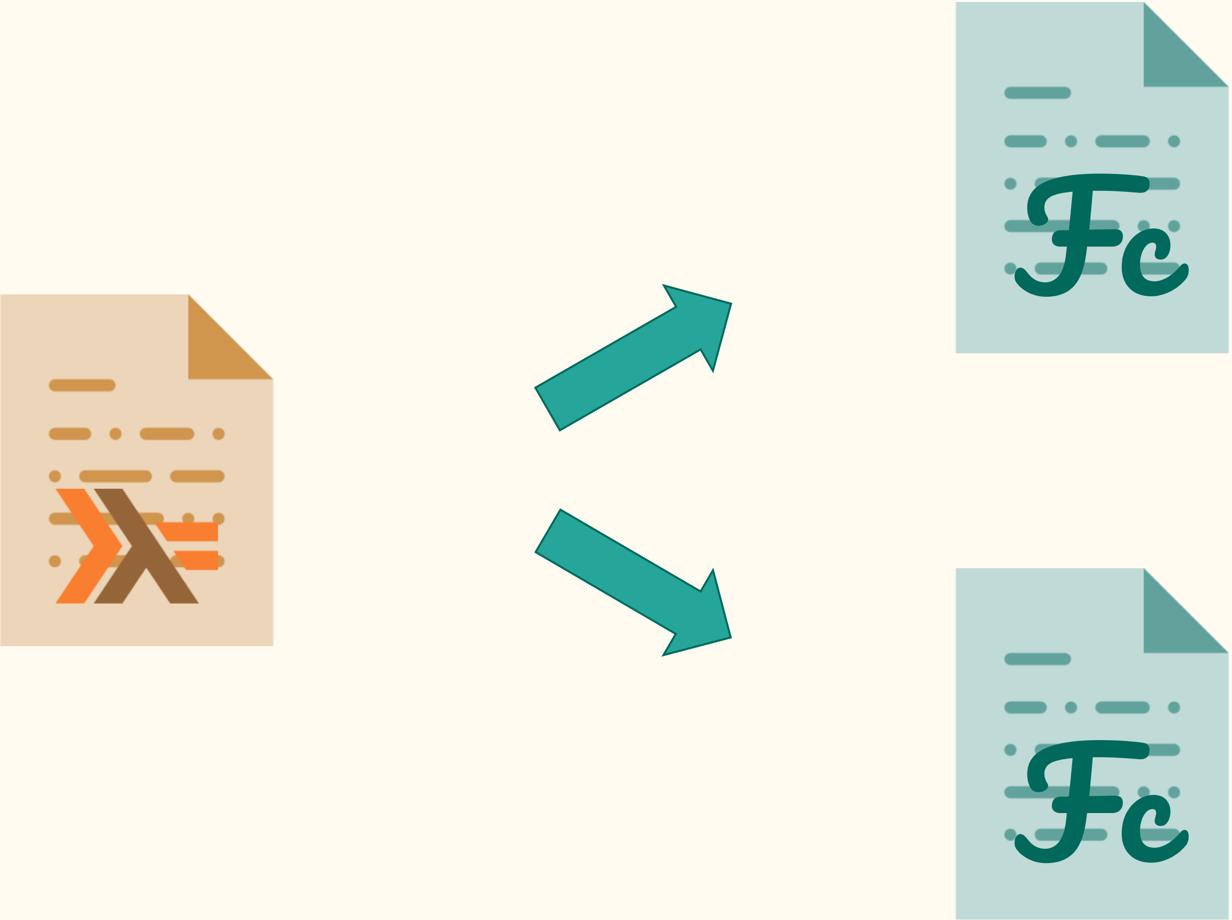


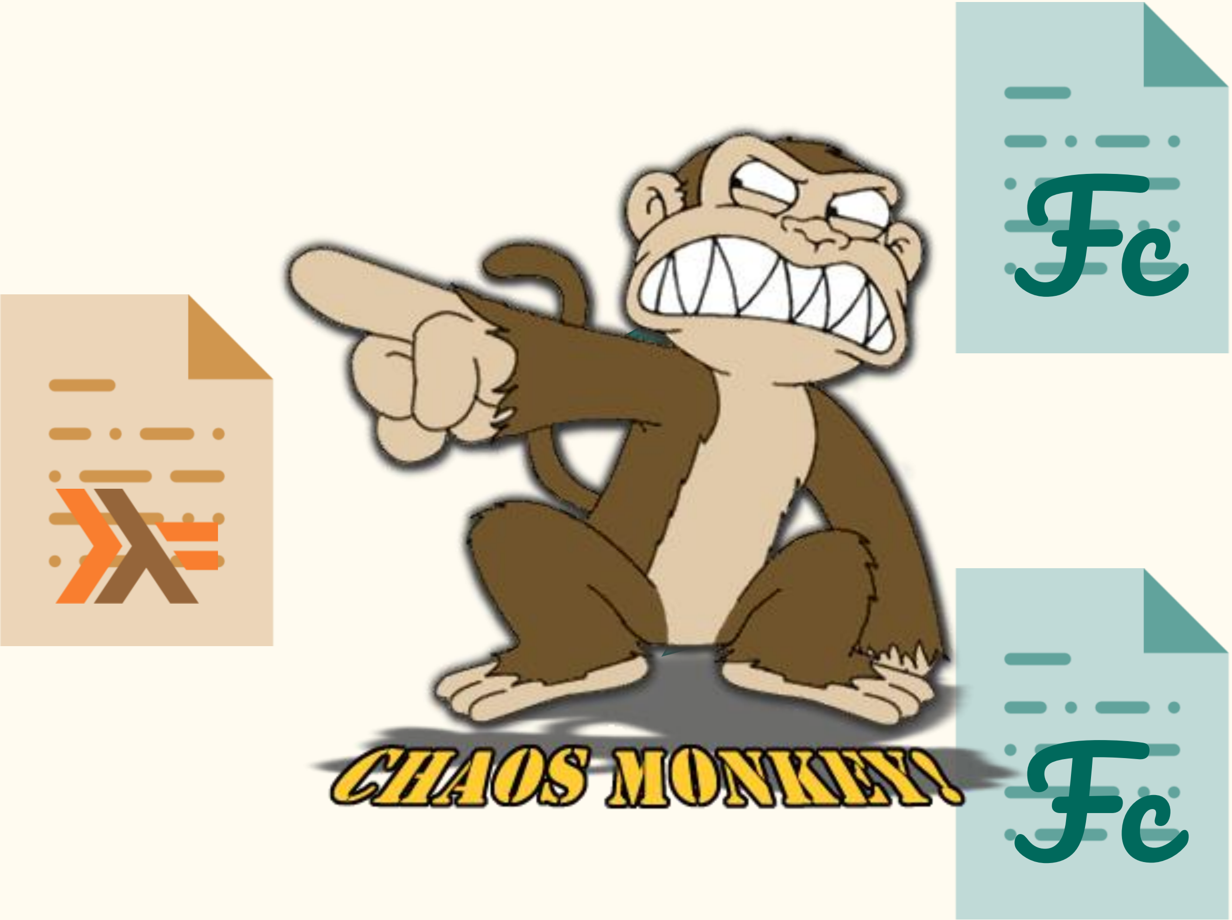
```
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
class Base a => Sub2

foo :: (Sub1 a, Su
        => a -> Bool
foo x = base x
```

