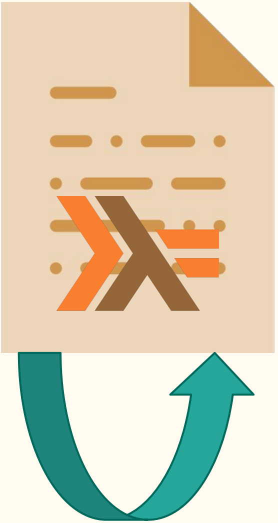
Proving Haskell's Type Class Resolution Coherent

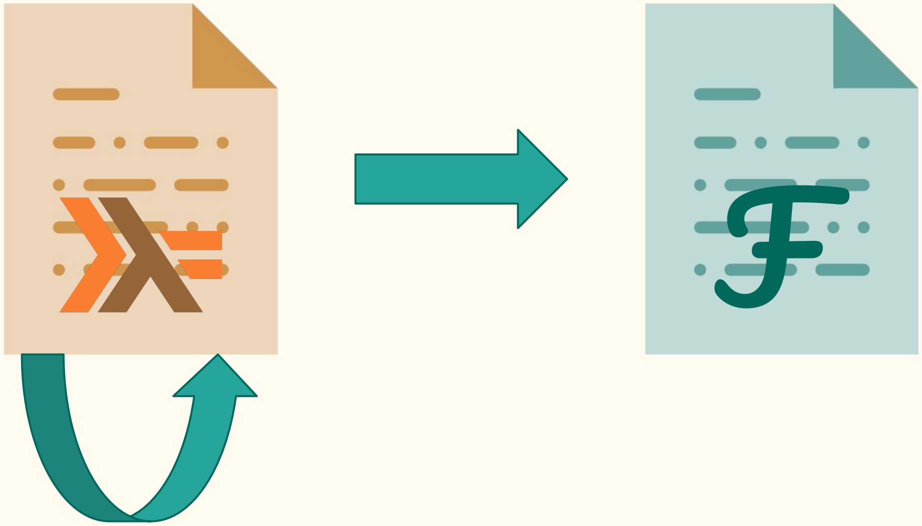
Gert-Jan Bottu, Ningning Xie,
Koar Marntirosian, Tom Schrijvers