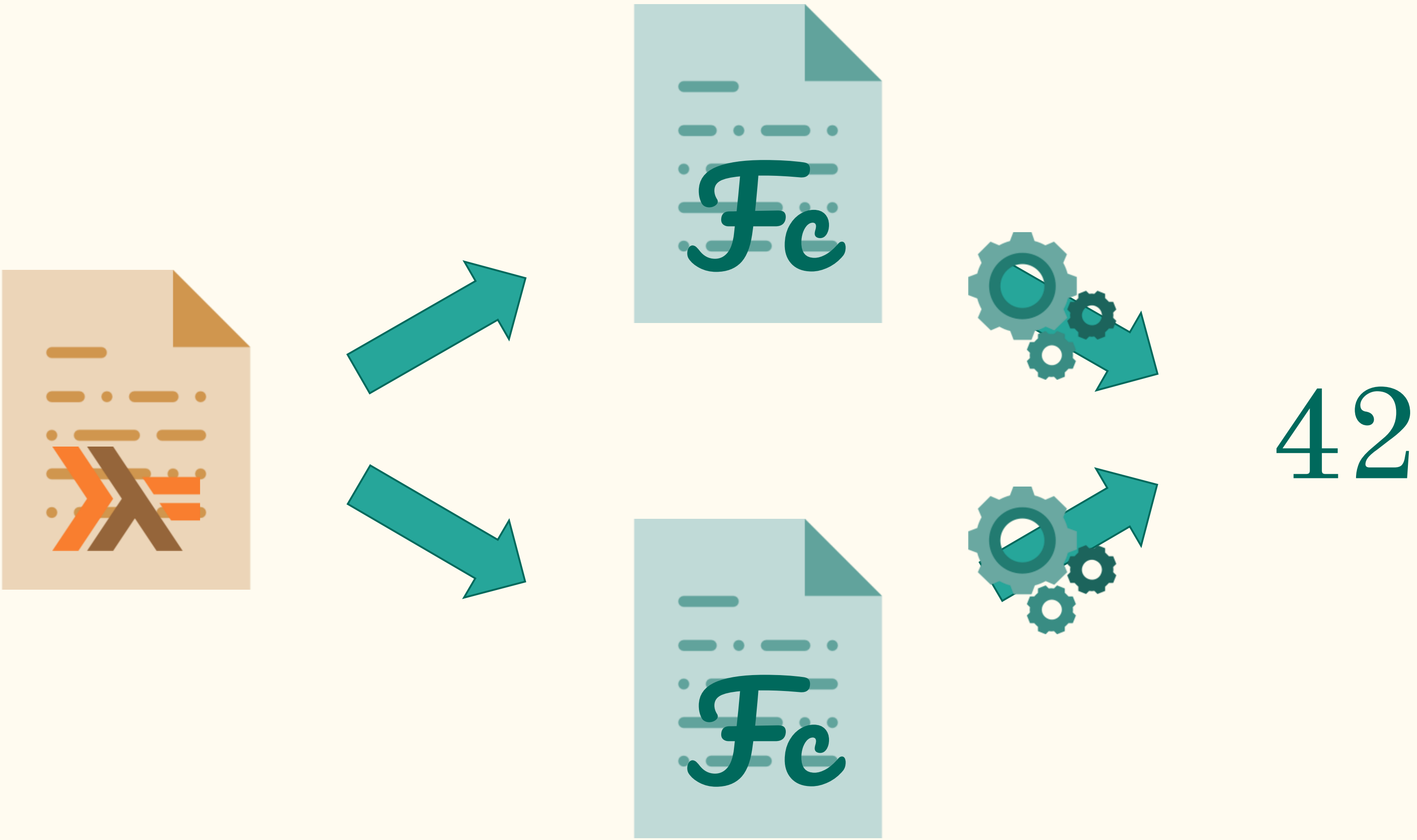


Uncharted Territory

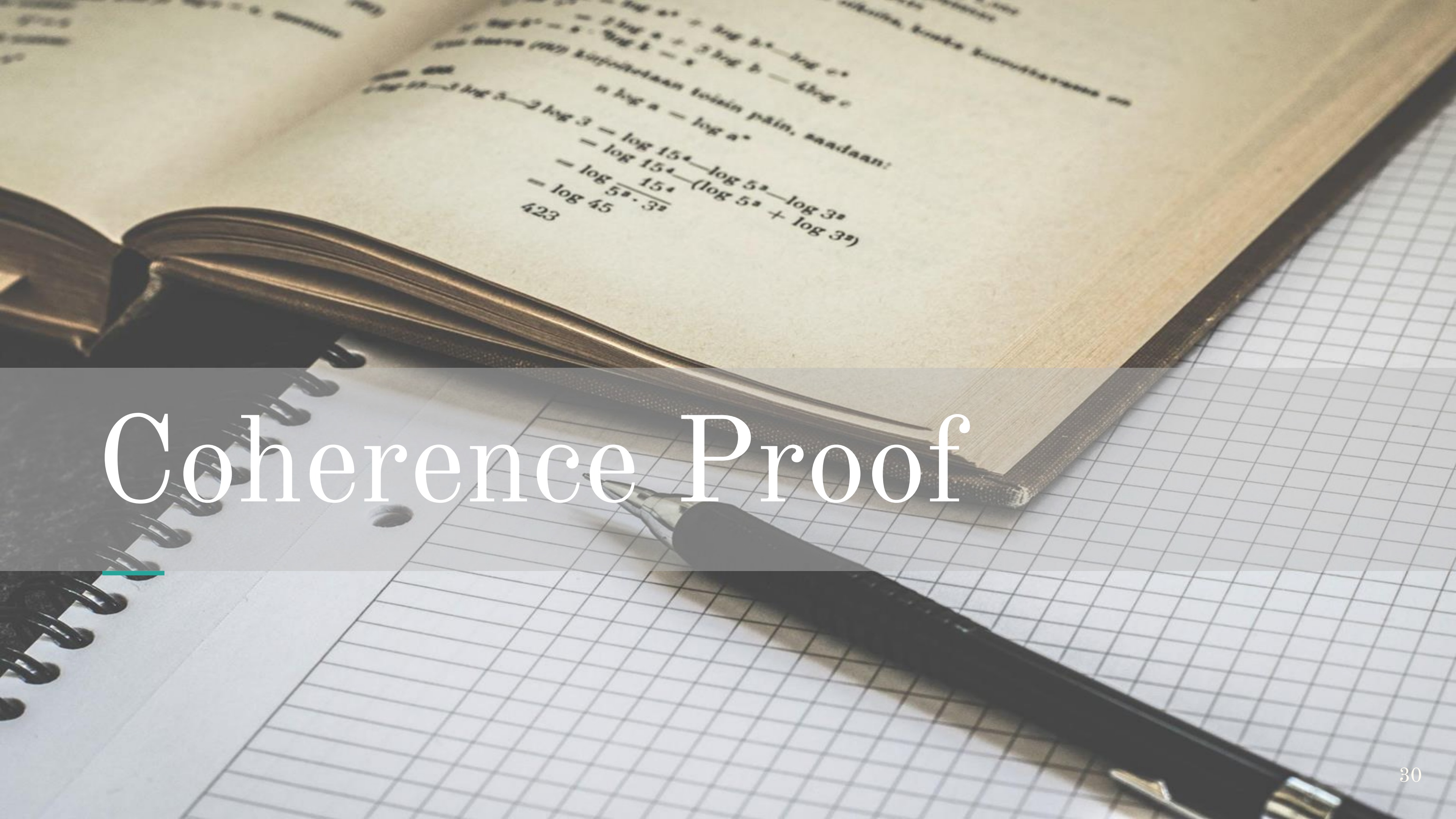

Execution
Fc
Type checking
Optimization




CHAOS MONKEY!

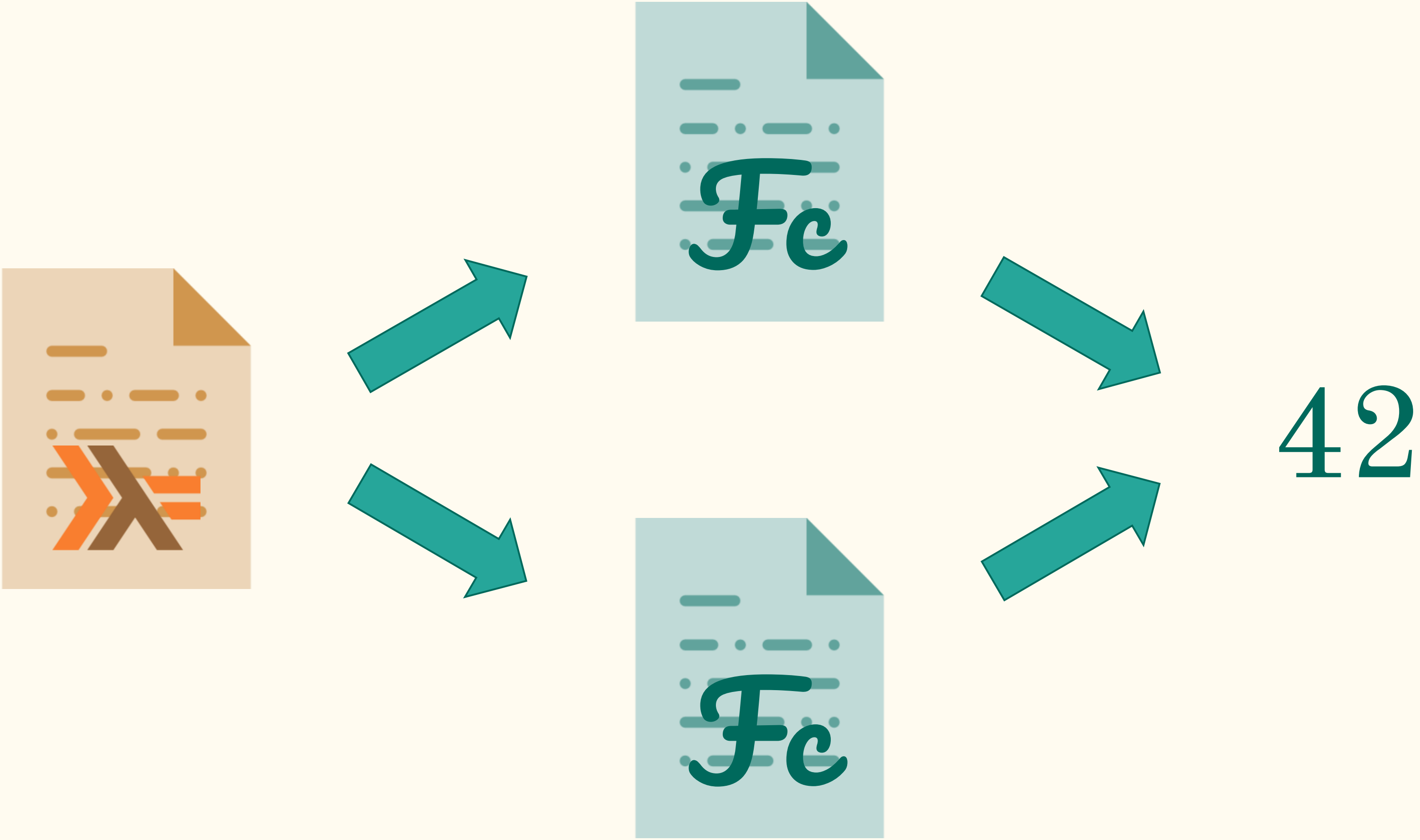

Fc
Fc
42

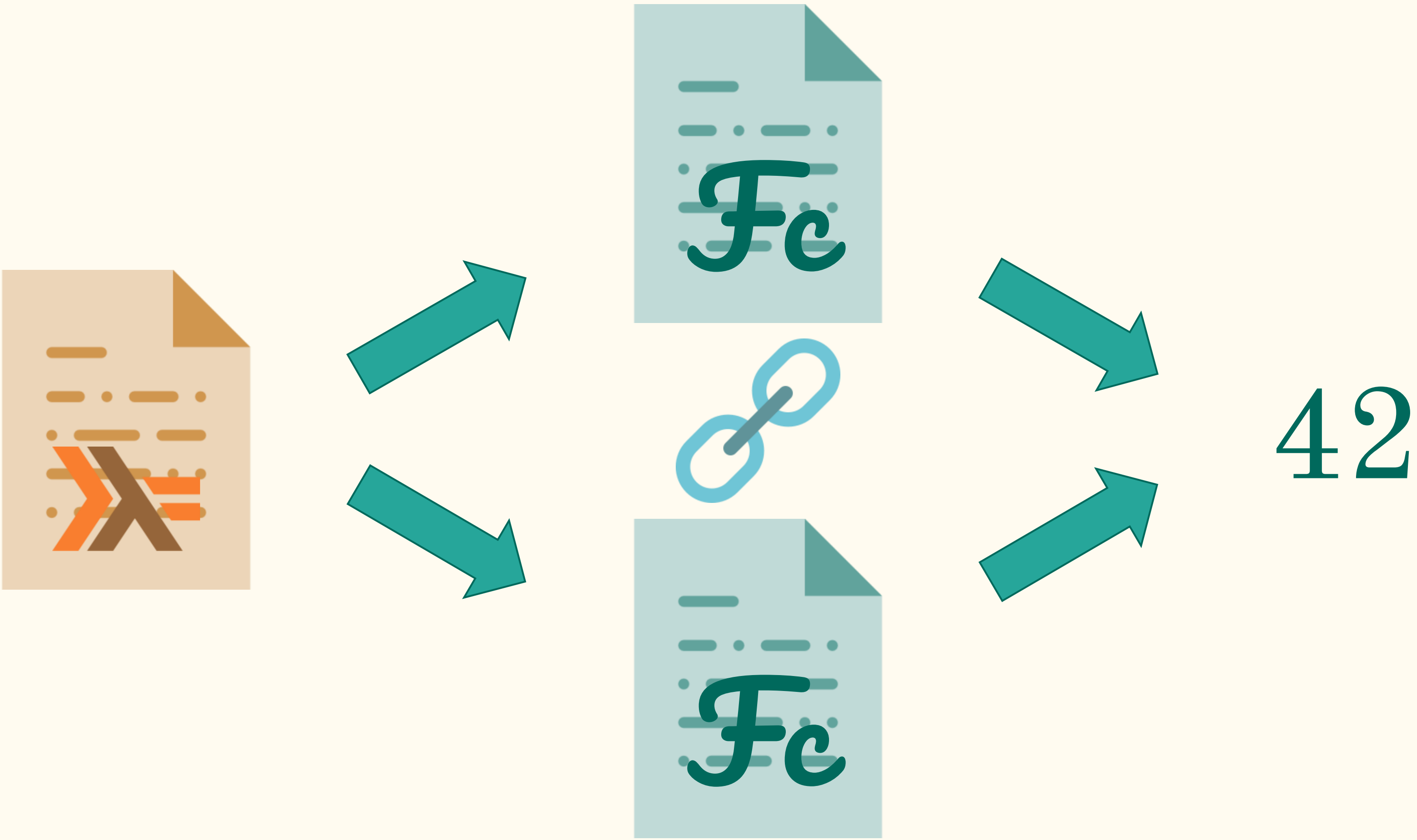
Coherence



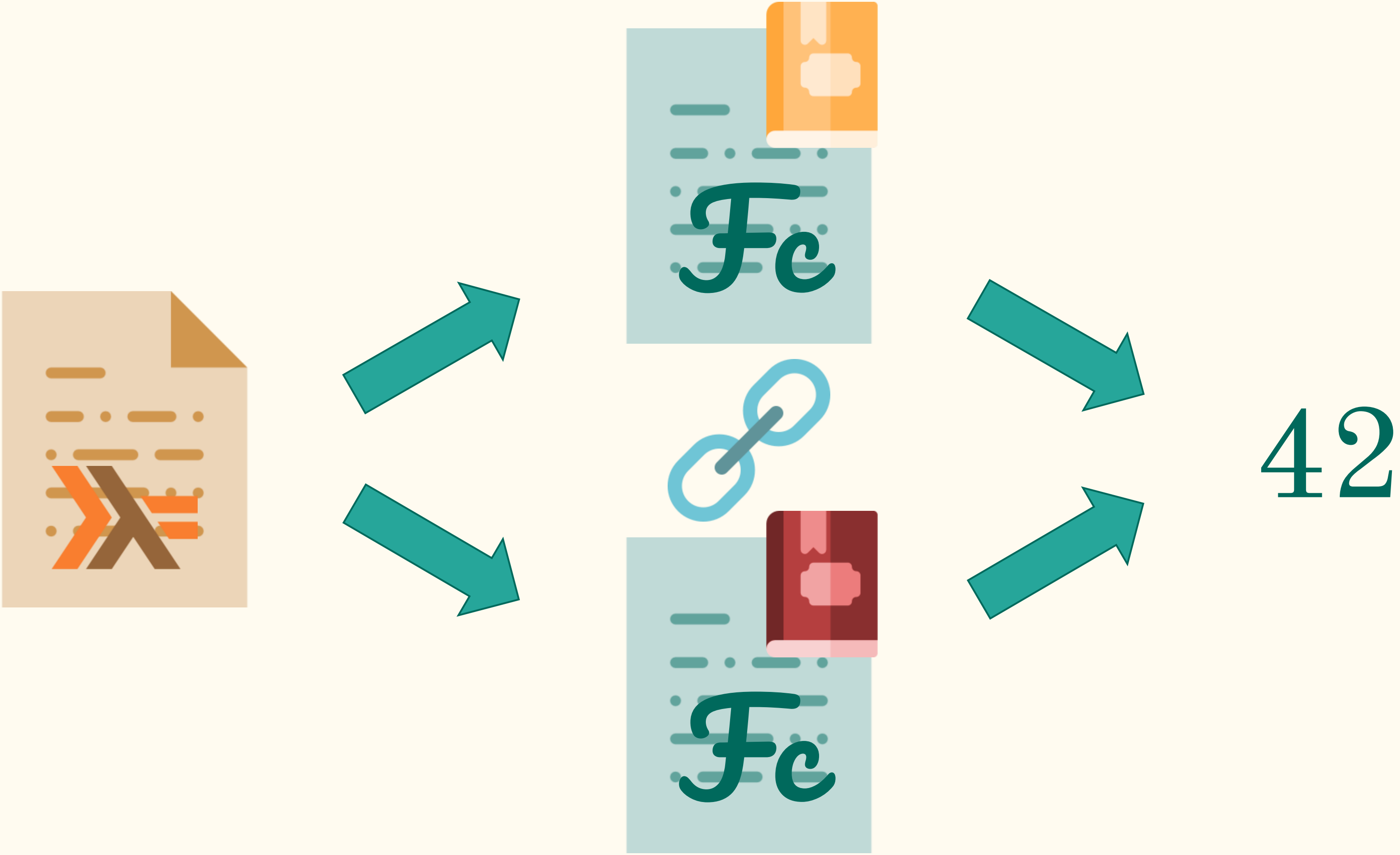
Coherence Proof

Proof through Logical Relations

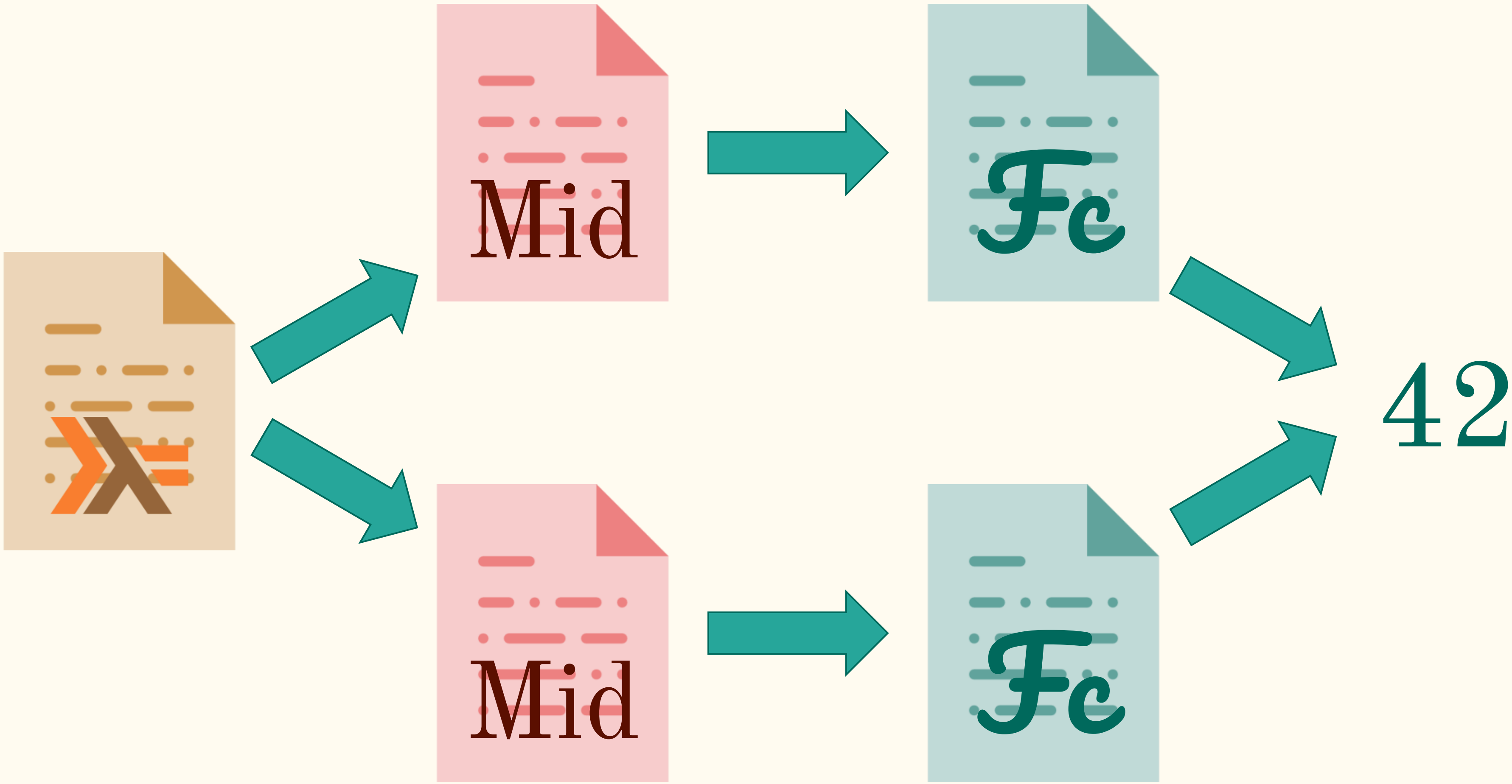

42

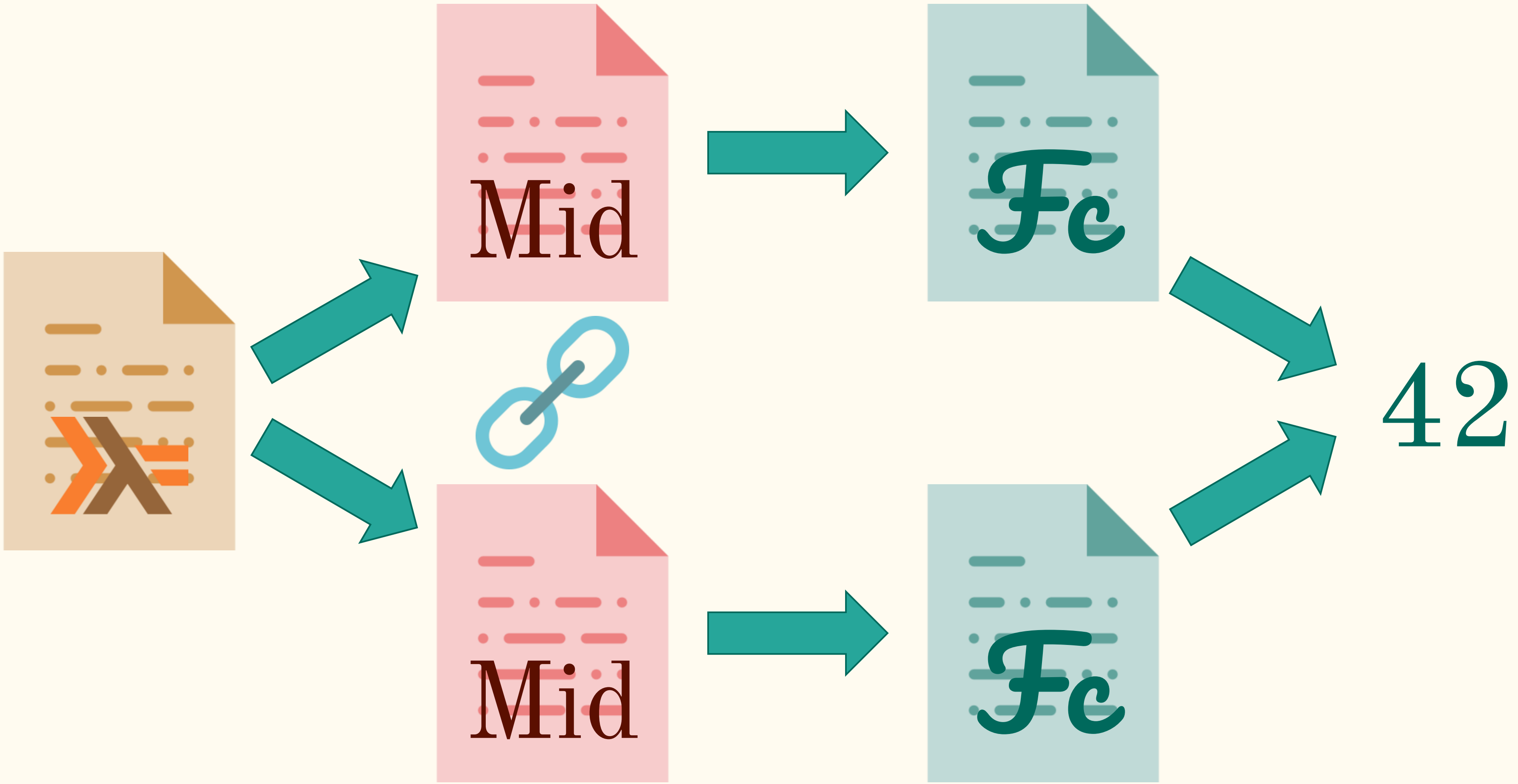


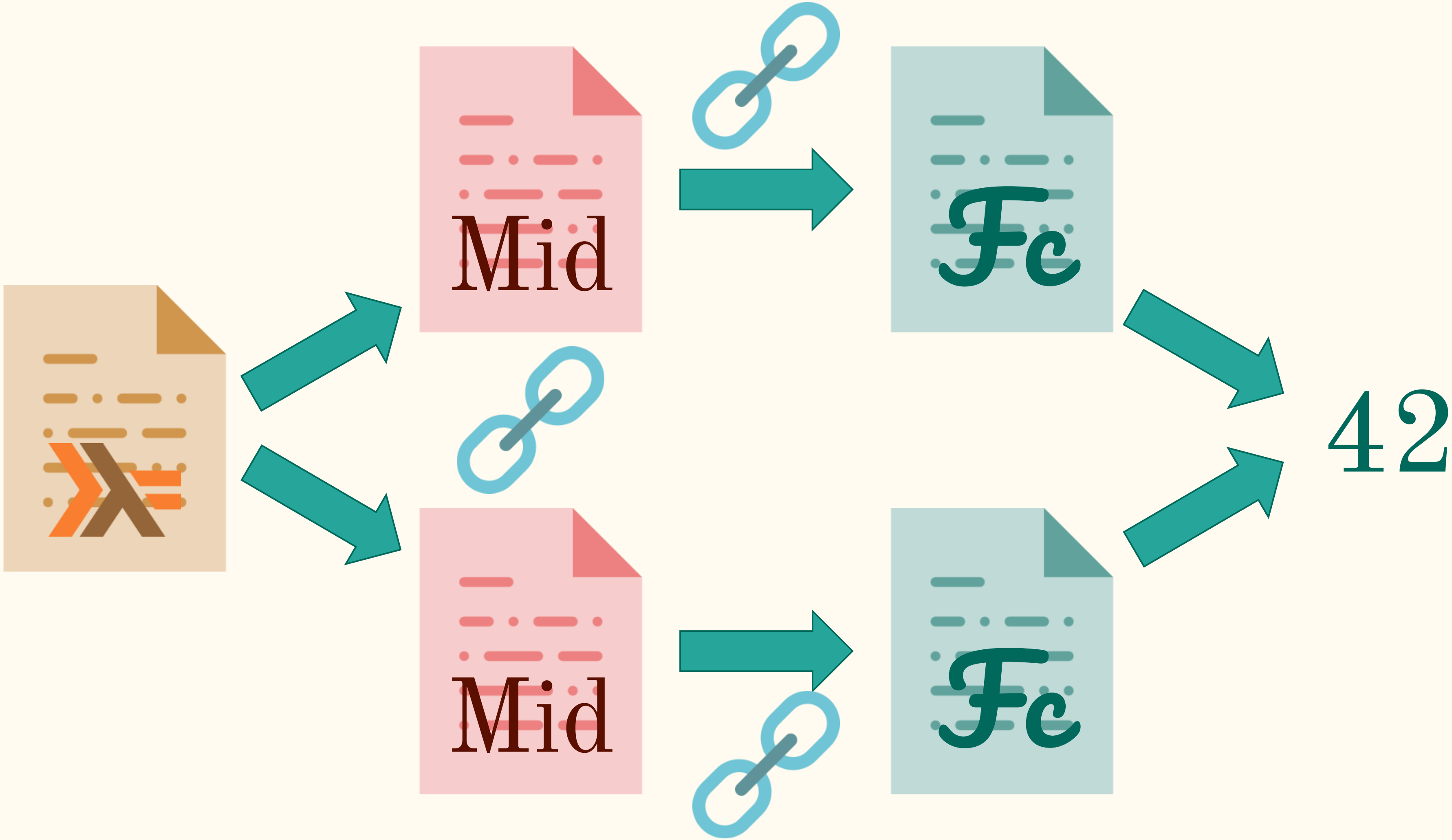
Fc
Fc
42

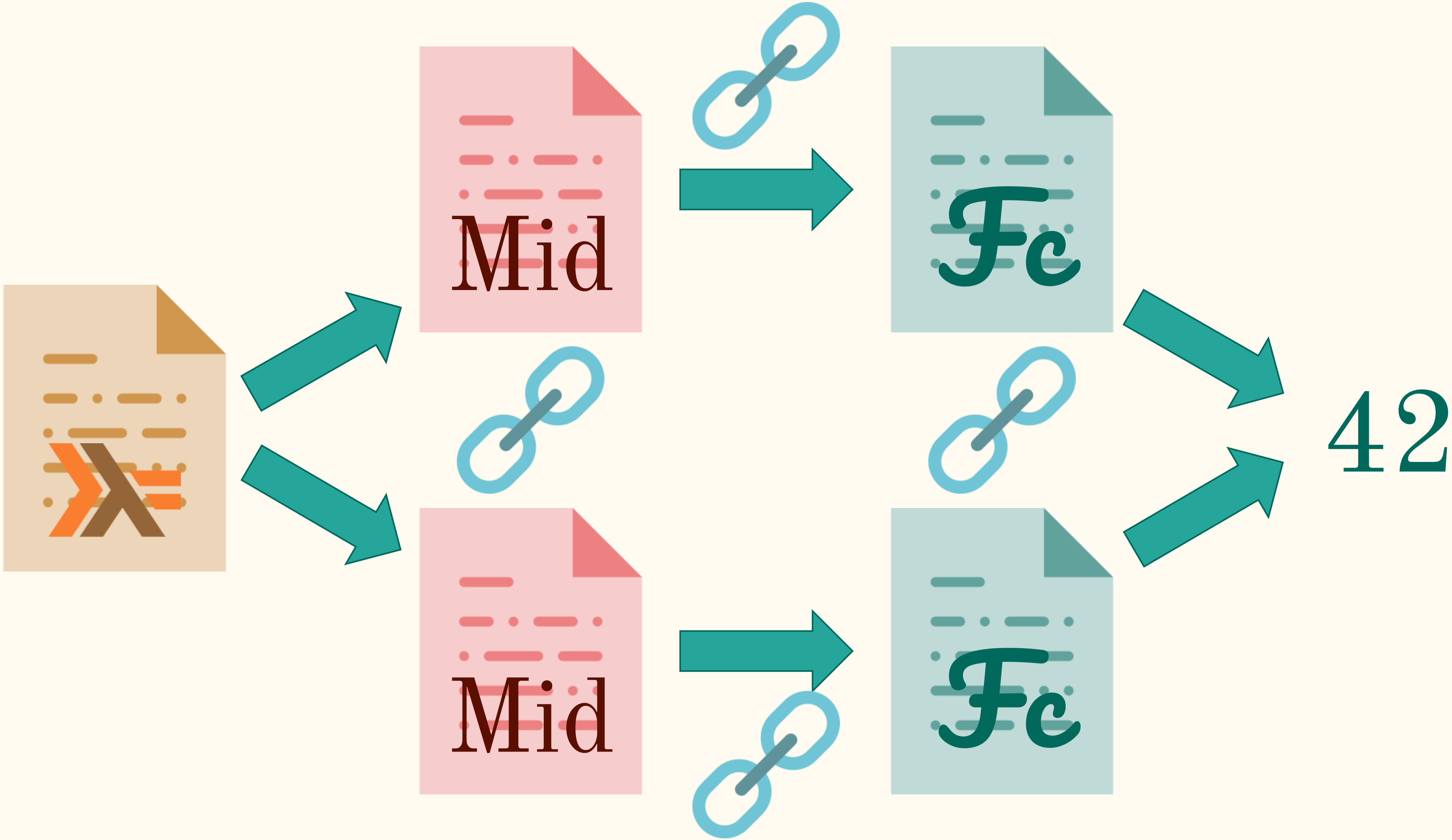


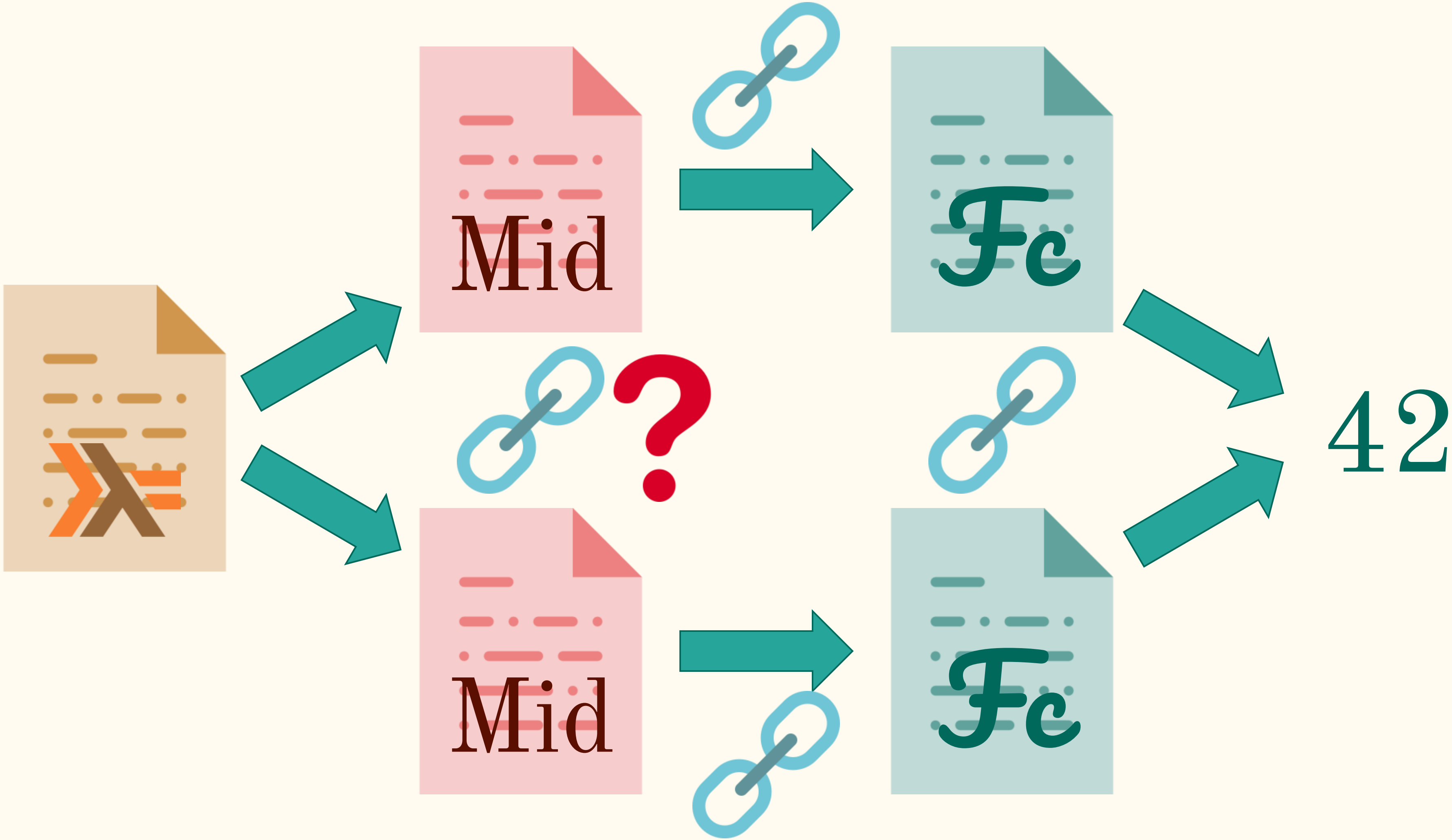
42


Mid
Mid
Fc
Fc
42

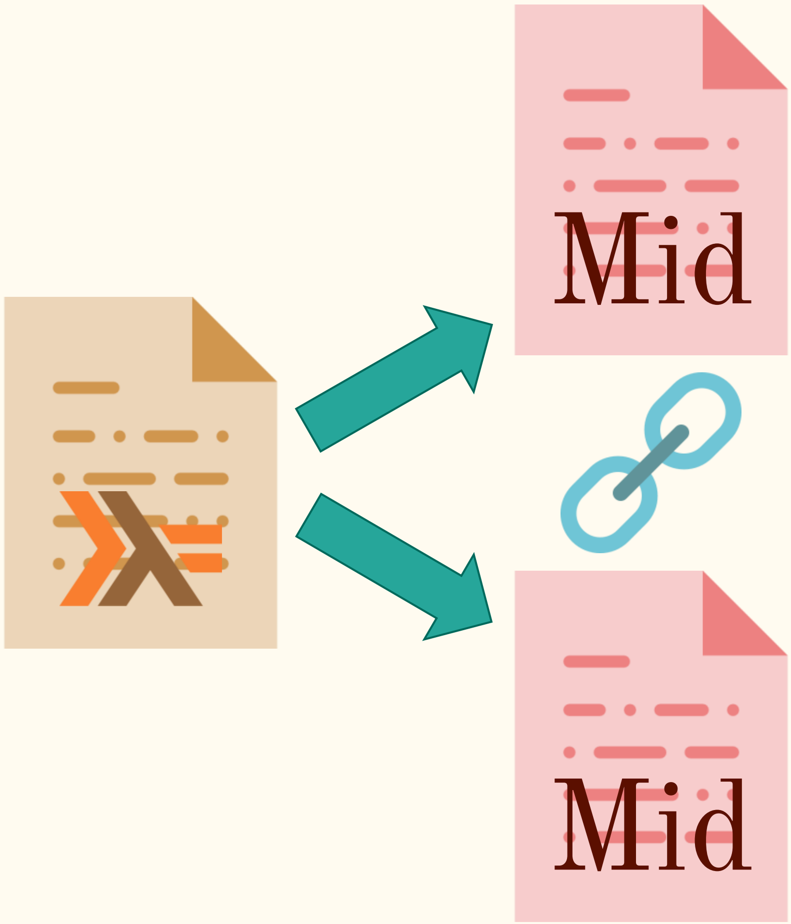

Mid
Mid
Fc
Fc
42

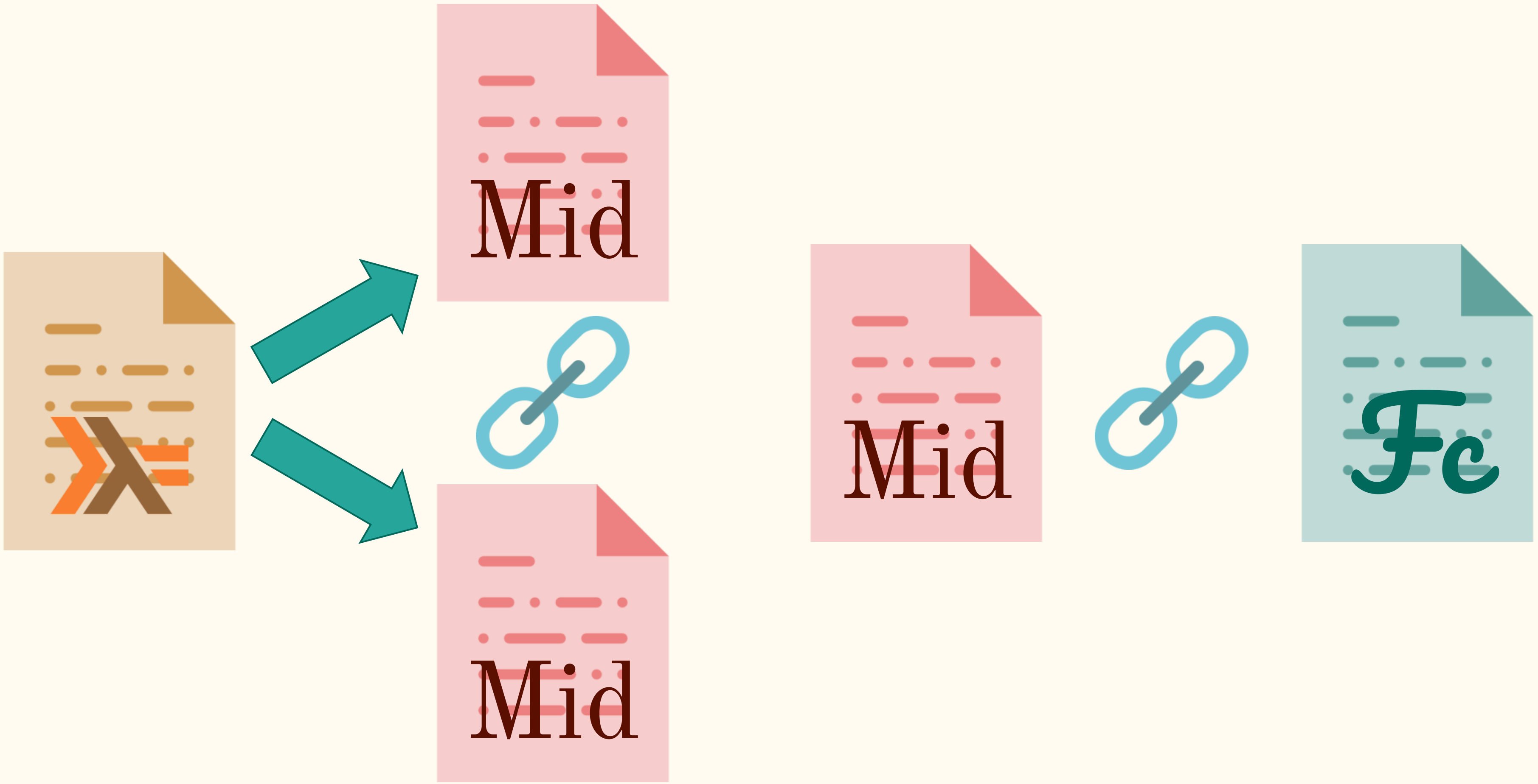

Mid
Mid
Fc
Fc
42


Mid
Fc
Mid