KU LEUVEN

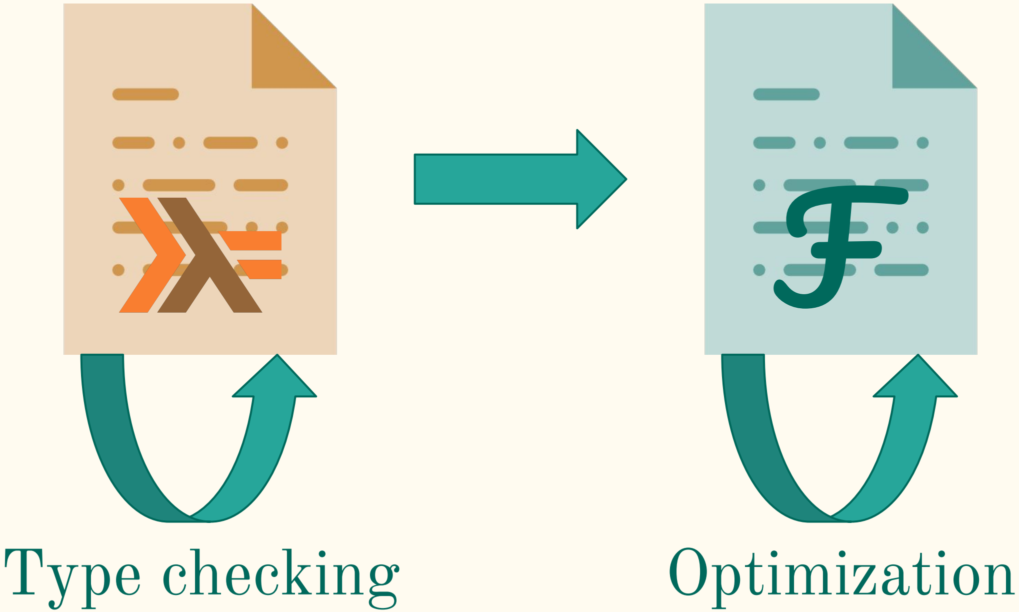
Background

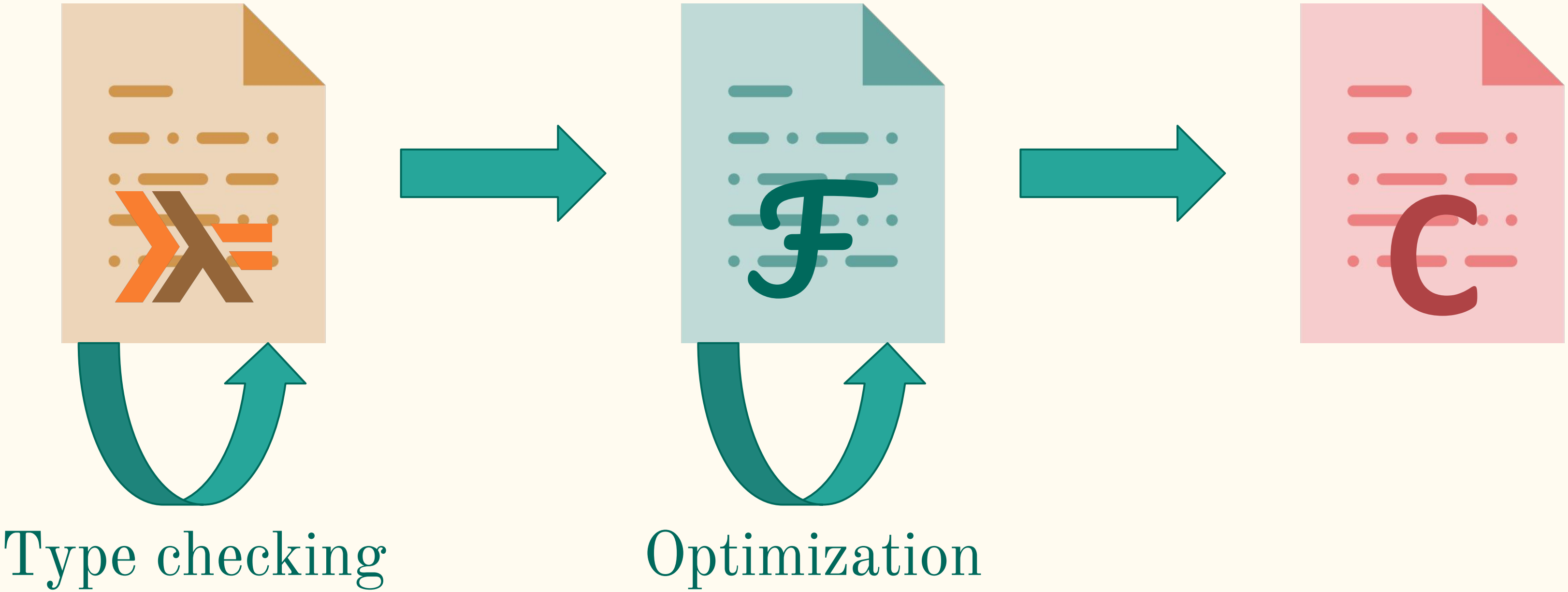


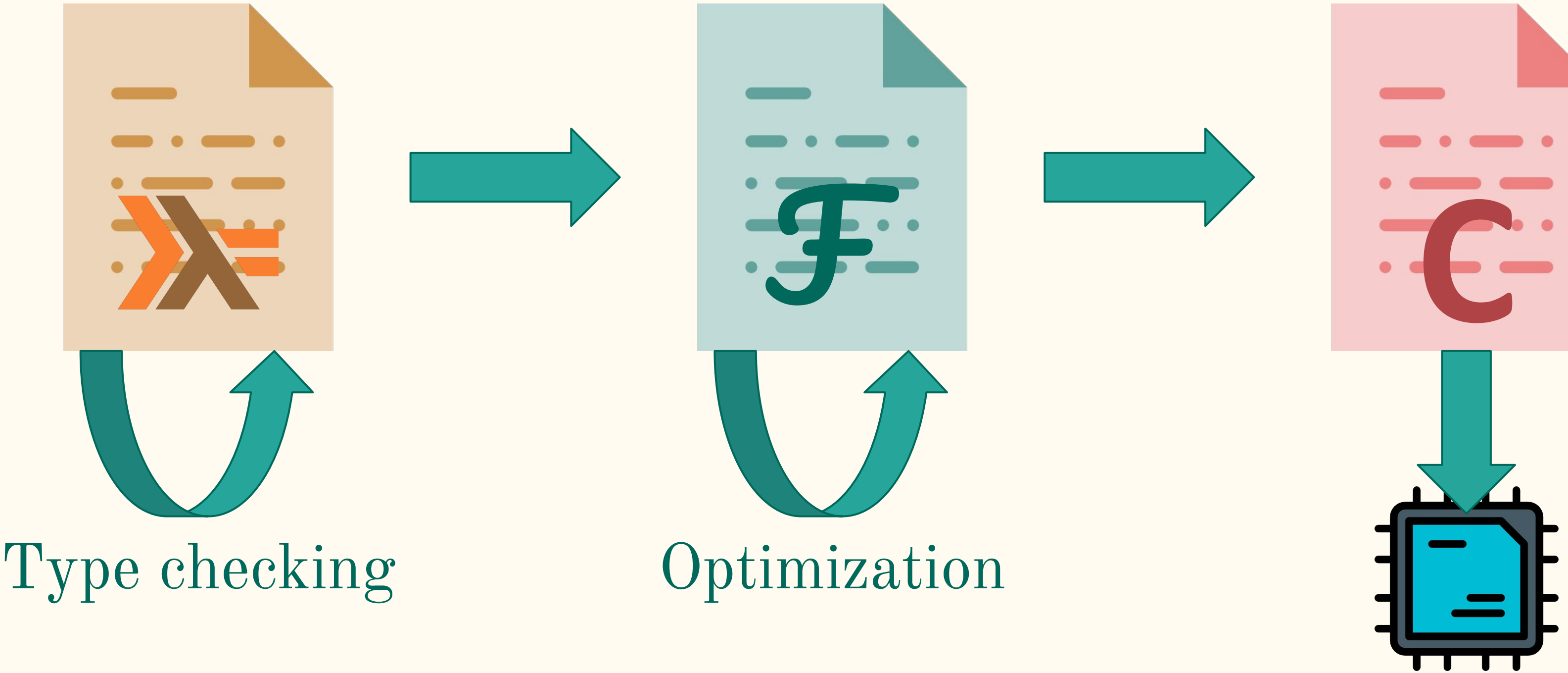
Type checking

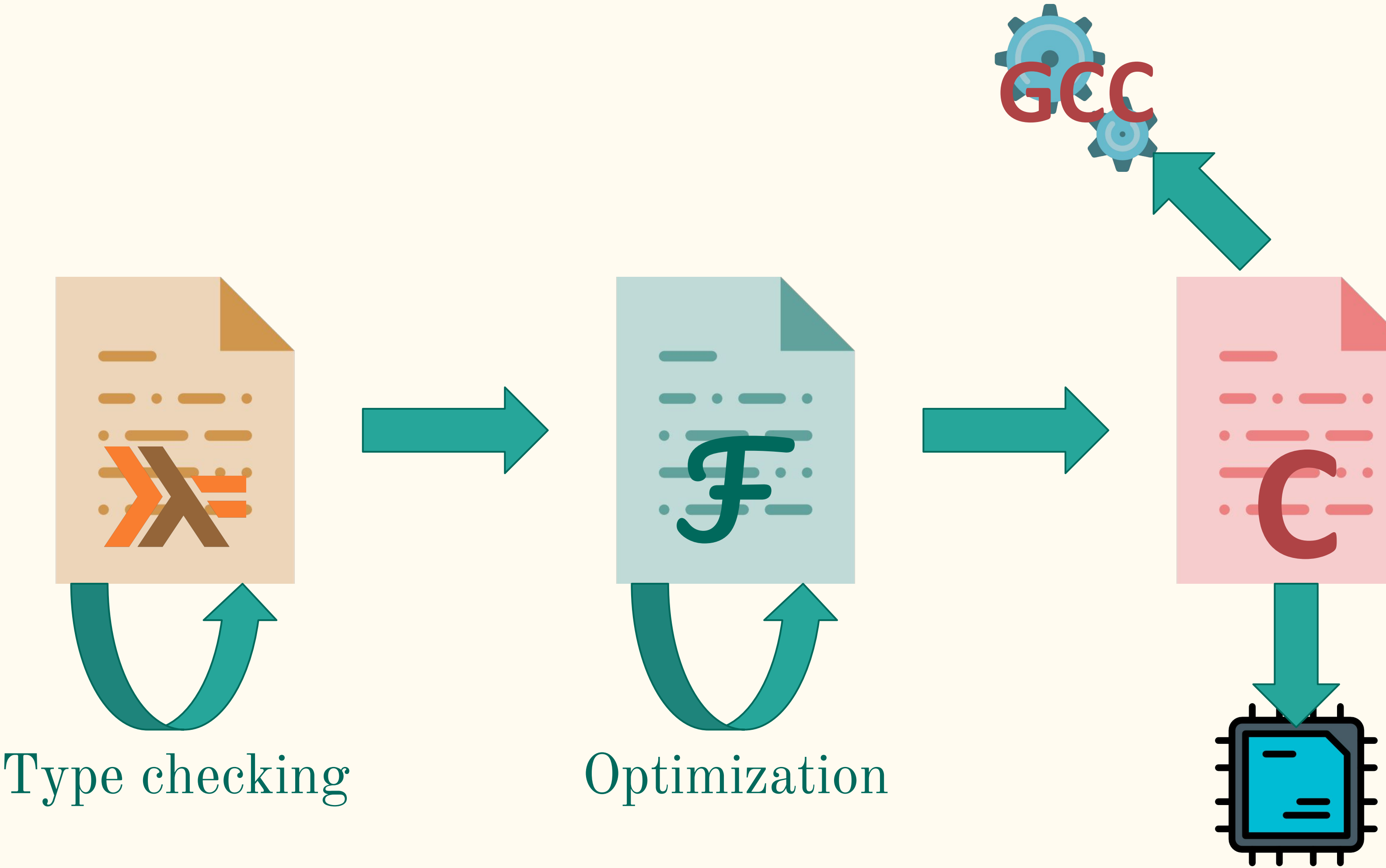


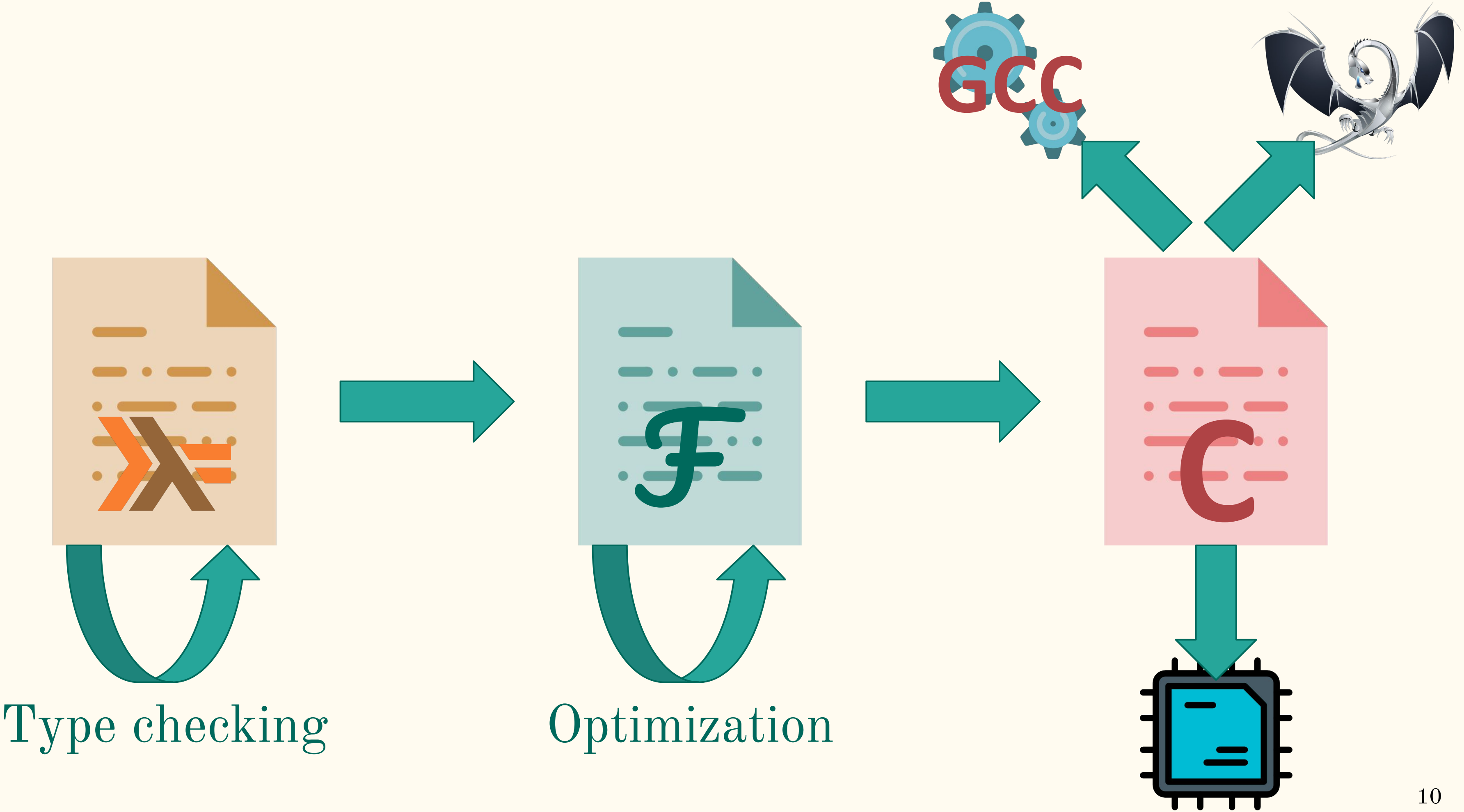
Type checking

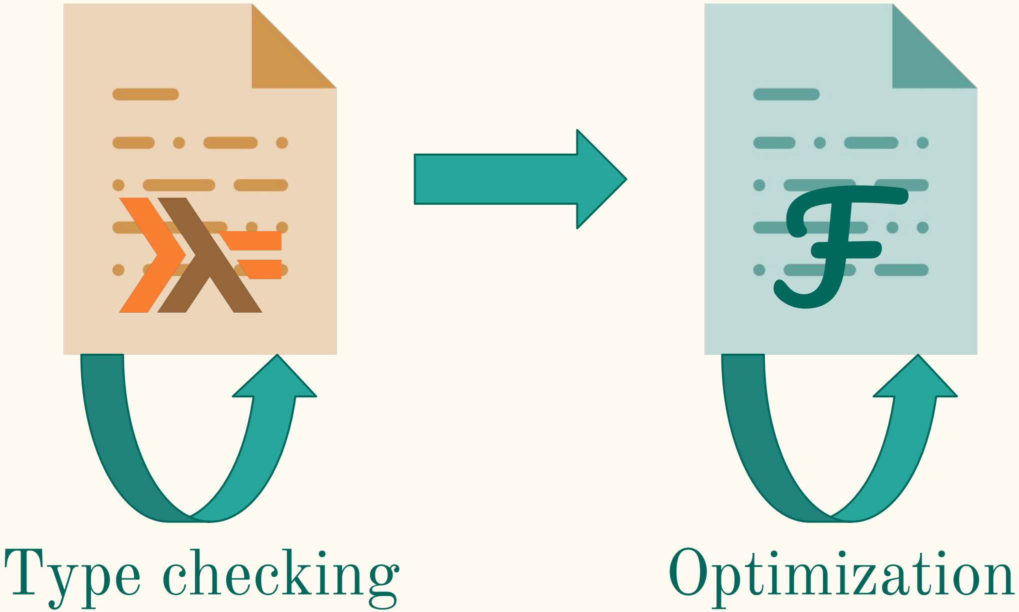

Type checking
Optimization

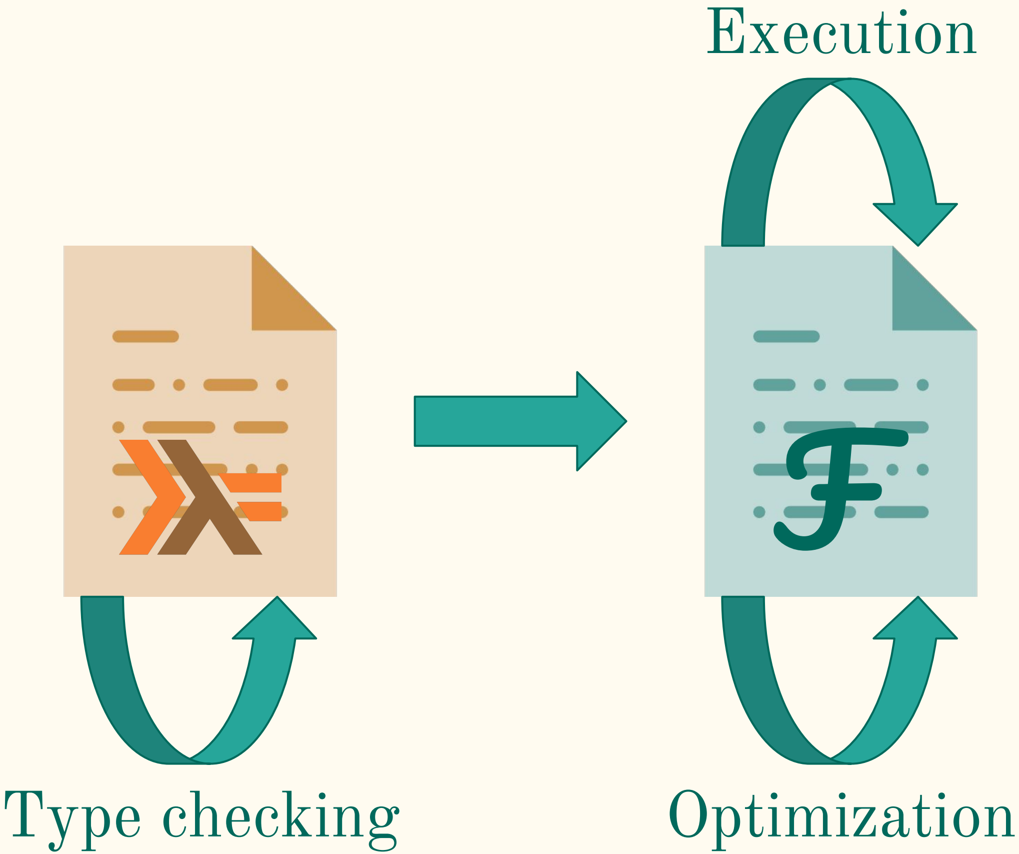

Type checking
Optimization


Type checking
Optimization


GCC
C
Type checking
Optimization


GCC
Type checking
Optimization


Type checking
Optimization


Execution
Type checking
Optimization

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

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

show :: Show a => a -> String
```

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

show :: Show a => a -> String

  > show 42
  "42"
```

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

show :: Show a => a -> String

  > show 42
  "42"

read :: Read a => String -> a
```

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

show :: Show a => a -> String

  > show 42
  "42"

read :: Read a => String -> a

  > (read "42") + 1
  43
```

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)
```

s

```
> foo "1"
```

a = ?

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

s read s

```
> foo "1"
```

a = ?

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

s   read s

1 (Int)

```
> foo "1"
```

a = ?

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

s

read s

1 (Int)

```
> foo "1"
```

1.0 (Float)

a = ?

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

s

read s

```
> foo "1"
```

1 (Int)

1.0 (Float)

True (Bool)

a = ?

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

s   read s   show (read s)

```
> foo "1"
```

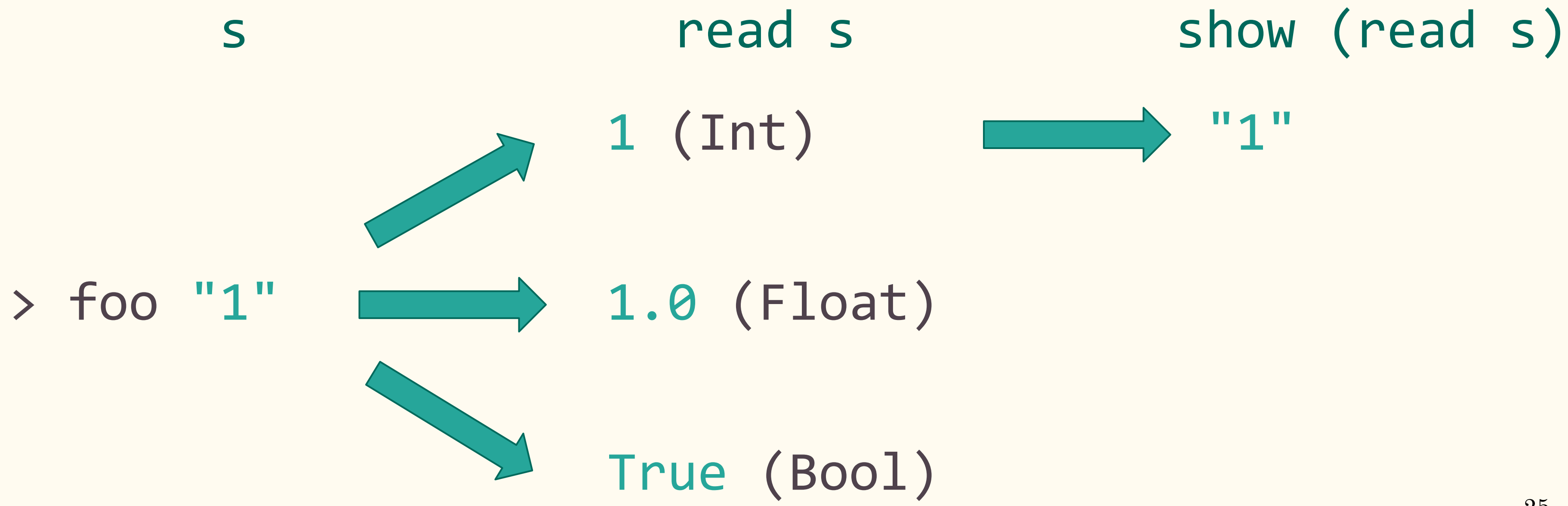
1 (Int)

1.0 (Float)

True (Bool)

a = ?

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



a = ?

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

s

read s

show (read s)

1 (Int)

"1"

> foo "1"

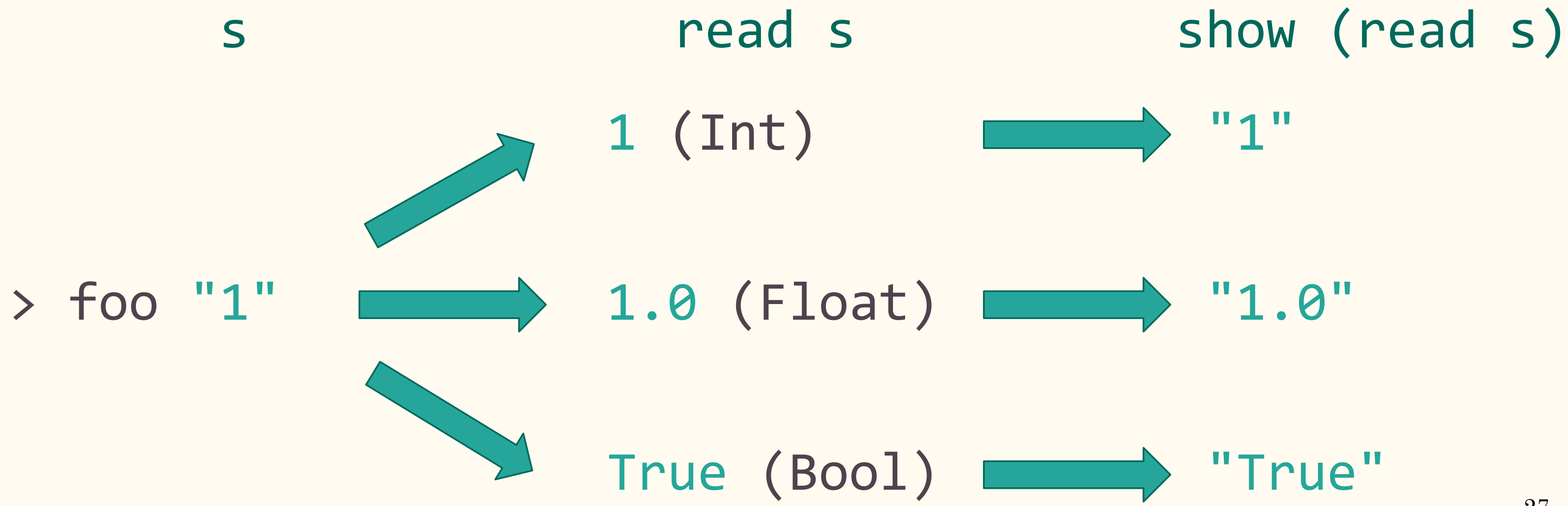
1.0 (Float)

"1.0"

True (Bool)

a = ?

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



Ambiguity!


PICKERING 8
WHITBY 29½
LEEDS 41
DOVER 297
NORWICH 189
LIVERPOOL 112
FIELD 78
HULL 38
NEWCASTLE 98
FILE
23
TADCASTER 29
ALFRETON 88
SCARBOROUGH 23
HARROGAT
YORK 17
223
ACCLESFIELD 100½
ONCASTER
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.

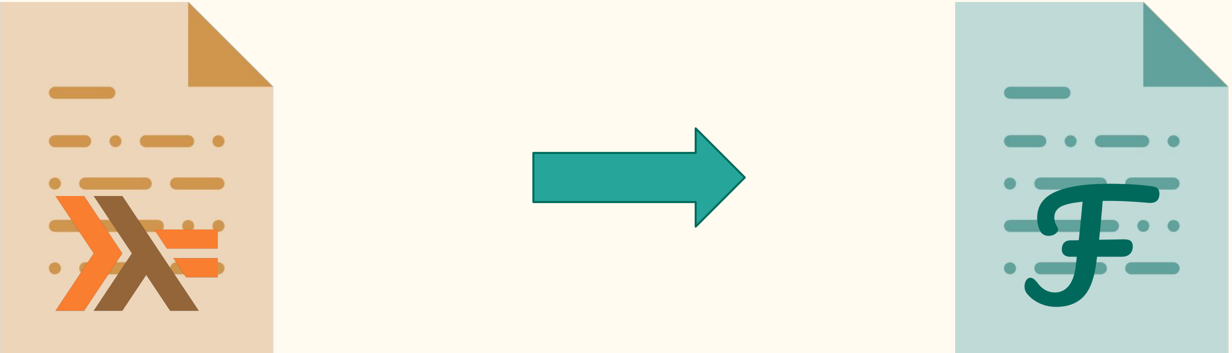
```haskell
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```

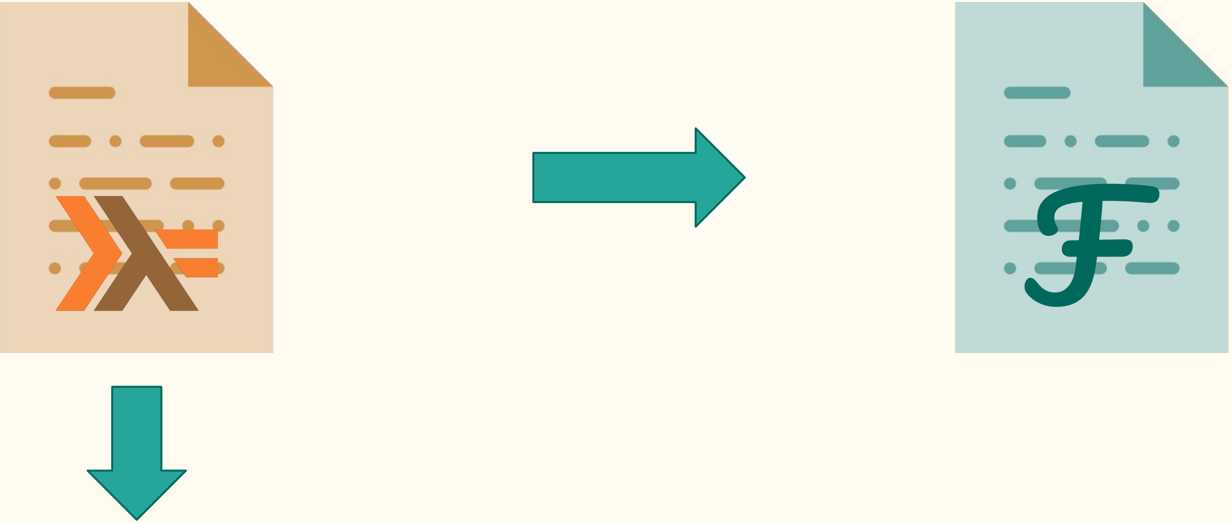
a = ?

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

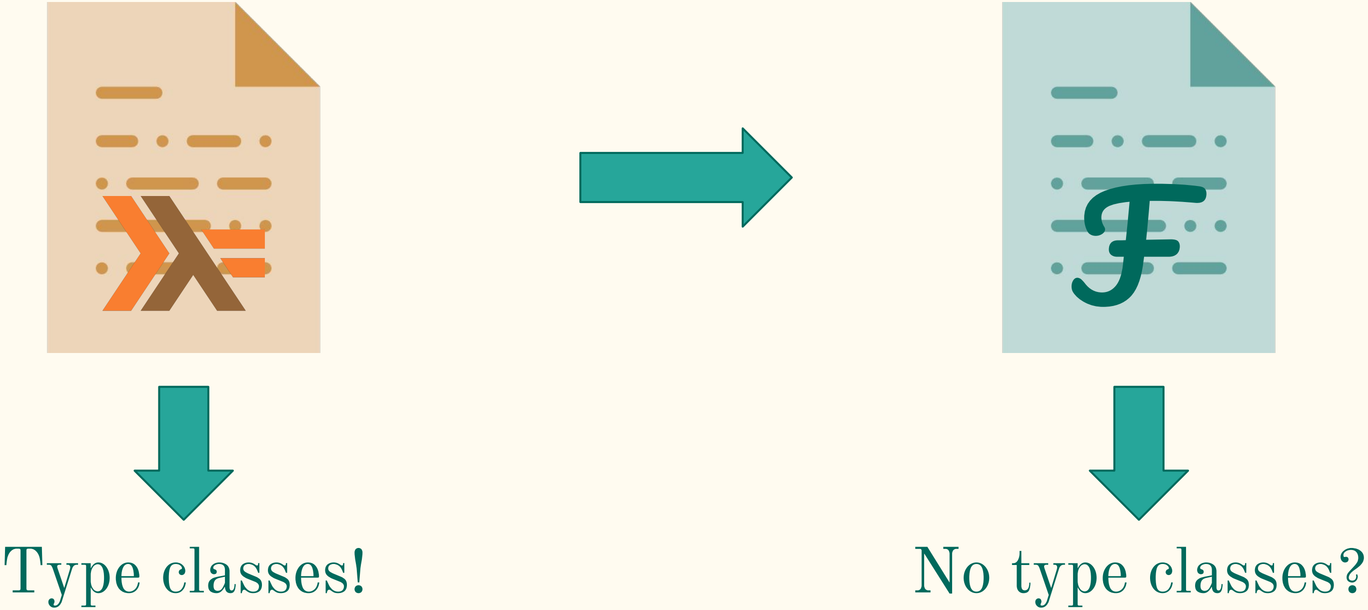
a = ?

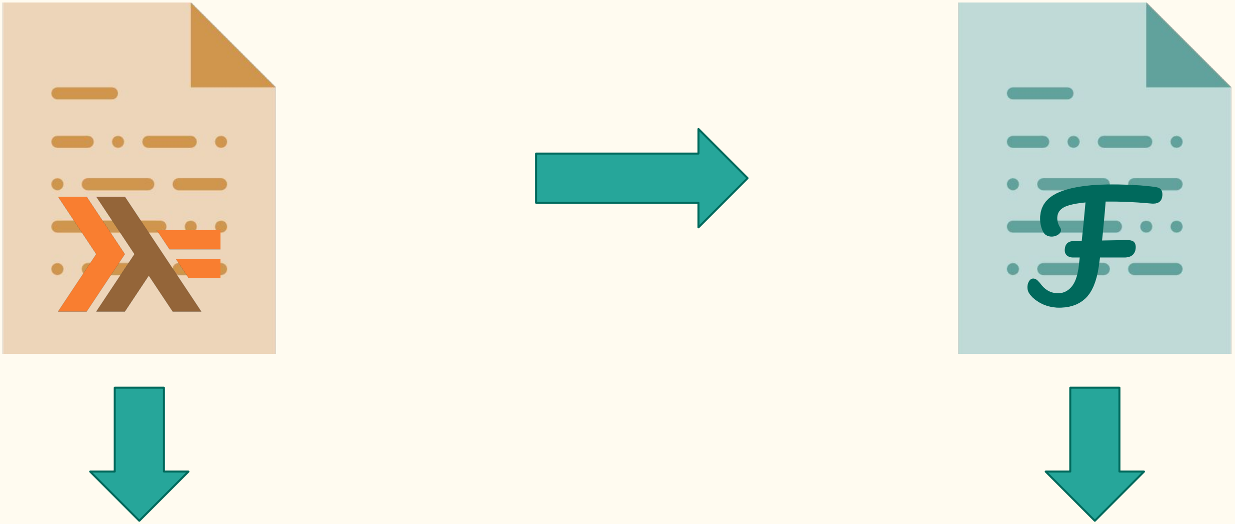
```
foo :: (Show a , Read a) => String -> String
foo s = show (read s)
```





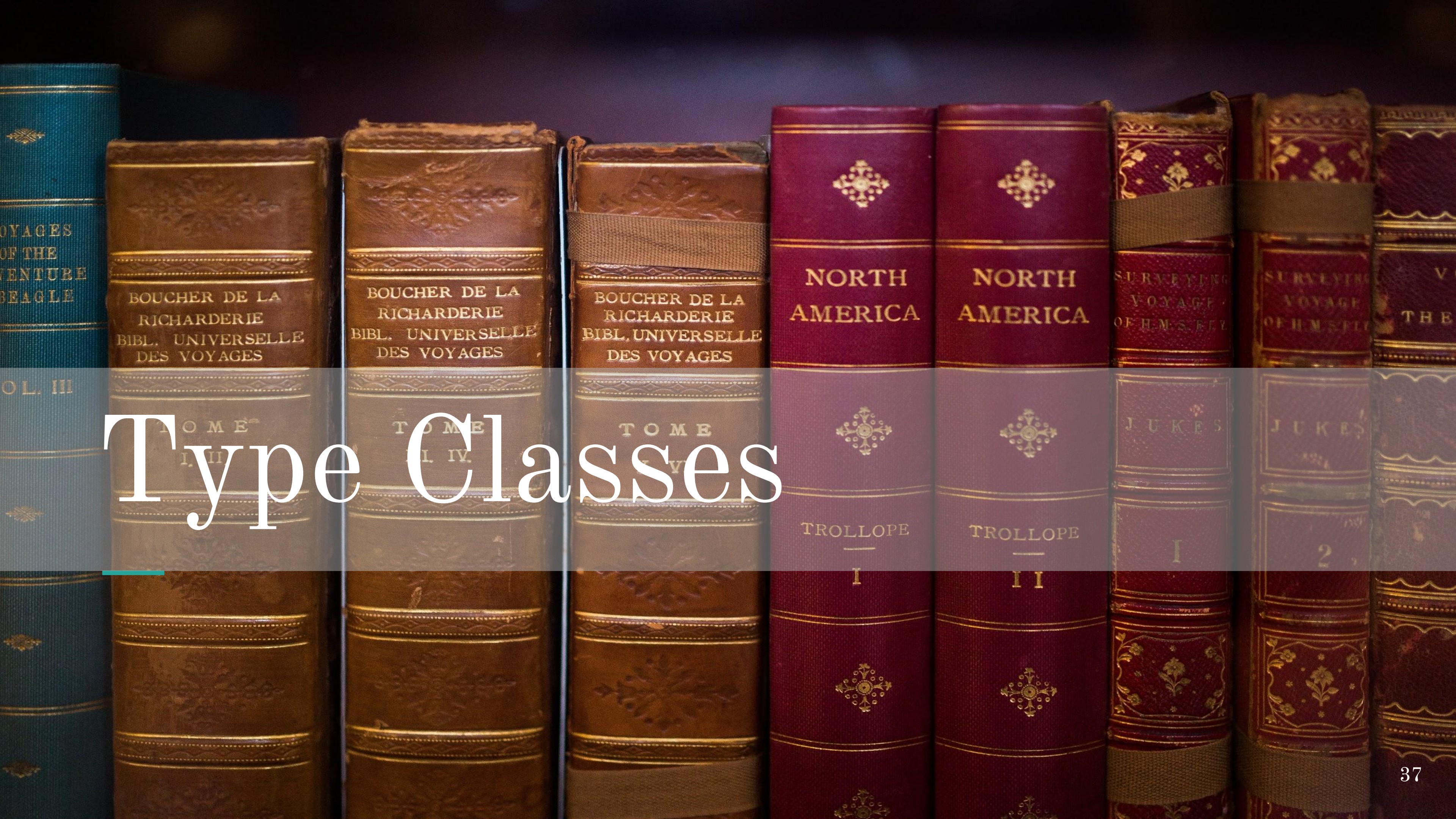
Type classes!


Type classes!
No type classes?



Type classes!

No type classes?
Dictionaries!



Type Classes

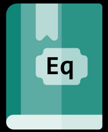
```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

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

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

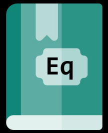
```
class Eq a where
  (==) :: a -> a -> Bool

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

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

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

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

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

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

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

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

data  =

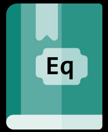
```
class Eq a where
  (==) :: a -> a -> Bool

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

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

data 

=

```
  EqDict
```

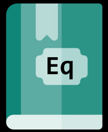
```
class Eq a where
  (==) :: a -> a -> Bool

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

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

data 

=

```
  EqDict (a -> a -> Bool)
```

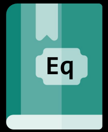
```
class Eq a where
  (==) :: a -> a -> Bool

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

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

data  =


```
  EqDict (a -> a -> Bool)

(==) ::
```

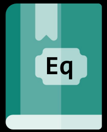
```
class Eq a where
  (==) :: a -> a -> Bool

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

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

```
data
```



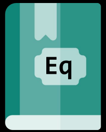

```
=
  EqDict (a -> a -> Bool)

(==) :: a -> a -> Bool
```

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

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

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

data 

=

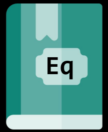
```
  EqDict (a -> a -> Bool)
```

(==) :: 

->

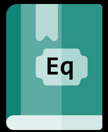
```
        a -> a -> Bool
```

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

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

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

data 

=

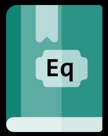
```
  EqDict (a -> a -> Bool)
```

(==) :: 

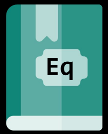
->

```
          a -> a -> Bool

(==) d =
```

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

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

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

data  =

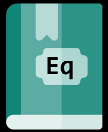
  EqDict (a -> a -> Bool)

(==) ::  ->

  a -> a -> Bool

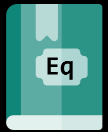
(==) (EqDict e) =

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

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

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

data 

=

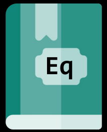
```haskell
  EqDict (a -> a -> Bool)
```

(==) :: 

->

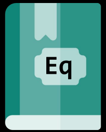
```haskell
          a -> a -> Bool

(==) (EqDict e) = e
```

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

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

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

data 

=

```haskell
  EqDict (a -> a -> Bool)
```

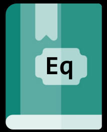
(==) :: 

->

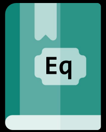
```haskell
        a -> a -> Bool

(==) (EqDict e) = e
```

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

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

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

```
data
```

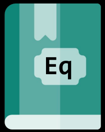



```
=
  EqDict (a -> a -> Bool)

(==) ::
```



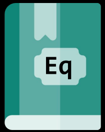

```
->
        a -> a -> Bool

(==) (EqDict e) = e

eqDBool ::
```

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

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

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

data 

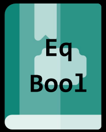
=

```
  EqDict (a -> a -> Bool)
```

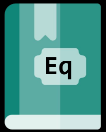
(==) :: 

->

```
          a -> a -> Bool

(==) (EqDict e) = e
```

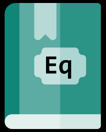
eqDBool :: 

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

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

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

`data` 

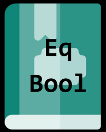
`=`

```haskell
  EqDict (a -> a -> Bool)
```

`(==) ::` 

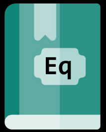
`->`

```haskell
          a -> a -> Bool
(==) (EqDict e) = e
```

`eqDBool ::` 

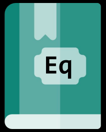

```haskell
eqDBool =
```

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

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

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

data 

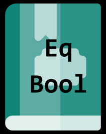
=

```
  EqDict (a -> a -> Bool)
```

(==) :: 

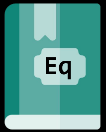
->

```
        a -> a -> Bool
(==) (EqDict e) = e
```

eqDBool :: 

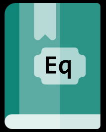

```
eqDBool = EqDict
```

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

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

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

data 

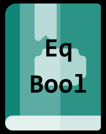
=

```
  EqDict (a -> a -> Bool)
```

(==) :: 

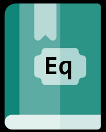
->

```
          a -> a -> Bool
(==) (EqDict e) = e
```

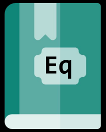
eqDBool :: 


```
eqDBool = EqDict (...)
```

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

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

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

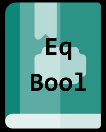
data 

=
  EqDict (a -> a -> Bool)

(==) :: 

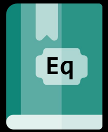
->
  a -> a -> Bool

(==) (EqDict e) = e

eqDBool :: 

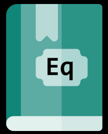
eqDBool = EqDict (...)

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

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

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

data 

=

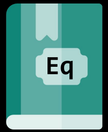
```
  EqDict (a -> a -> Bool)
```

(==) :: 

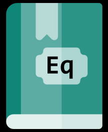
->

```
          a -> a -> Bool

(==) (EqDict e) = e
```

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

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

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

data 

=
  EqDict (a -> a -> Bool)

(==) :: 

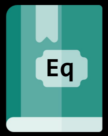
->
        a -> a -> Bool

(==) (EqDict e) = e

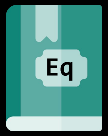
foo =

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

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

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

```
data
```

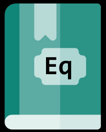



```
=
  EqDict (a -> a -> Bool)

(==) ::
```



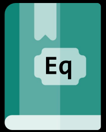

```
->
        a -> a -> Bool

(==) (EqDict e) = e

foo = \d :     .
```

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

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

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

`data`


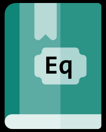
`=`

```
  EqDict (a -> a -> Bool)
```

`(==) ::`


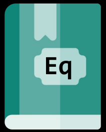
`->`

```
         a -> a -> Bool

(==) (EqDict e) = e
```

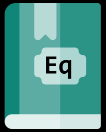
`foo = \d :`


`.`

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

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

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

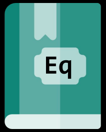
data 

=
  EqDict (a -> a -> Bool)

(==) :: 

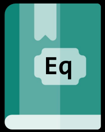
->
  a -> a -> Bool

(==) (EqDict e) = e

foo = \d : 

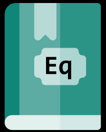
.
  \x : a.

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

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

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

data 

=

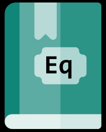
  EqDict (a -> a -> Bool)

(==) :: 

->

  a -> a -> Bool

(==) (EqDict e) = e

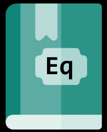
foo = \d : 

.

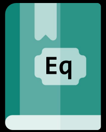
  \x : a.