Fc
42


Mid
Mid
Fc
Fc
42

Intermediate Language


Mid
Mid


Mid
Mid
Mid
Fc


Mid


Fc

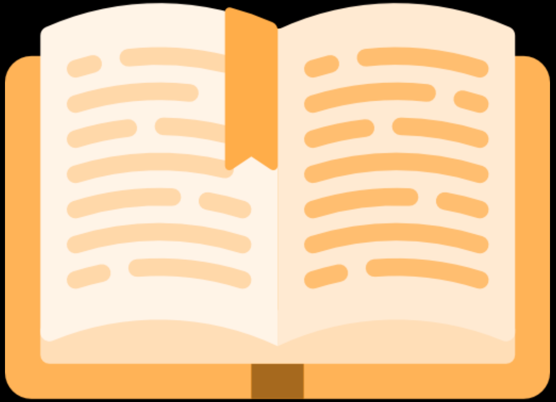

Mid

Fc

Dict

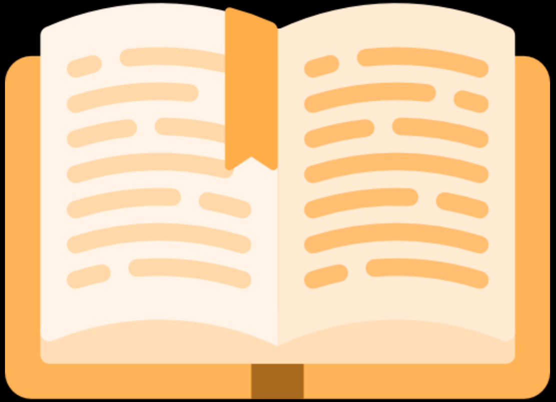

Mid


Fc

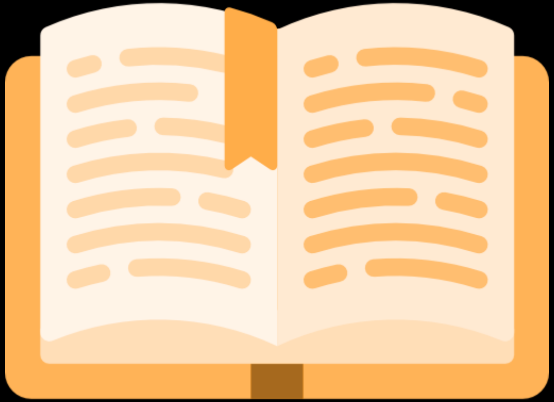



Mid

Fc



Pointer	Implem.
	…
	…
	…



Fundamental Theorem of Software Engineering

We can solve any problem by introducing an extra level of indirection.

```
(==) Bool
```


```
True True
```

`(==) Bool`  `True True`

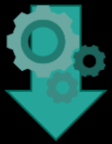
`(==) Bool` 

`True True`

(==) Bool  True True

(==) Bool 

True True