  (==)

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

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

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

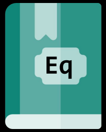
data 

=
  EqDict (a -> a -> Bool)

(==) :: 

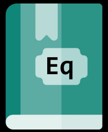
->
  a -> a -> Bool

(==) (EqDict e) = e

foo = \d : 

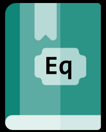
.
  \x : a.
  (==) d

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

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

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

data 

=

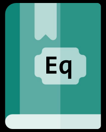
    EqDict (a -> a -> Bool)

(==) :: 

->

        a -> a -> Bool

(==) (EqDict e) = e

foo = \d : 

.

      \x : a.

      (==) d x x

Superclasses

```haskell
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
```

data 

=

```
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
```

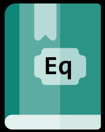
```
data
```



```
= OrdDict
```

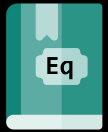
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

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

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

data 

= OrdDict 

```
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
```

data 

= OrdDict 

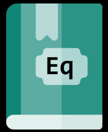

```
  (a -> a -> Bool)
```

```
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
```

```
data Ord = OrdDict Eq
  (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
```

data 

= OrdDict 

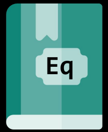

```
  (a -> a -> Bool)

(>) :: 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
```

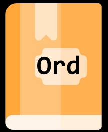
```
data [Ord] = OrdDict [Eq]
  (a -> a -> Bool)

(>) :: [Ord] ->
          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
```

data 

= OrdDict 

(a -> a -> Bool)

(>) :: 

->

a -> a -> Bool

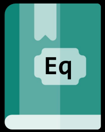
(>) d =

```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
```

```haskell
data Ord = OrdDict Eq
  (a -> a -> Bool)

(>) :: Ord ->
         a -> a -> Bool
(>) (OrdDict d g) =
```

```
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
```

data 

= OrdDict 

(a -> a -> Bool)

(>) :: 

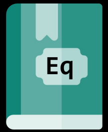
->

a -> a -> Bool

(>) (OrdDict d g) = g

```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
```

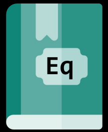
data 

= OrdDict 

(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
```

data 

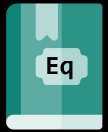
= OrdDict 

(a -> a -> Bool)

ordDBool :: 

```
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
```

data 

= OrdDict 

(a -> a -> Bool)

ordDBool :: 

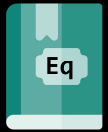
ordDBool =

```
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
```

```
data Ord = OrdDict Eq
  (a -> a -> Bool)

ordDBool :: Ord Bool
ordDBool =
  OrdDict
```

```
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
```

data 

= OrdDict 

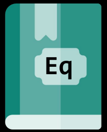
  (a -> a -> Bool)

ordDBool :: 

ordDBool =
  OrdDict eqDBool

```
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
```

data 

= OrdDict 


```
  (a -> a -> Bool)
```

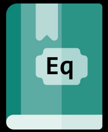
ordDBool :: 


```
ordDBool =
  OrdDict eqDBool (...)
```

```
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
```

```
data
```


```
= OrdDict
```




```
  (a -> a -> Bool)

ordDBool ::
```

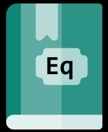

```
ordDBool =
  OrdDict eqDBool (...)
```

```
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
```

```
data Ord = OrdDict Eq
  (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
```

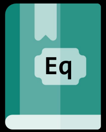
data 

= OrdDict 

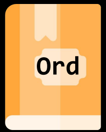

```
  (a -> a -> Bool)

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
```

data 

= OrdDict 

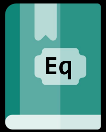
(a -> a -> Bool)

foo = \do : 

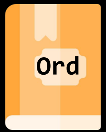
.

```
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
```

data 

= OrdDict 

(a -> a -> Bool)

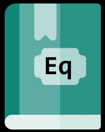
foo = \do : 

.

\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
```

data 

= OrdDict 

(a -> a -> Bool)

foo = \do : 

.
\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
```

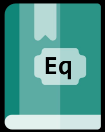
```
data Ord = OrdDict Eq
  (a -> a -> Bool)

(==) :: Eq ->
          a -> a -> Bool
(==) (EqDict e) = e

foo = \do : Ord .
        \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
```

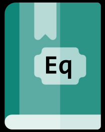
```
data Ord = OrdDict Eq
  (a -> a -> Bool)

(==) :: Eq ->
          a -> a -> Bool
(==) (EqDict e) = e

foo = \do : Ord .
      \x : a .
      (==) de
```

```
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
```

data 

= OrdDict 

  (a -> a -> Bool)

(==) :: 

->

  a -> a -> Bool

(==) (EqDict e) = e

foo = \do : 

.

  \x : a .

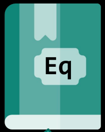
  (==) de x x

```
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
```

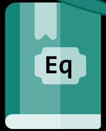
```
data Ord = OrdDict Eq
  (a -> a -> Bool)

(==) :: Eq ->
          a -> a -> Bool
(==) (EqDict e) = e

foo = \do : Ord .
      \x : a .
      (==) de 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
```

data 

= OrdDict 

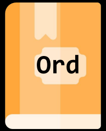
(a -> a -> Bool)

(==) :: 

->

a -> a -> Bool

(==) (EqDict e) = e

foo = \do : 

.

\x : a .

(==) de 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
```

```
data Ord = OrdDict Eq
   (a -> a -> Bool)

(==) :: Eq ->
          a -> a -> Bool

(==) (EqDict e) = e

foo = \do : Ord .
        \x : a .
          (==) de x x
```

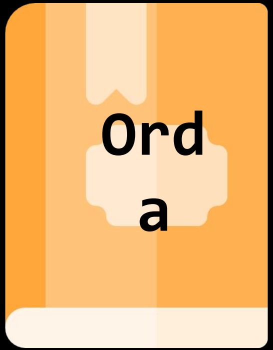
```
class Eq a => Ord a where
  (>) :: a -> a -> Bool

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

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

```
data Ord = OrdDict Eq
  (a -> a -> Bool)

(==) :: Eq ->
          a -> a -> Bool
(==) (EqDict e) = e

foo = \OrdDict de o : Ord .
      \x : a .
      (==) de x x
```

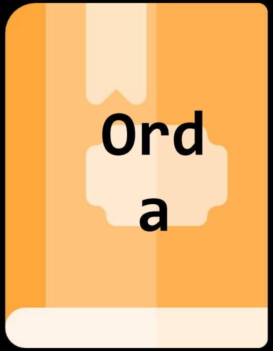
Just One More!

```
instance Ord a where
  _    > _     = False
```

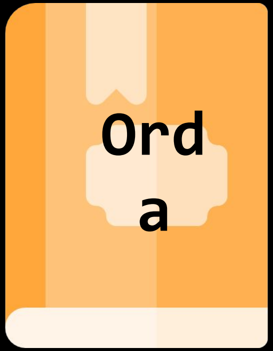
```
instance Ord a where
  _     > _      = False
```

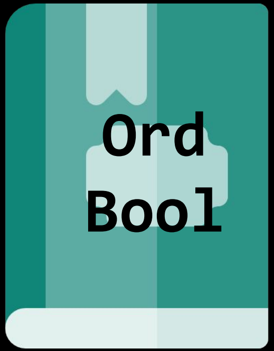


```
instance Ord a where
  _    > _     = False
instance Ord Bool where
  True > False = True
  _    > _     = False
```

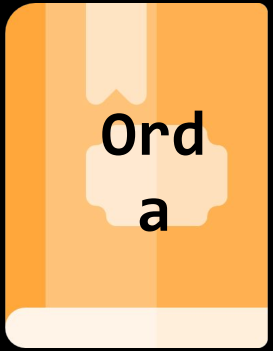


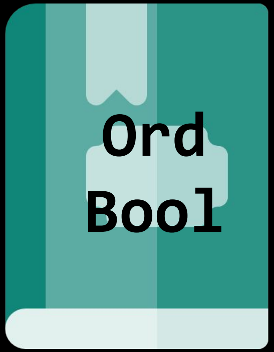
```haskell
instance Ord a where
  _    > _     = False
instance Ord Bool where
  True > False = True
  _    > _     = False
```



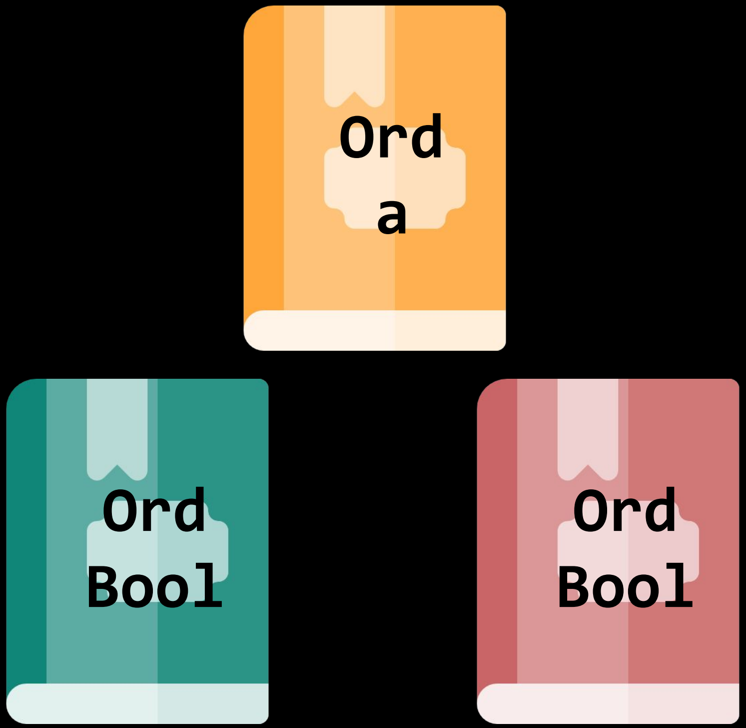


```
instance Ord a where
  _    > _     = False
instance Ord Bool where
  True > False = True
  _    > _     = False
instance Ord Bool where
  True > True  = True
  True > False = True
  _    > _     = False
```

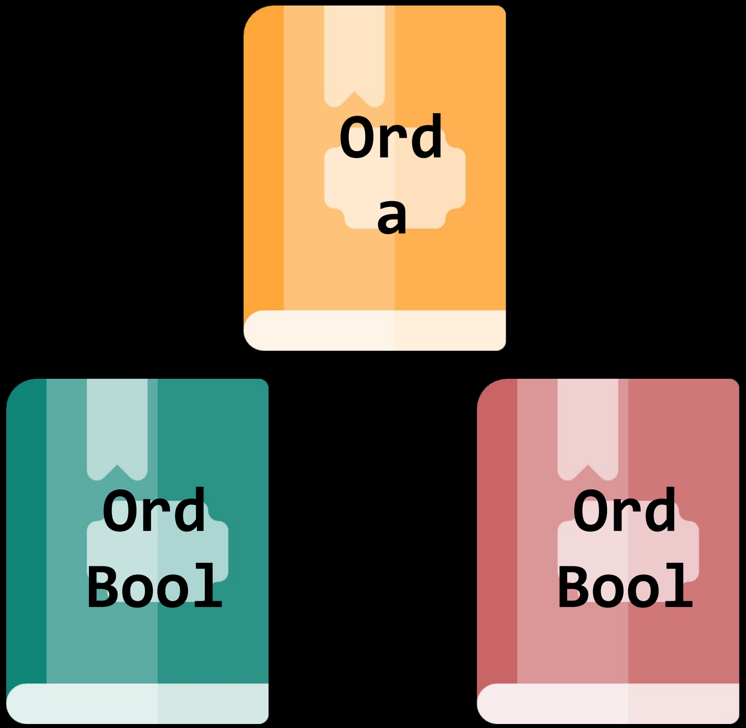




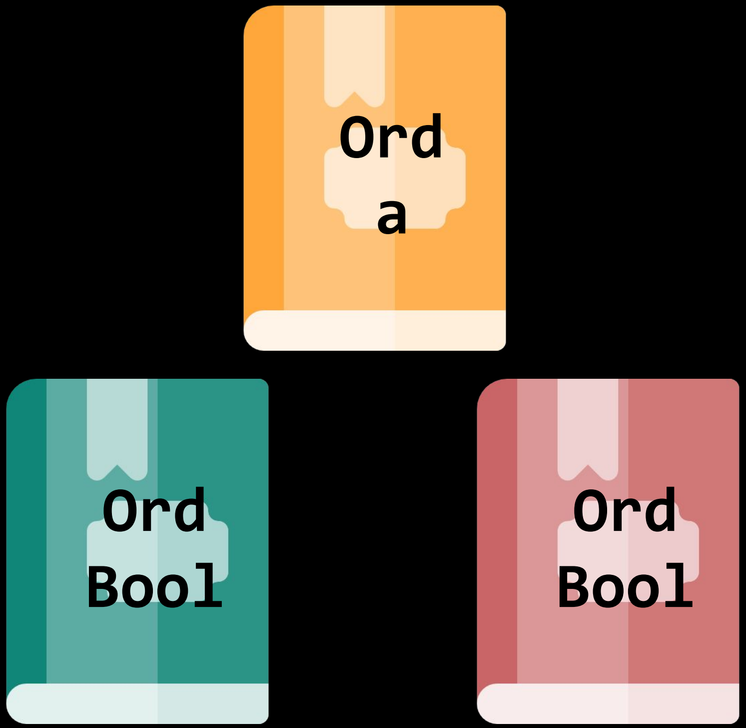
```
instance Ord a where
  _     > _     = False
instance Ord Bool where
  True > False = True
  _     > _     = False
instance Ord Bool where
  True > True  = True
  True > False = True
  _     > _     = False
```



```haskell
instance Ord a where
  _    > _     = False
instance Ord Bool where
  True > False = True
  _    > _     = False
instance Ord Bool where
  True > True  = True
  True > False = True
  _    > _     = False
foo :: Bool
foo = True > True
```

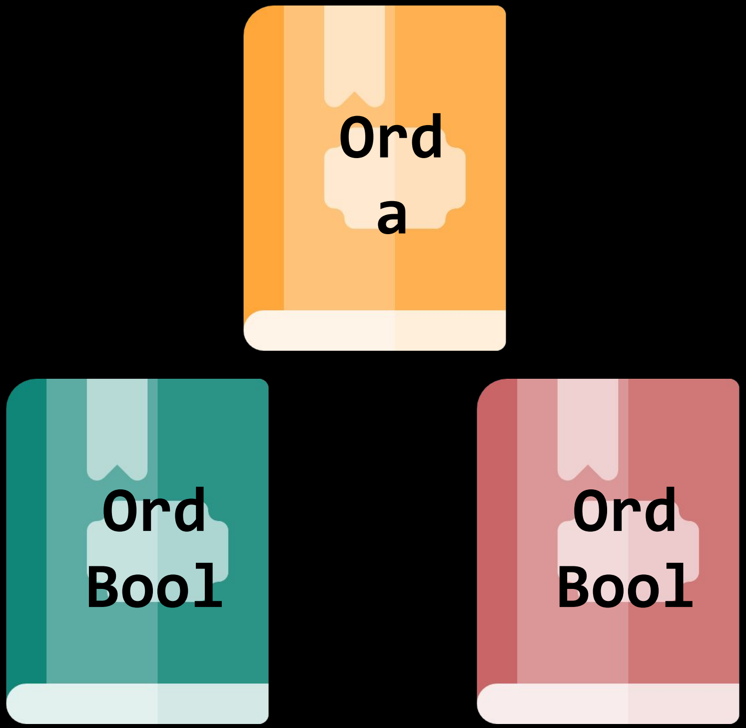


```haskell
instance Ord a where
  _     > _      = False
instance Ord Bool where
  True > False = True
  _     > _      = False
instance Ord Bool where
  True > True   = True
  True > False = True
  _     > _      = False
foo :: Bool
foo = True > True
```



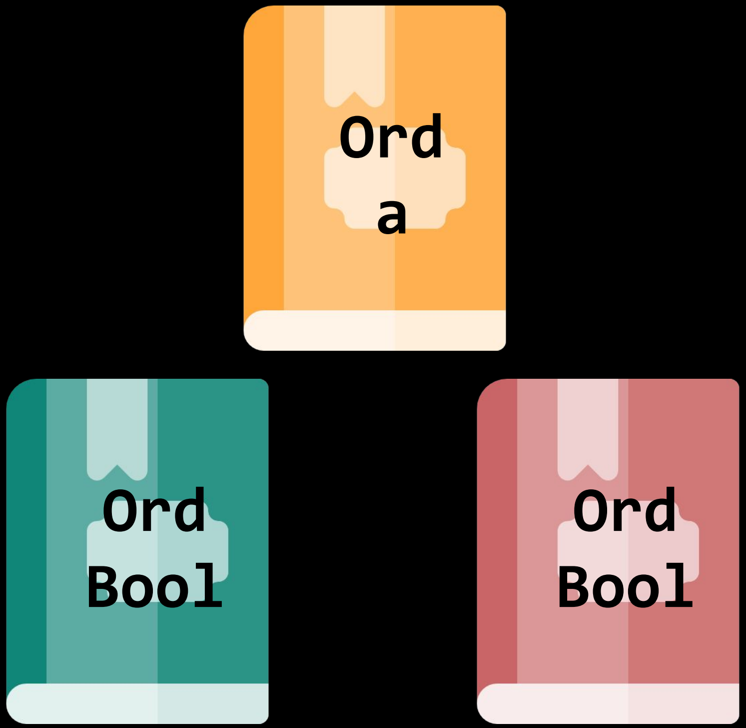
foo =

```haskell
instance Ord a where
  _    > _      = False
instance Ord Bool where
  True > False = True
  _    > _      = False
instance Ord Bool where
  True > True   = True
  True > False = True
  _    > _      = False
foo :: Bool
foo = True > True
```



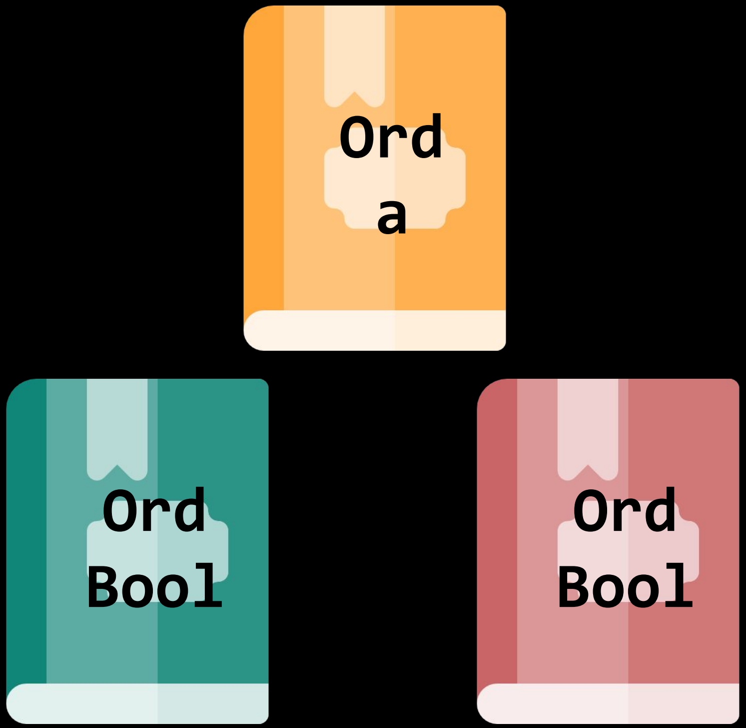

```haskell
foo = (>)
```

```haskell
instance Ord a where
  _     > _     = False
instance Ord Bool where
  True  > False = True
  _     > _     = False
instance Ord Bool where
  True  > True  = True
  True  > False = True
  _     > _     = False
foo :: Bool
foo = True > True
```



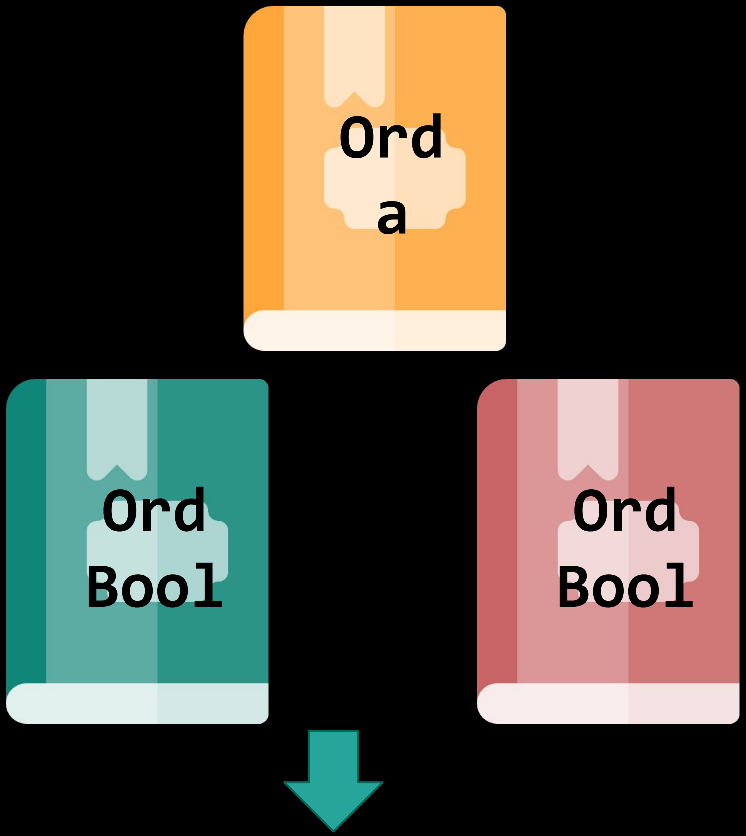

```haskell
foo = (>) d
```

```
instance Ord a where
  _     > _     = False
instance Ord Bool where
  True > False = True
  _     > _     = False
instance Ord Bool where
  True > True  = True
  True > False = True
  _     > _     = False
foo :: Bool
foo = True > True
```



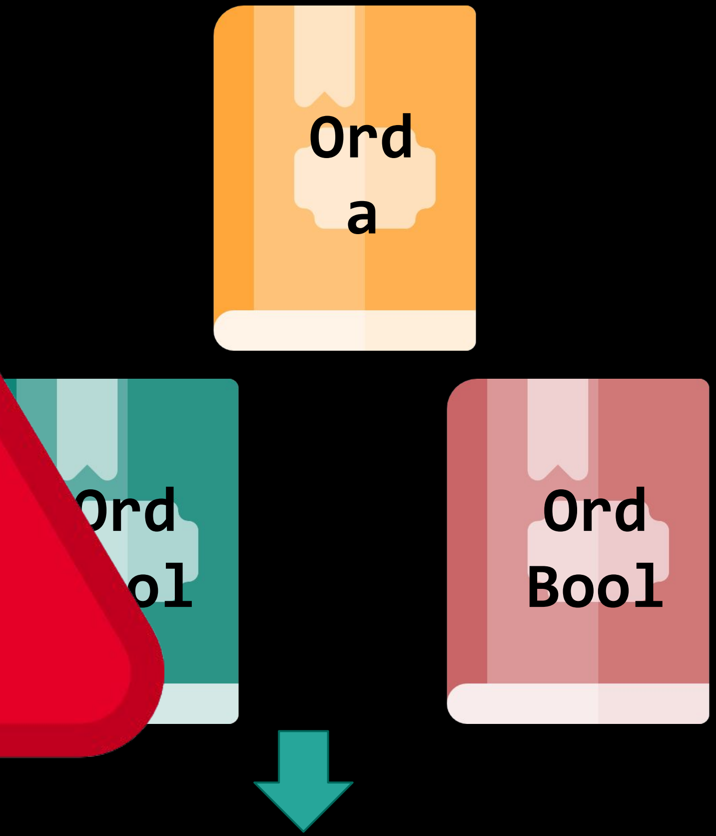

```
foo = (>) d True True
```

```haskell
instance Ord a where
  _     > _      = False
instance Ord Bool where
  True > False = True
  _     > _      = False
instance Ord Bool where
  True > True   = True
  True > False = True
  _     > _      = False
foo :: Bool
foo = True > True
```



foo = (>) d True True

```
instance Ord a where
  _     > _     = False
instance Ord Bool where
  True > False = True
  _     > _     = False
instance Ord Bool wh
  True > True  = Tr
  True > False = T
  _     > _     = False
foo :: Bool
foo = True > True
```




```
foo = (>) d True True
```

No Overlapping Instances

Let’s Dig Deeper

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

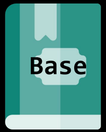
```haskell
class Base a where
  base :: a -> Bool

class Base a => Sub1 a
```

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

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

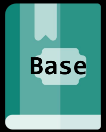
```haskell
class Base a where
  base :: a -> Bool

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
```

data 

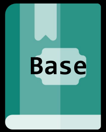
=

```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
```

data  =

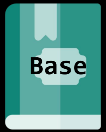

```haskell
  BaseDict
```

```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
```

data  =


```haskell
  BaseDict (a -> Bool)
```

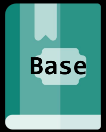
```
class Base a where
  base :: a -> Bool

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

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

data 

=

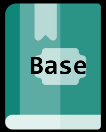
```
  BaseDict (a -> Bool)

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
```

data 

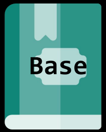
=

BaseDict (a -> Bool)

base :: 

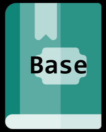
-> a -> Bool

```
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
```

data 

=

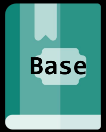
BaseDict (a -> Bool)

base :: 

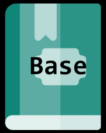
-> a -> Bool

base d =

```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
```

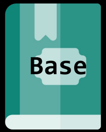
data  =

```haskell
  BaseDict (a -> Bool)
```

base ::  -> a -> Bool

```haskell
base (BaseDict b) =
```

```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
```

data 

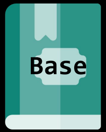
=
  BaseDict (a -> Bool)

base :: 

-> a -> Bool

base (BaseDict b) = b

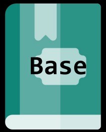
```haskell
class Base a where
  base :: a -> Bool

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

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

data 

=

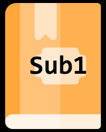
```haskell
  BaseDict (a -> Bool)
```

```
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
```

data 

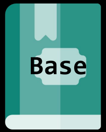
=

```
  BaseDict (a -> Bool)
```

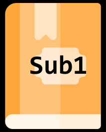
data 

=

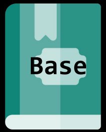
```haskell
class Base a where
  base :: a -> Bool

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

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

data  =


```haskell
  BaseDict (a -> Bool)
```

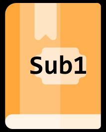
data  = Sub1Dict

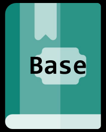
```haskell
class Base a where
  base :: a -> Bool

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

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

data 

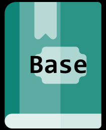
=

```haskell
  BaseDict (a -> Bool)
```

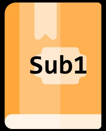
data 

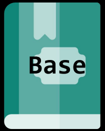
= Sub1Dict 

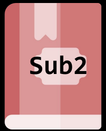
```haskell
class Base a where
  base :: a -> Bool

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

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

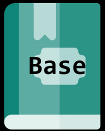
data 

=
  BaseDict (a -> Bool)

data 

= Sub1Dict 

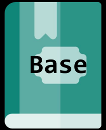
data 

= Sub2Dict 

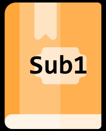
```haskell
class Base a where
  base :: a -> Bool

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

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

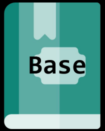
```haskell
data Base =
  BaseDict (a -> Bool)

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base
```

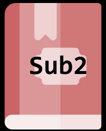
```haskell
class Base a where
  base :: a -> Bool

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

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

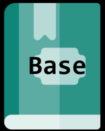
data 

=
  BaseDict (a -> Bool)

data 

= Sub1Dict 

data 

= Sub2Dict 

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
```

```haskell
data Base =
  BaseDict (a -> Bool)

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base

foo = \d1 : Sub1 .
```

```
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
```

```
data Base =
  BaseDict (a -> Bool)

data Sub1 = Sub1Dict Base

data Sub2 = Sub2Dict Base

foo = \d1 : Sub1 .
      \d2 : Sub2 .
```

```
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
```

```
data Base =
  BaseDict (a -> Bool)

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base

foo = \d1 : Sub1 .
      \d2 : Sub2 .
      \x : 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
```

```
data Base =
  BaseDict (a -> Bool)

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base
foo = \d1 : Sub1 .
      \d2 : Sub2 .
      \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 :: Base -> a -> Bool

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base
foo = \d1 : Sub1 .
      \d2 : Sub2 .
      \x : a.
      base
```

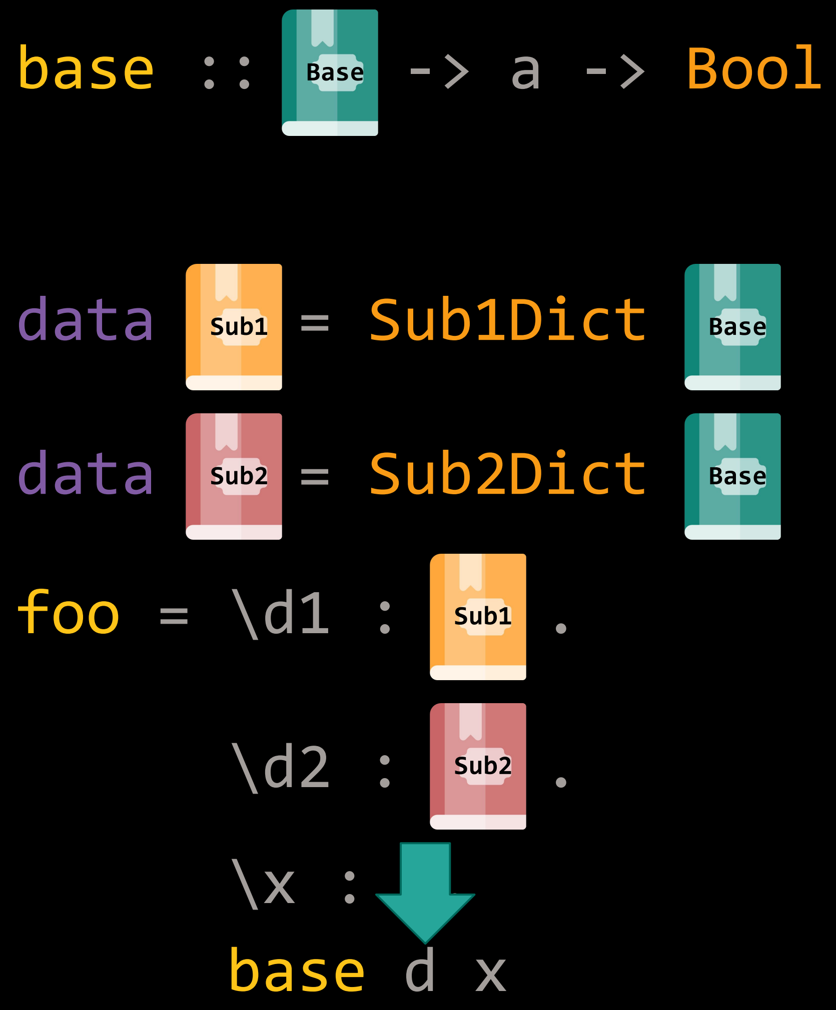
```
class Base a where
  base :: a -> Bool