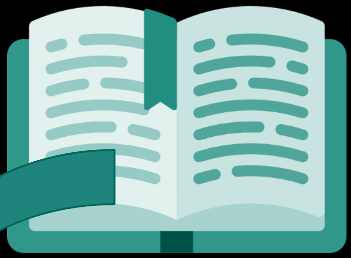


(==) Bool  True True

(==) Bool 

True True

(==) Bool  True True

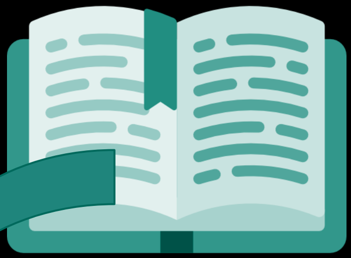


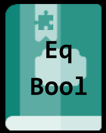
(==) Bool 

True True

```
(\x : Bool. \y : Bool. case x,y of
   True ,True  -> True
   False,False -> True
   _    ,_     -> False)
True True
```

(==) Bool  True True



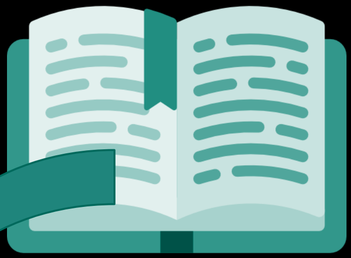
(==) Bool 

True True

```
(\x : Bool. \y : Bool. case x,y of
  True ,True  -> True
  False,False -> True
  _    ,_     -> False)
True True
```

Pointer	Implem.
	...