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

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

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

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base
foo = \d1 : Sub1 .
      \d2 : Sub2 .
      \x : a.
      base d
```

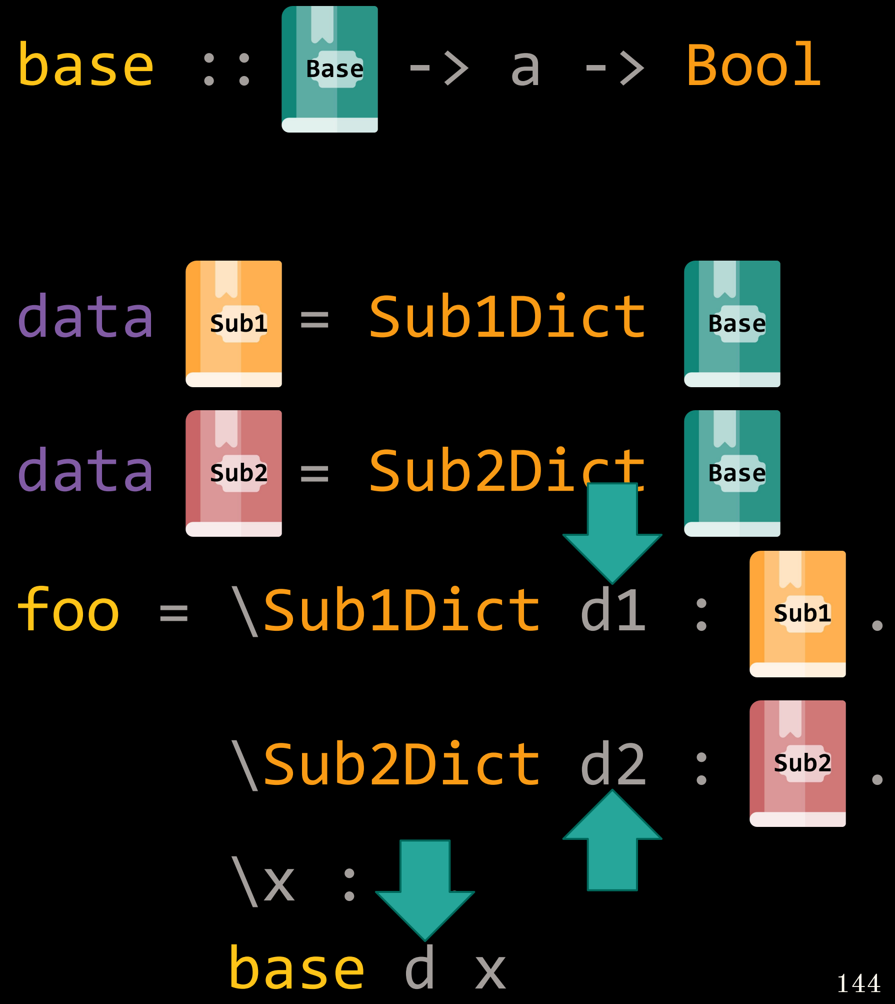
```
class Base a where
  base :: a -> Bool

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

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

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

data Sub1 = Sub1Dict Base
data Sub2 = Sub2Dict Base
foo = \d1 : Sub1 .
      \d2 : Sub2 .
      \x : a.
      base d x
```

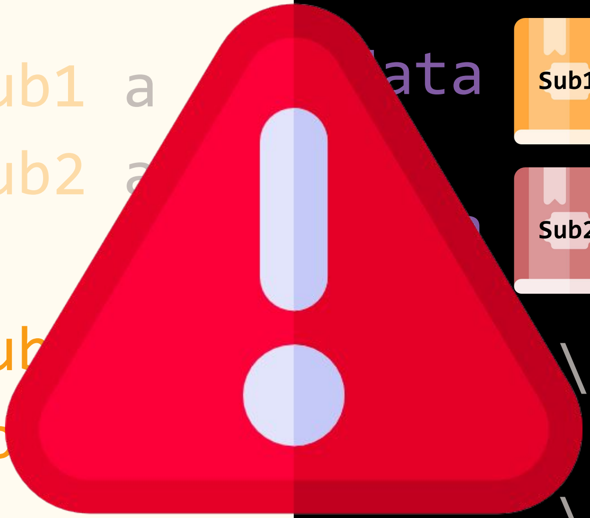
```haskell
class Base a where
  base :: a -> Bool

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

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



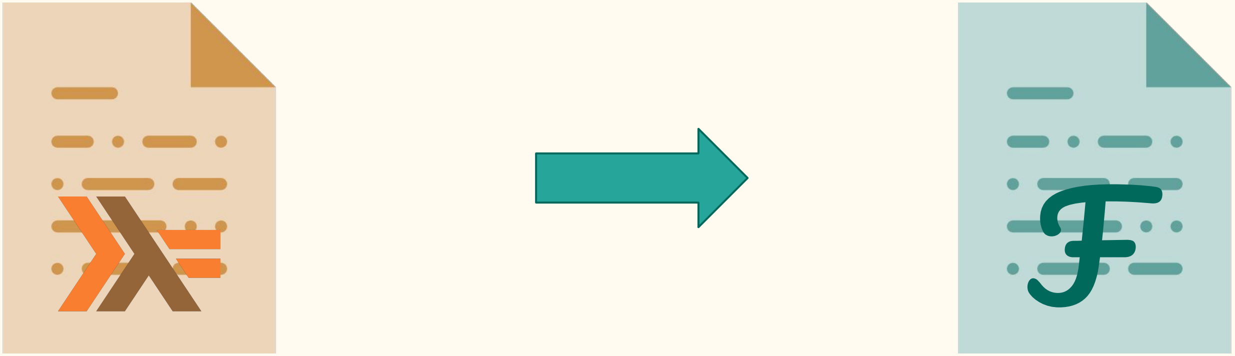
```haskell
class Base a where
  base :: a -> Bool

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

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



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

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

foo :: (Sub1 a, Sub
        => a -> Boo
foo x = base x
```

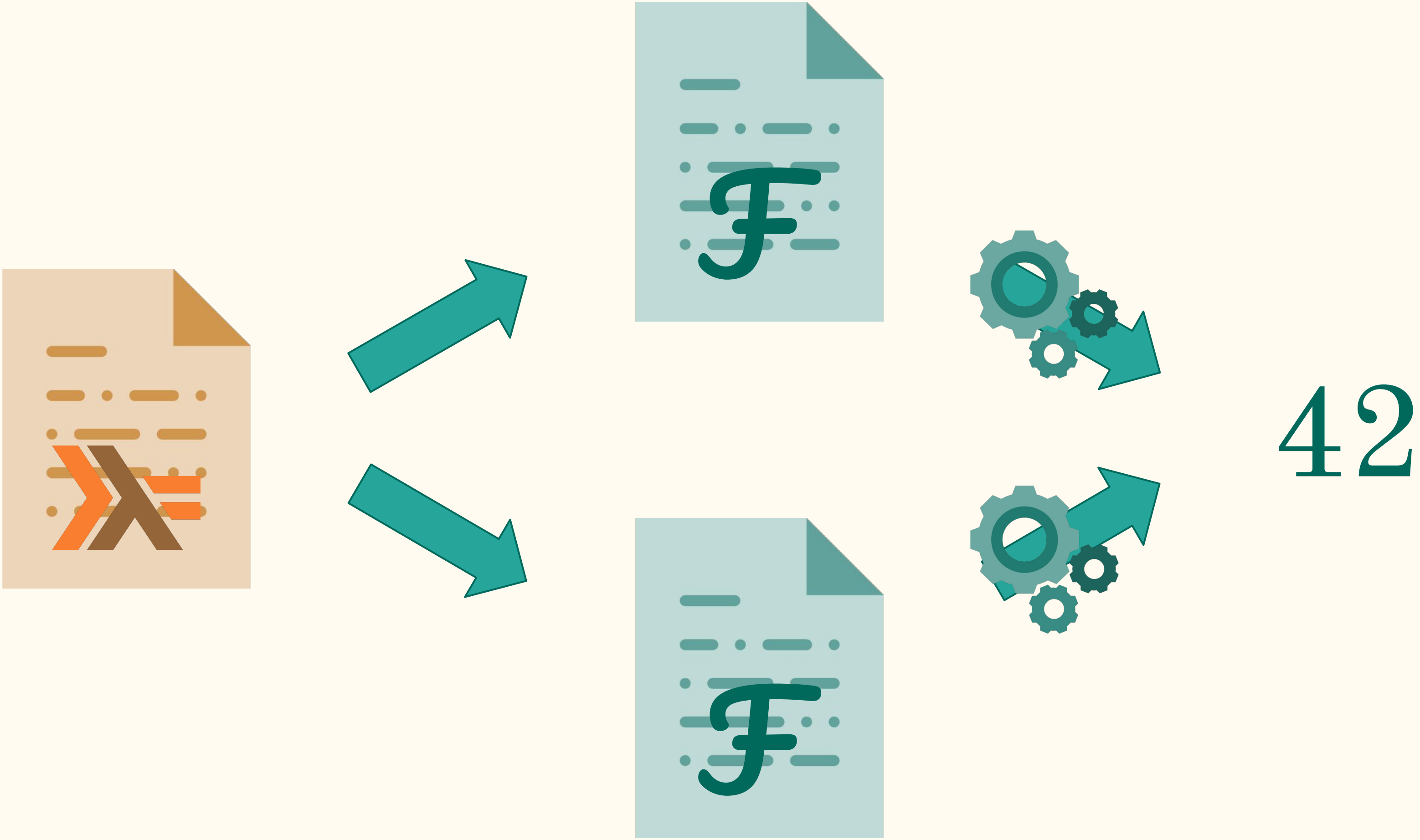


```
base :: Base -> a -> Bool

ata Sub1 = Sub1Dict Base
    Sub2 = Sub2Dict Base

\Sub1Dict d1 : Sub1 .
\Sub2Dict d2 : Sub2 .
\x :
base d x
```

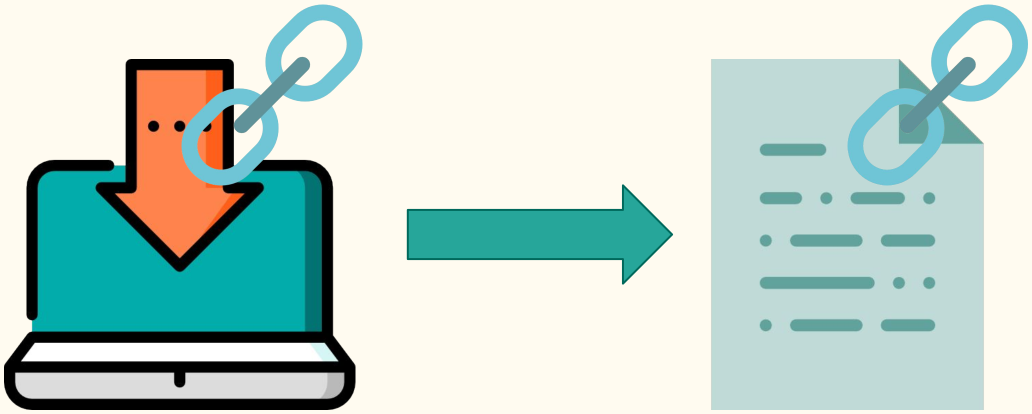



CHAOS MONKEY!