(==) Bool  True True



(==) Bool 

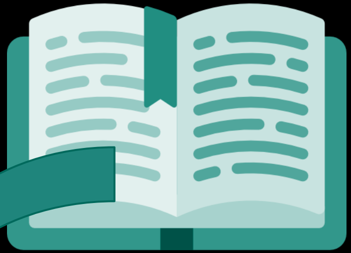
True True

```
(\x : Bool. \y : Bool. case x,y of
   True ,True  -> True
   False,False -> True
   _    ,_     -> False)
True True
```

Pointer	Implem.
	...

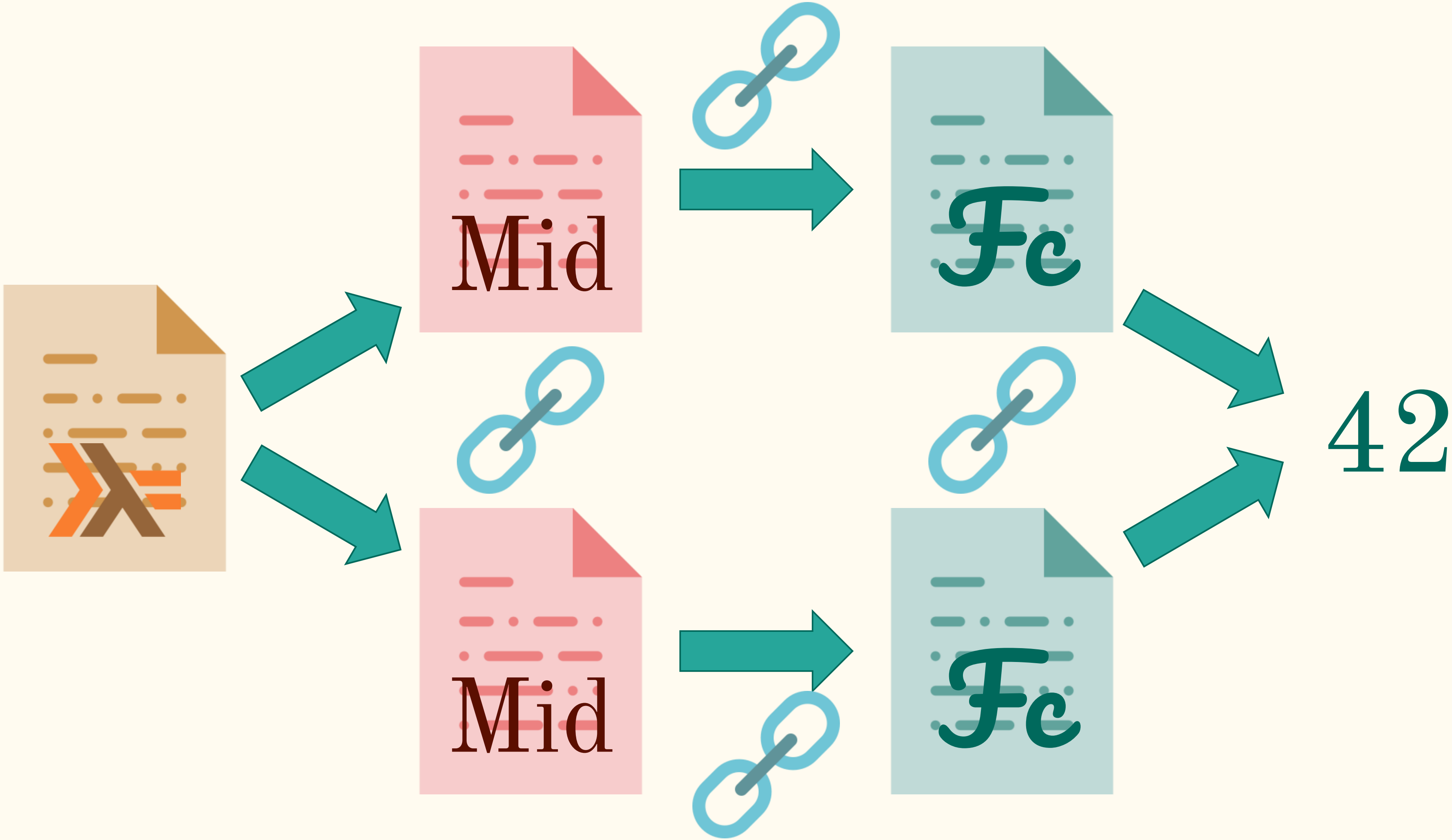
(==) Bool  True True

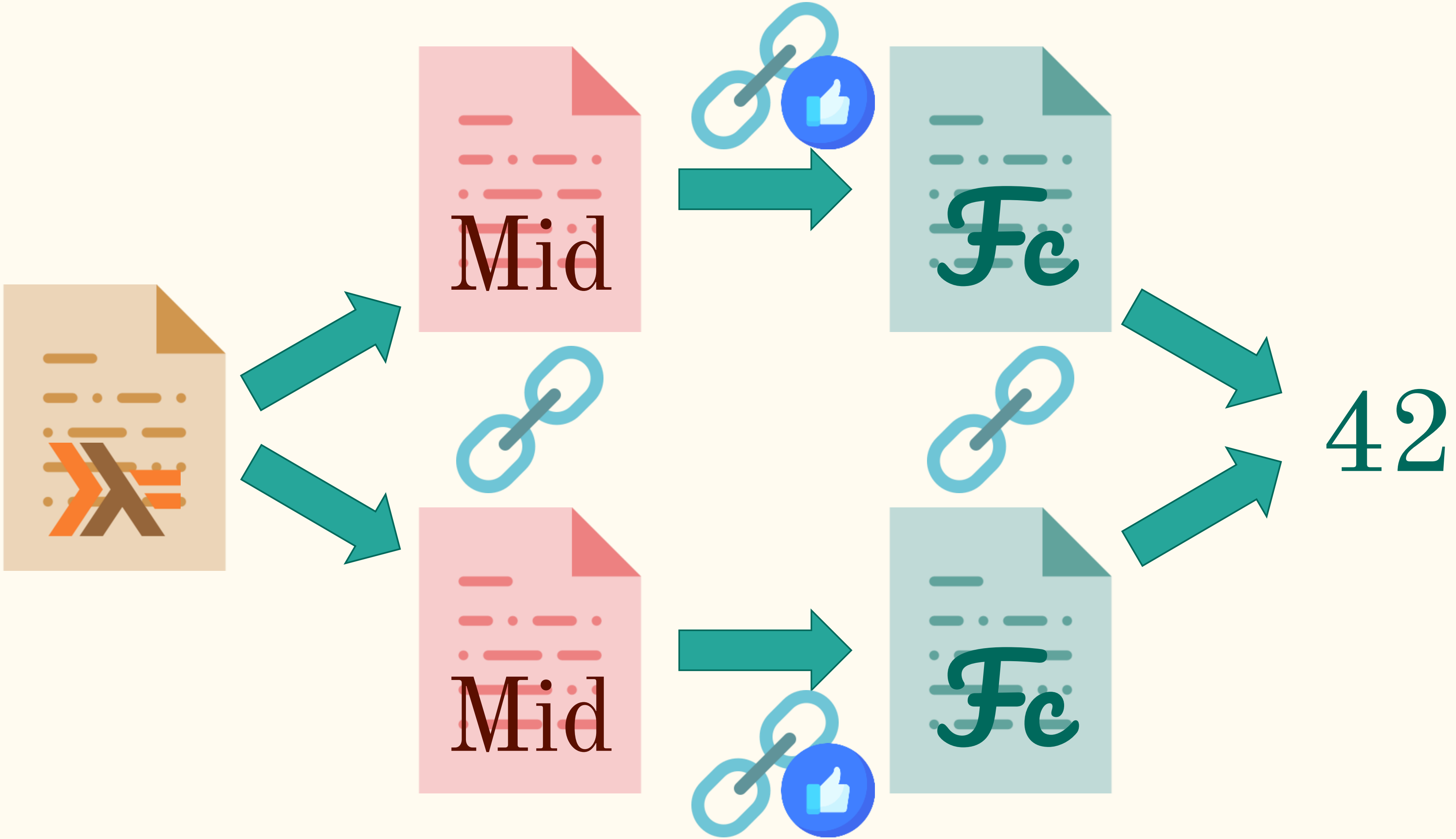
```
(\x : Bool. \y : Bool. case x,y of
  True ,True  -> True
  False,False -> True
  _    ,_     -> False)
True True
```



(==) Bool  True True

```
(\x : Bool. \y : Bool. case x,y of
  True ,True  -> True
  False,False -> True
  _    ,_     -> False)
True True
```


Mid
Mid
Fc
Fc
42


Mid
Mid
Fc
Fc
42

```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

not :: Eq Bool => Bool -> Bool
not n = n == False
```

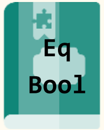
```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


not :: Eq Bool => Bool -> Bool
not n = n == False
```

```
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


not :: Eq Bool => Bool -> Bool
not n = n == False
```

```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

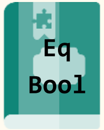

```haskell
not :: Eq Bool => Bool -> Bool
not n = n == False
```

```
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```




```
not :: Eq Bool => Bool -> Bool
not n = n == False
```

```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```