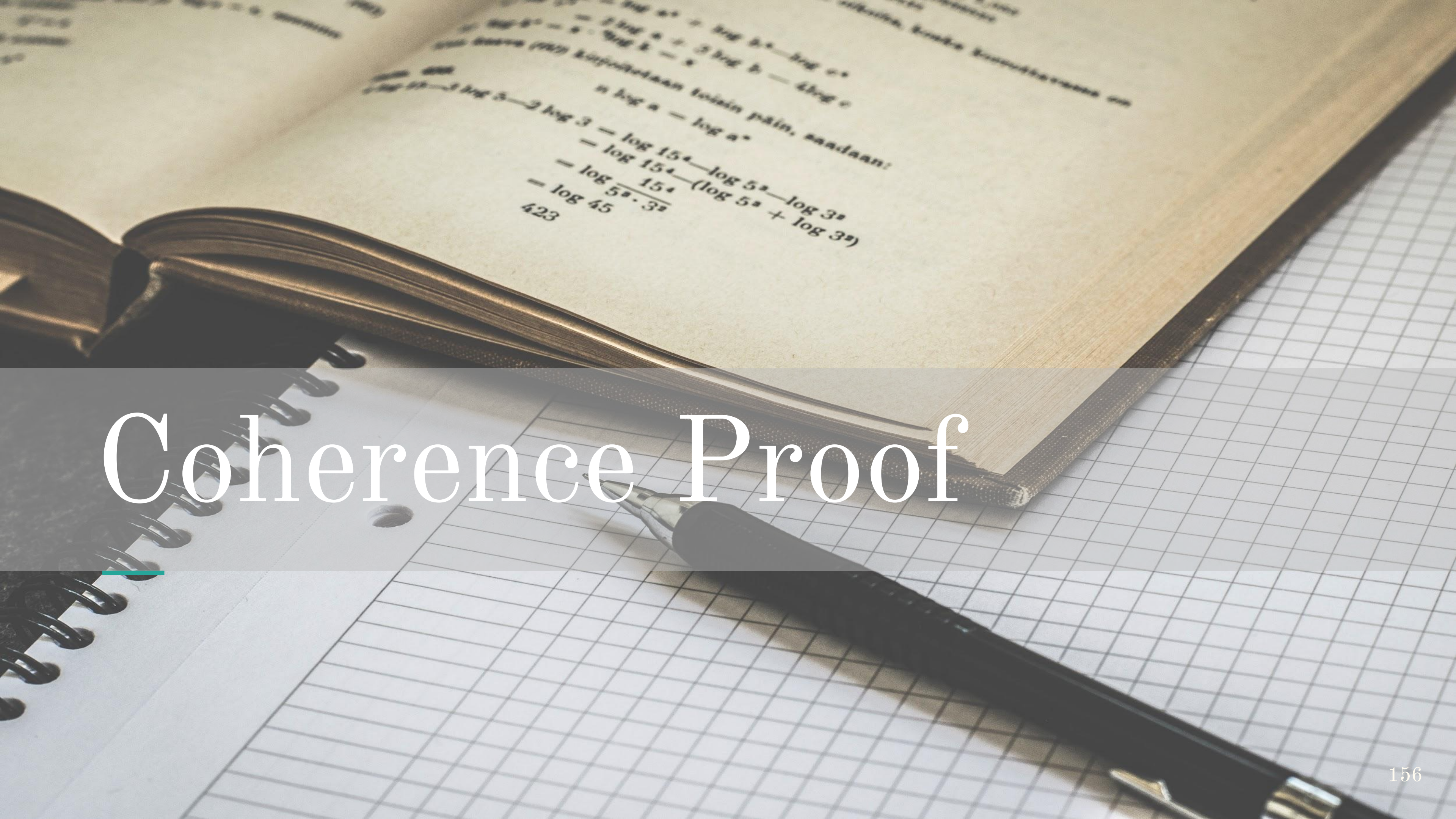
Coherence


42

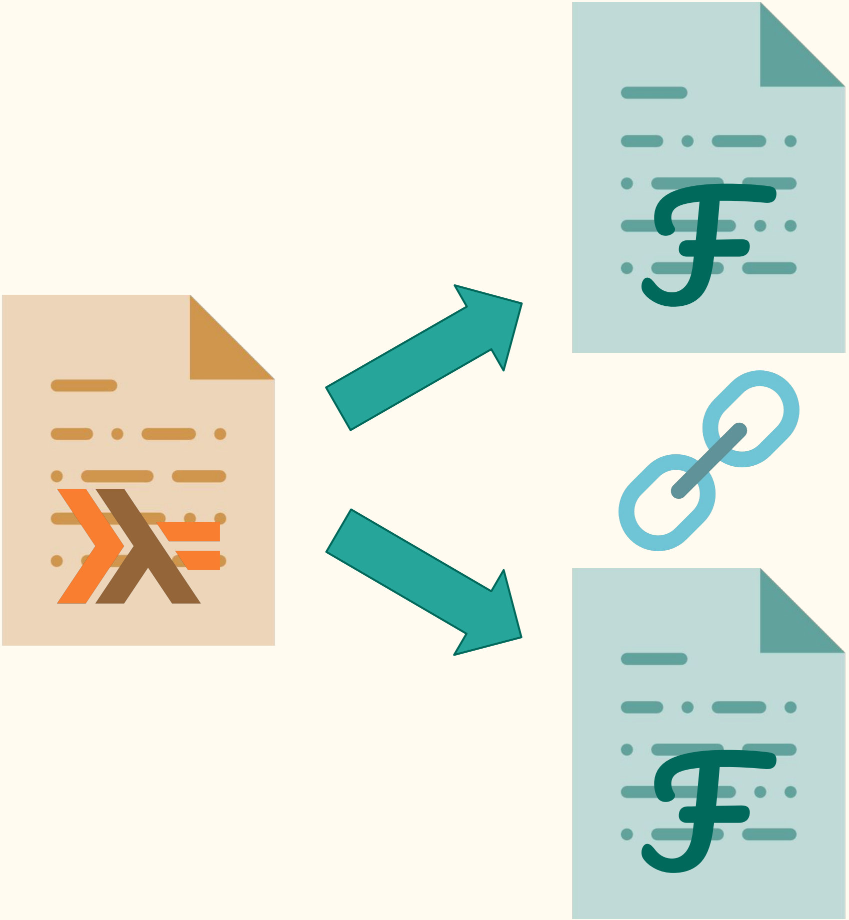
Proof through Logical Relations

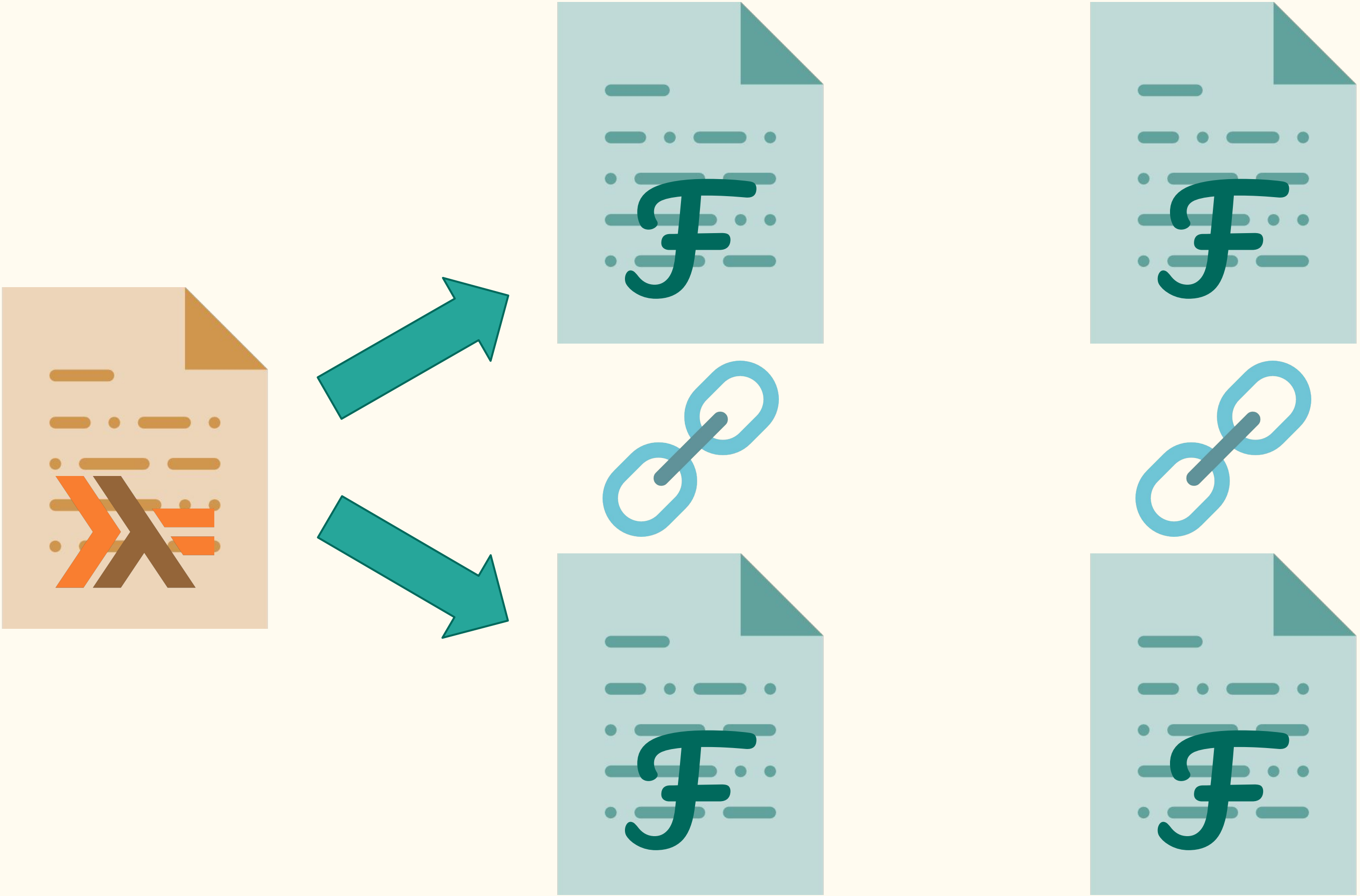


Coherence Proof

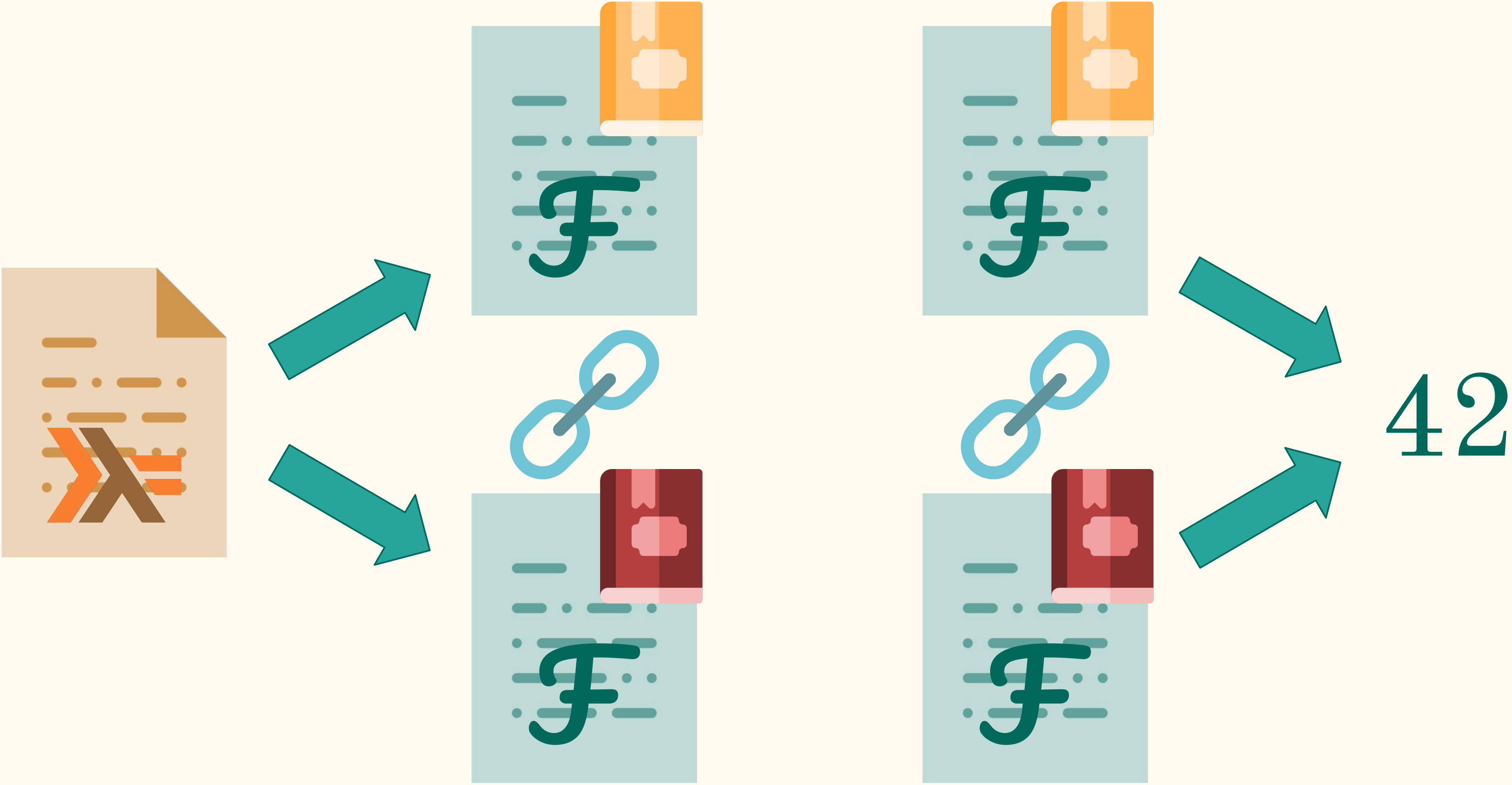


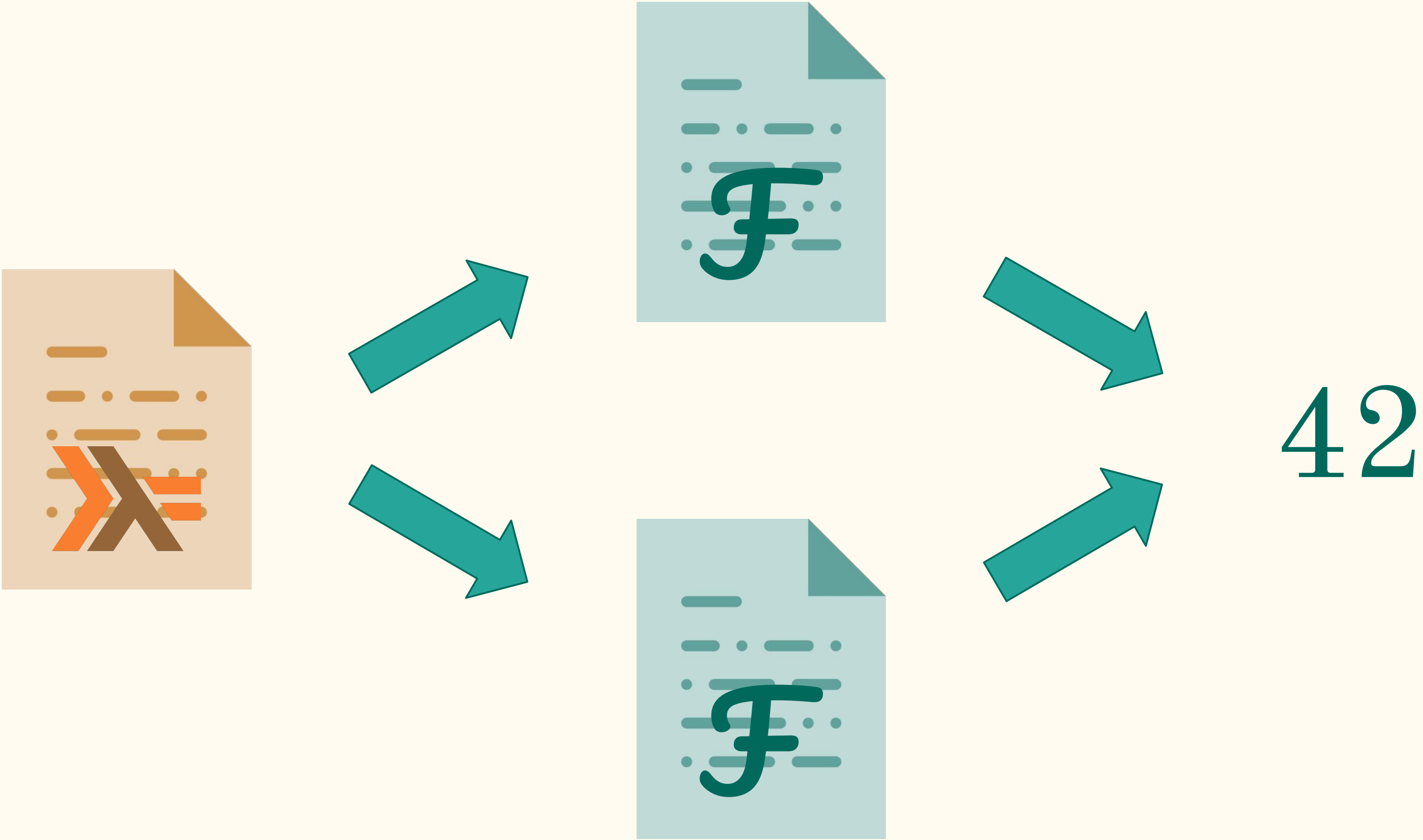

42

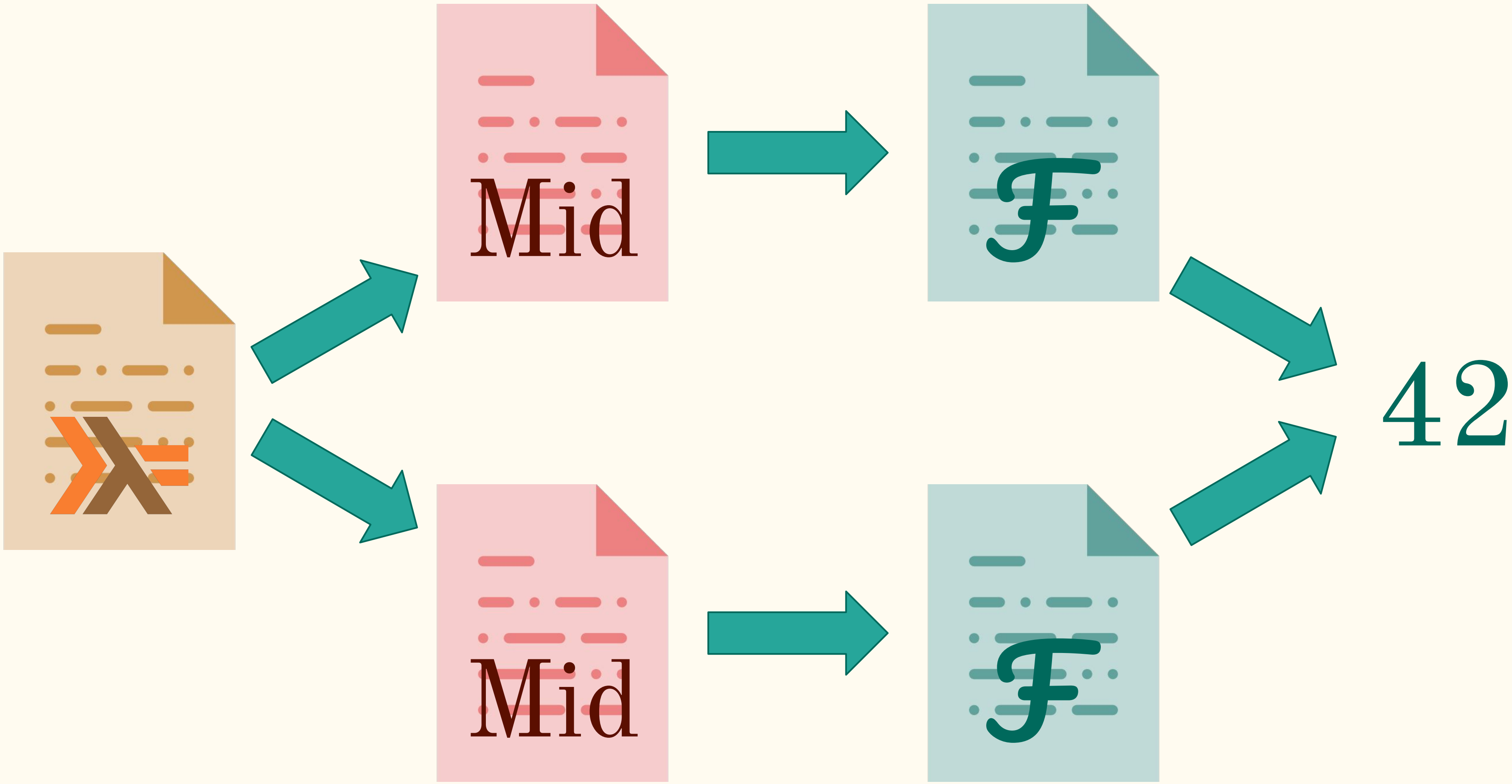