```haskell
not :: Eq Bool => Bool -> Bool
not n = n == False
```

¯_(ツ)_/¯

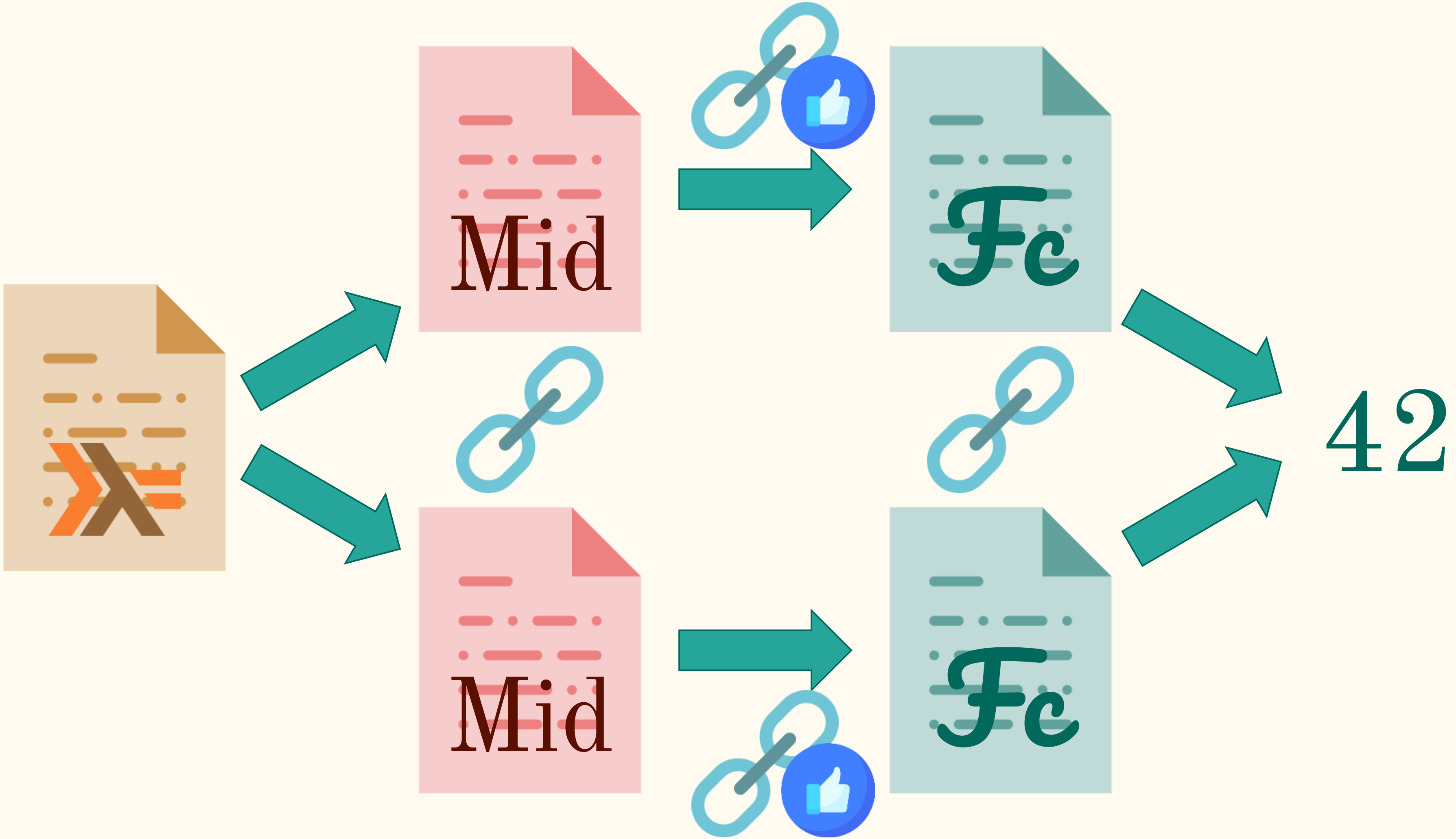
```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


not :: Eq Bool => Bool -> Bool
not n = n == False
```

```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


not :: Eq Bool => Bool -> Bool
not n = n == False
```

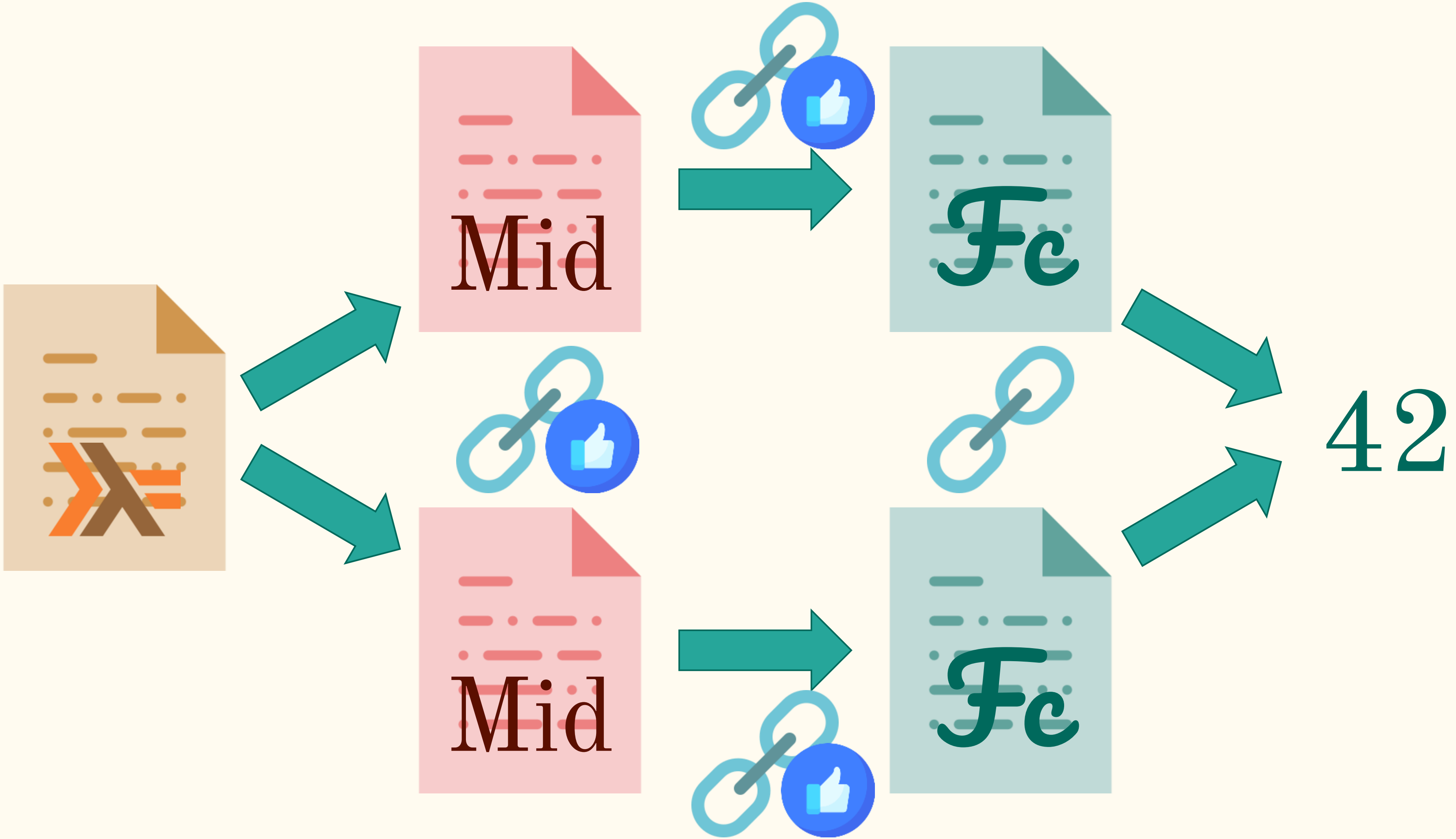
Pointer	Implem.
	Eq Bool : ...

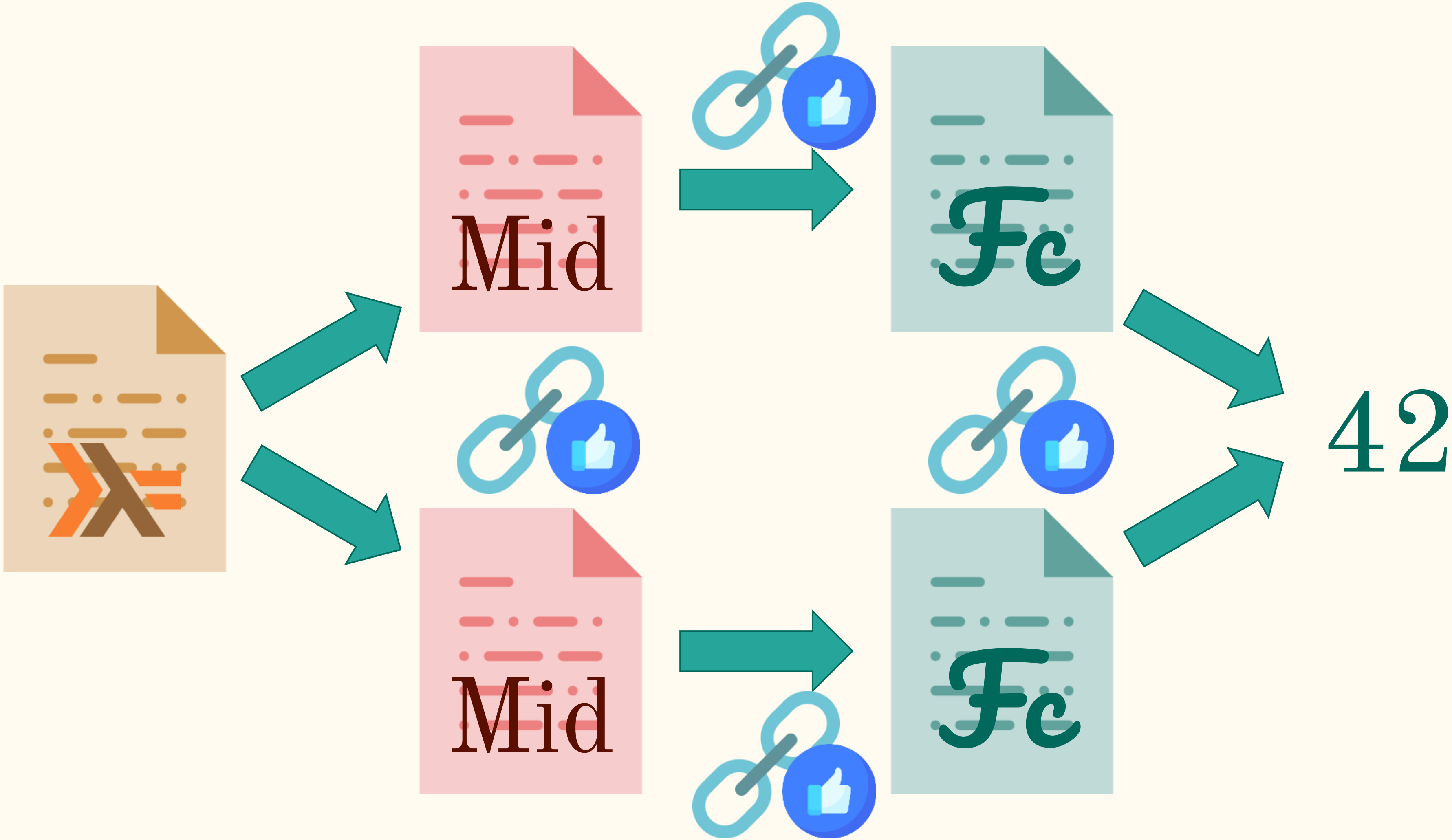
```haskell
{-# LANGUAGE FlexibleContexts #-}

inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


not :: Eq Bool => Bool -> Bool
not n = n == False
```

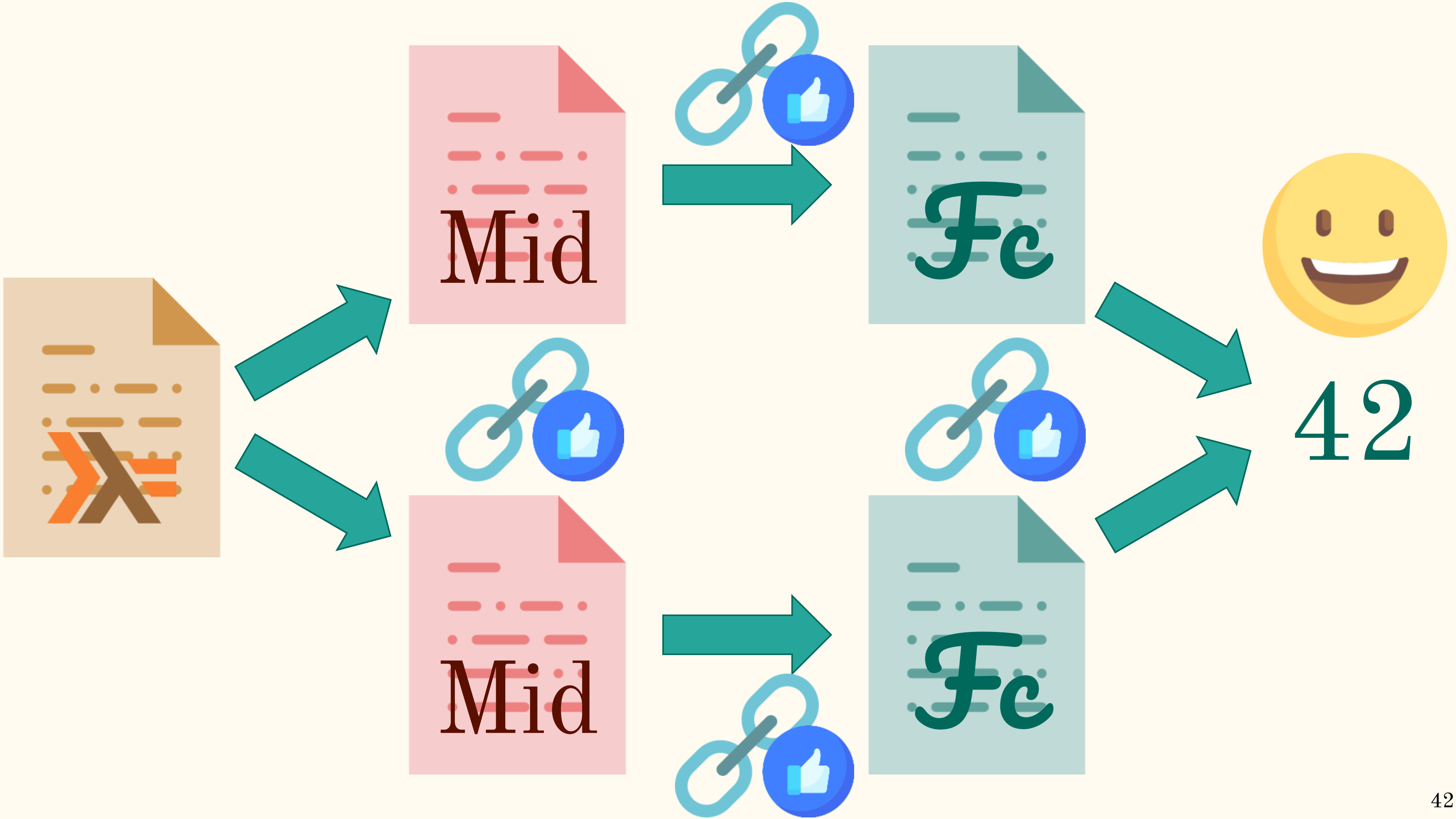
Pointer	Implem.
	Eq Bool : ...



Mid
Mid
Fc
Fc
42


Mid
Fc
Mid
Fc
42


Mid
Mid
Fc
Fc
42


Mid
Mid
Fc
Fc
42

Future Work

Image by Arkarti from DeviantArt

From the rule premise:

$$M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma', x : \sigma_1 \Rightarrow \sigma') \rightsquigarrow M' \tag{193}$$

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \tag{194}$$

$$\Gamma_C;\Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \tag{195}$$

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 induction hypothesis and Equation 193, it follows that:

$$\Sigma;\Gamma_C;\Gamma', x : \sigma_1 \vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e] \tag{196}$$

The goal follows from iTm-let, in combination with Equations 194, 195 and 196.

□

Theorem 18 (Strong Normalization).
If $\Sigma;\Gamma_C;\bullet \vdash_{tm} e : \sigma$ *then* $\exists v : \Sigma \vdash e \longrightarrow^* v$.

Proof. By Theorem 19 and 20, with $R^{SN} = \bullet$, $\phi^{SN} = \bullet$, $\gamma^{SN} = \bullet$, since $\Gamma = \bullet$. □

Lemma 35 (Well Typedness from Strong Normalization).
$e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $\Sigma;\Gamma_C;\bullet \vdash_{tm} e : R^{SN}(\sigma)$

Proof. The goal is baked into the relation. It follows by simple induction on σ. □

Lemma 36 (Strong Normalization preserved by forward/backward reduction).
Suppose $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : R^{SN}(\sigma)$, *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.

Bool

$$\begin{aligned} e_1 \in \mathcal{SN}[\![Bool]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq\ & \Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : Bool \\ & \wedge \exists v : \Sigma \vdash e_1 \longrightarrow^* v \end{aligned}$$

By Preservation (Theorem 8), we know that $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_2 : Bool$. Because the evaluation process is deterministic, given $\Sigma \vdash e_1 \longrightarrow^* v$, we have $\Sigma \vdash e_2 \longrightarrow^* v$.

Type variable

$$\begin{aligned} e_1 \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq\ & \Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : {R^{SN}}_1(a) \\ & \wedge \exists v : \Sigma \vdash e_1 \longrightarrow^* v \\ & \wedge v \in {R^{SN}}_2(a) \end{aligned}$$

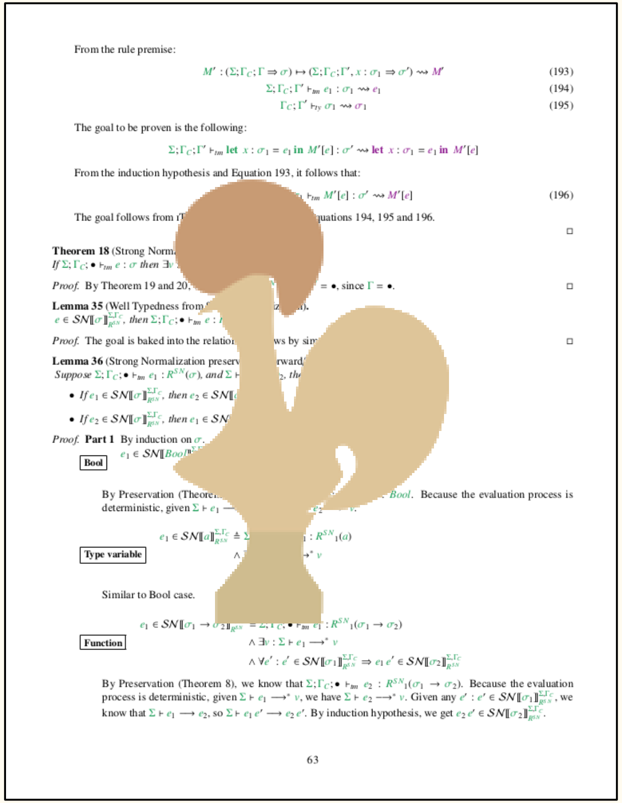
Similar to Bool case.

Function

$$\begin{aligned} e_1 \in \mathcal{SN}[\![\sigma_1 \to \sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq\ & \Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : {R^{SN}}_1(\sigma_1 \to \sigma_2) \\ & \wedge \exists v : \Sigma \vdash e_1 \longrightarrow^* v \\ & \wedge \forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e_1\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \end{aligned}$$

By Preservation (Theorem 8), we know that $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_2 : {R^{SN}}_1(\sigma_1 \to \sigma_2)$. Because the evaluation process is deterministic, 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 $e_2\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

63


From the rule premise:

$$M' : (\Sigma;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma;\Gamma_C;\Gamma', x : \sigma_1 \Rightarrow \sigma') \rightsquigarrow M' \quad (193)$$

$$\Sigma;\Gamma_C;\Gamma' \vdash_{tm} e_1 : \sigma_1 \rightsquigarrow e_1 \quad (194)$$

$$\Gamma_C;\Gamma' \vdash_{ty} \sigma_1 \rightsquigarrow \sigma_1 \quad (195)$$

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 induction hypothesis and Equation 193, it follows that:

$$\vdash_{tm} M'[e] : \sigma' \rightsquigarrow M'[e] \quad (196)$$

The goal follows from quations 194, 195 and 196.

□

Theorem 18 (Strong Norma
If $\Sigma;\Gamma_C;\bullet \vdash_{tm} e : \sigma$ *then* $\exists v$

Proof. By Theorem 19 and 20, $= \bullet$, since $\Gamma = \bullet$. □

Lemma 35 (Well Typedness from).
$e \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $\Sigma;\Gamma_C;\bullet \vdash_{tm} e :$

Proof. The goal is baked into the relatio ws by sim □

Lemma 36 (Strong Normalization preserv rward
Suppose $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : R^{SN}(\sigma)$, *and* $\Sigma \vdash$ $_2$, *the*

- *If* $e_1 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $e_2 \in \mathcal{SN}[\![$
- *If* $e_2 \in \mathcal{SN}[\![\sigma]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$, *then* $e_1 \in \mathcal{SN}$

Proof. **Part 1** By induction on σ.

Bool $e_1 \in \mathcal{SN}[\![Bool]\!]$

By Preservation (Theore $Bool$. Because the evaluation process is deterministic, given $\Sigma \vdash e_1$ e_2 v.

Type variable $e_1 \in \mathcal{SN}[\![a]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \triangleq \Sigma$ $: R^{SN}{}_1(a)$ $\wedge$ $\rightarrow^* v$

Similar to Bool case.

Function

$$e_1 \in \mathcal{SN}[\![\sigma_1 \rightarrow \sigma_2]\!]_{R^{SN}} \triangleq \Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : R^{SN}{}_1(\sigma_1 \rightarrow \sigma_2)$$
$$\wedge\ \exists v : \Sigma \vdash e_1 \longrightarrow^* v$$
$$\wedge\ \forall e' : e' \in \mathcal{SN}[\![\sigma_1]\!]^{\Sigma,\Gamma_C}_{R^{SN}} \Rightarrow e_1\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$$

By Preservation (Theorem 8), we know that $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_2 : R^{SN}{}_1(\sigma_1 \rightarrow \sigma_2)$. Because the evaluation process is deterministic, 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 $e_2\, e' \in \mathcal{SN}[\![\sigma_2]\!]^{\Sigma,\Gamma_C}_{R^{SN}}$.

63

Questions?

References

Icons made by DinosoftLabs, Freepik, Smashicons & Vectors Market from www.flaticon.com