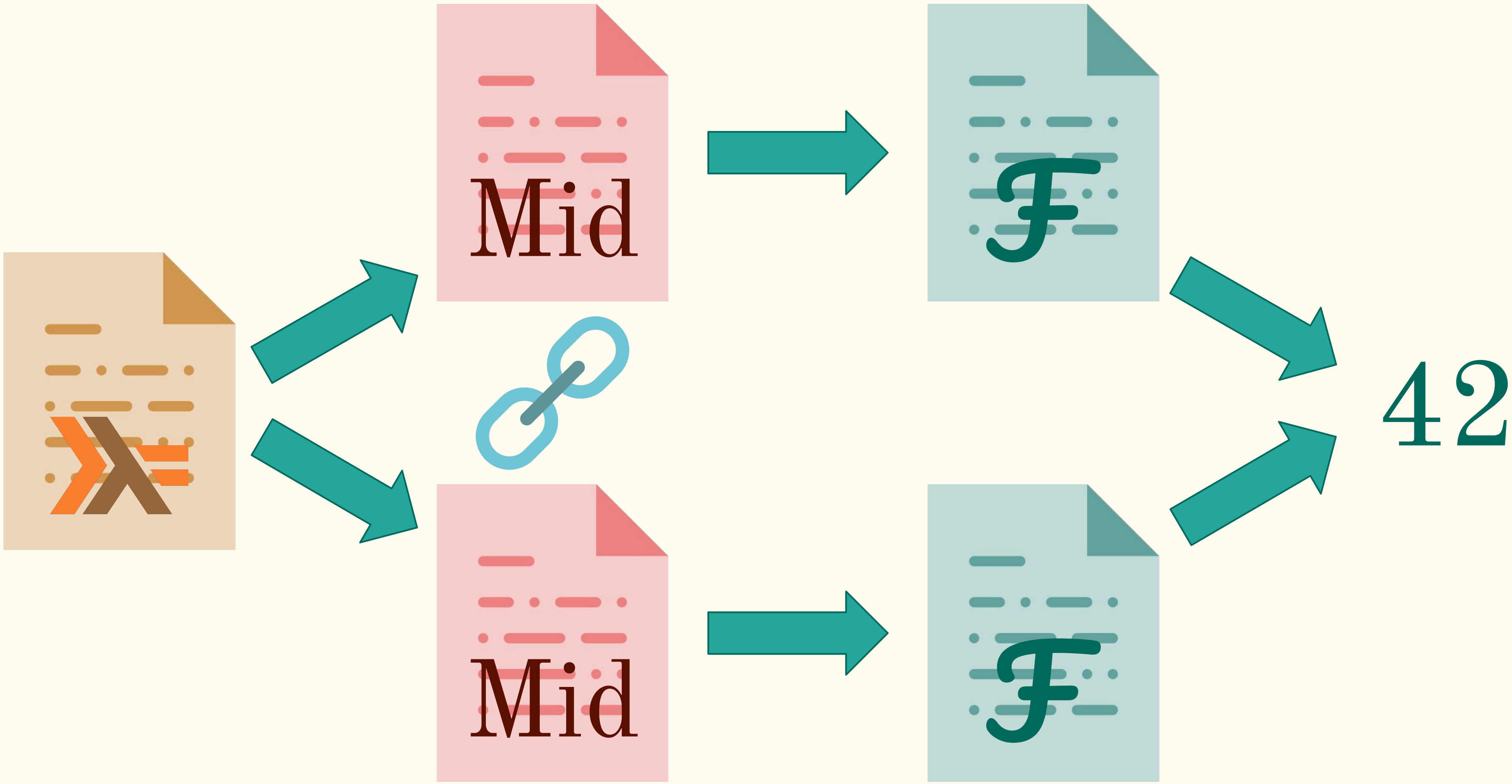



42

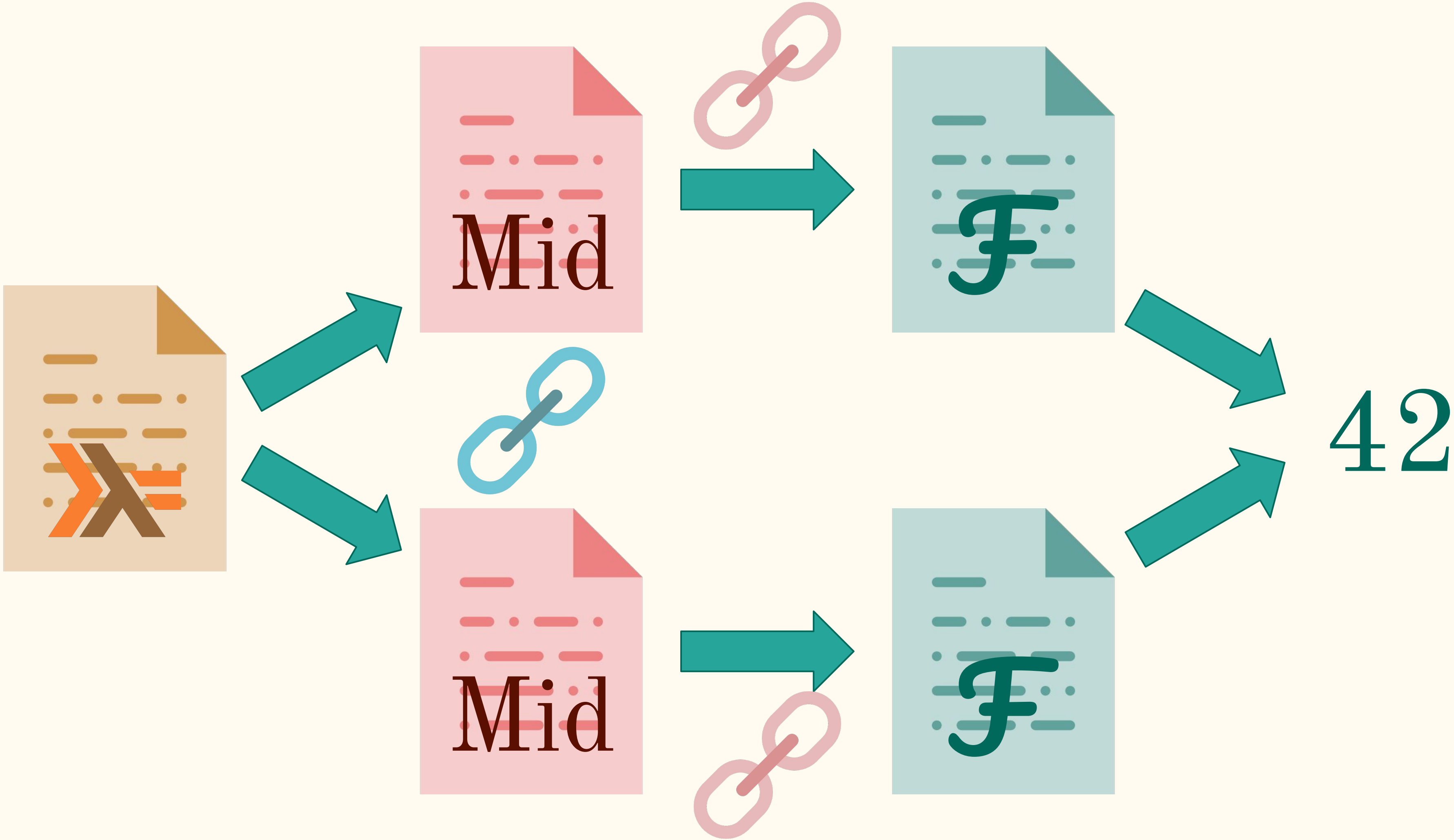

42

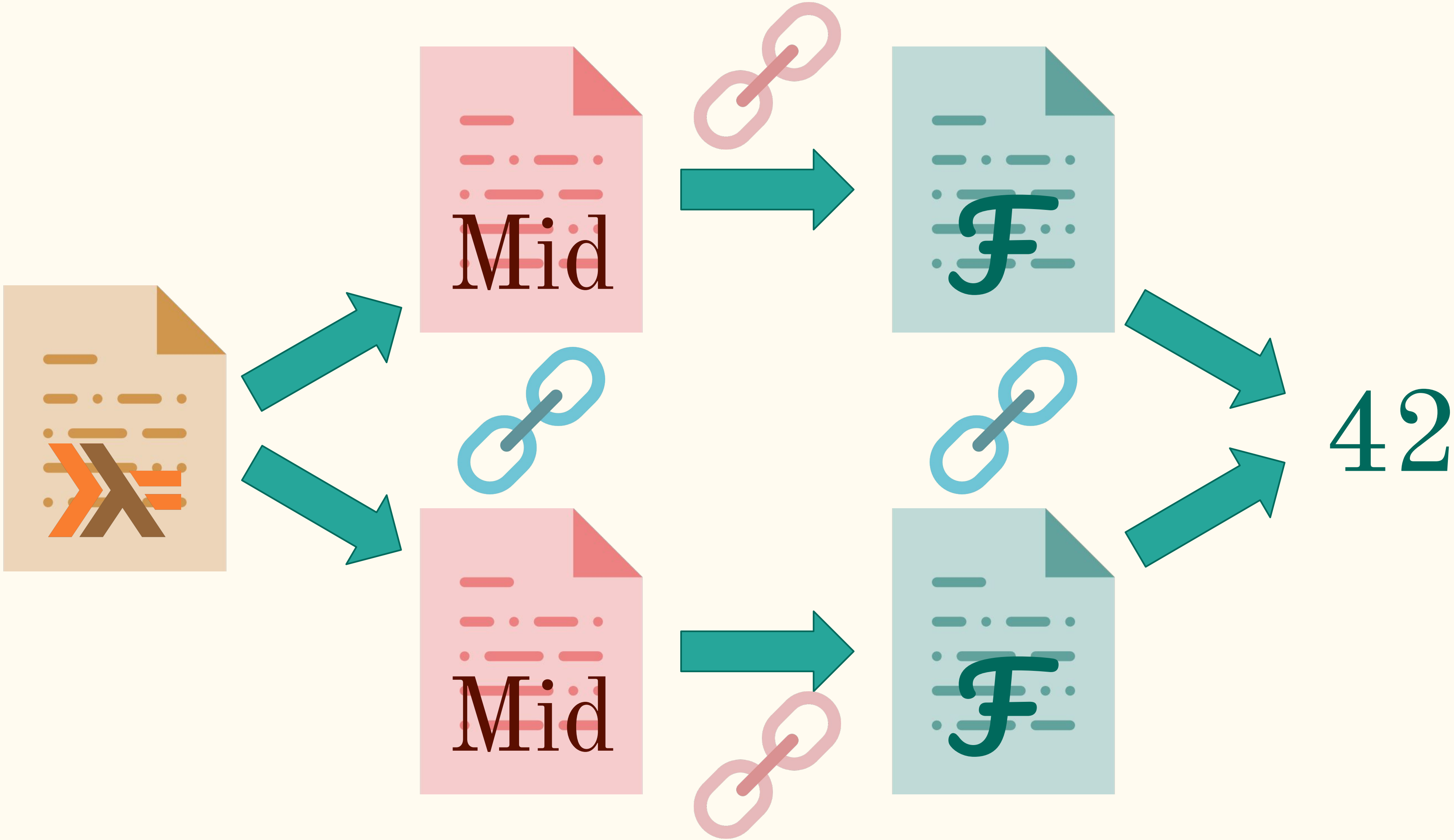

42

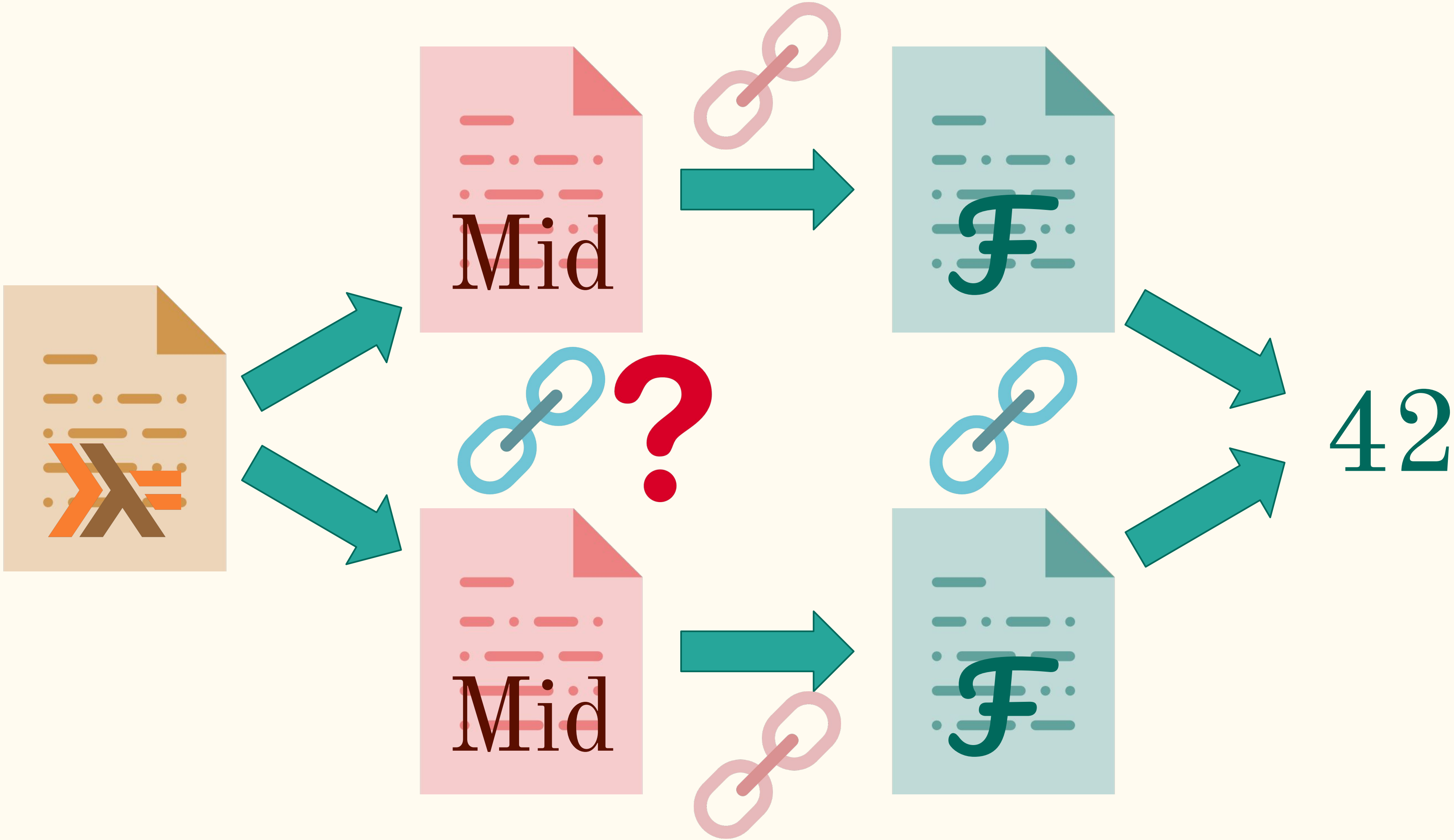

Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42



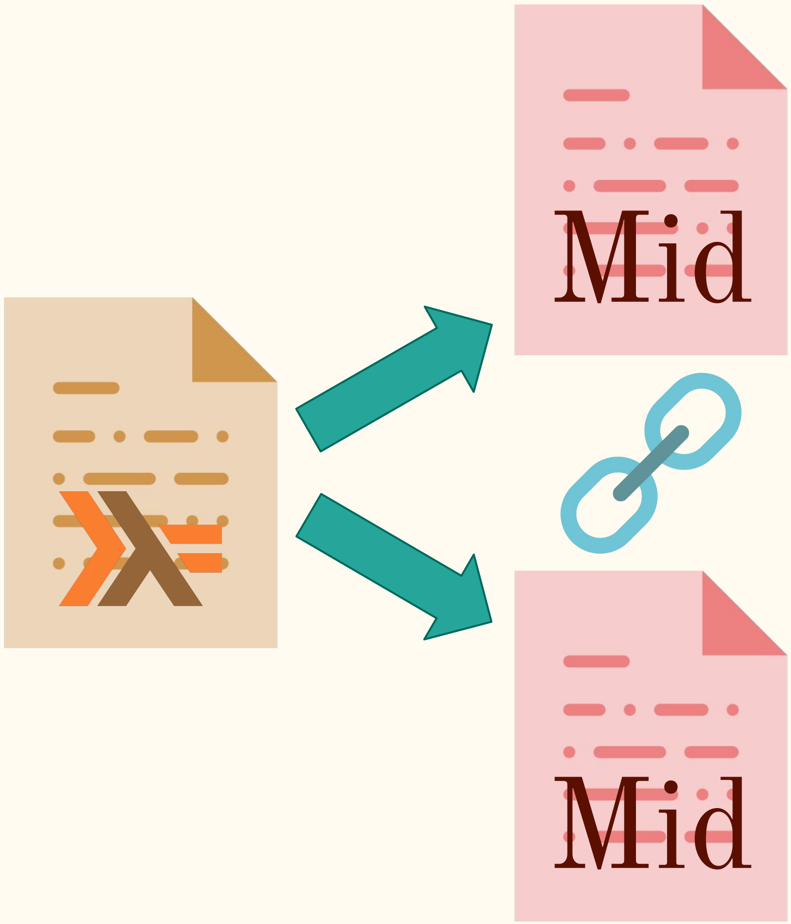
Mid
Mid
42

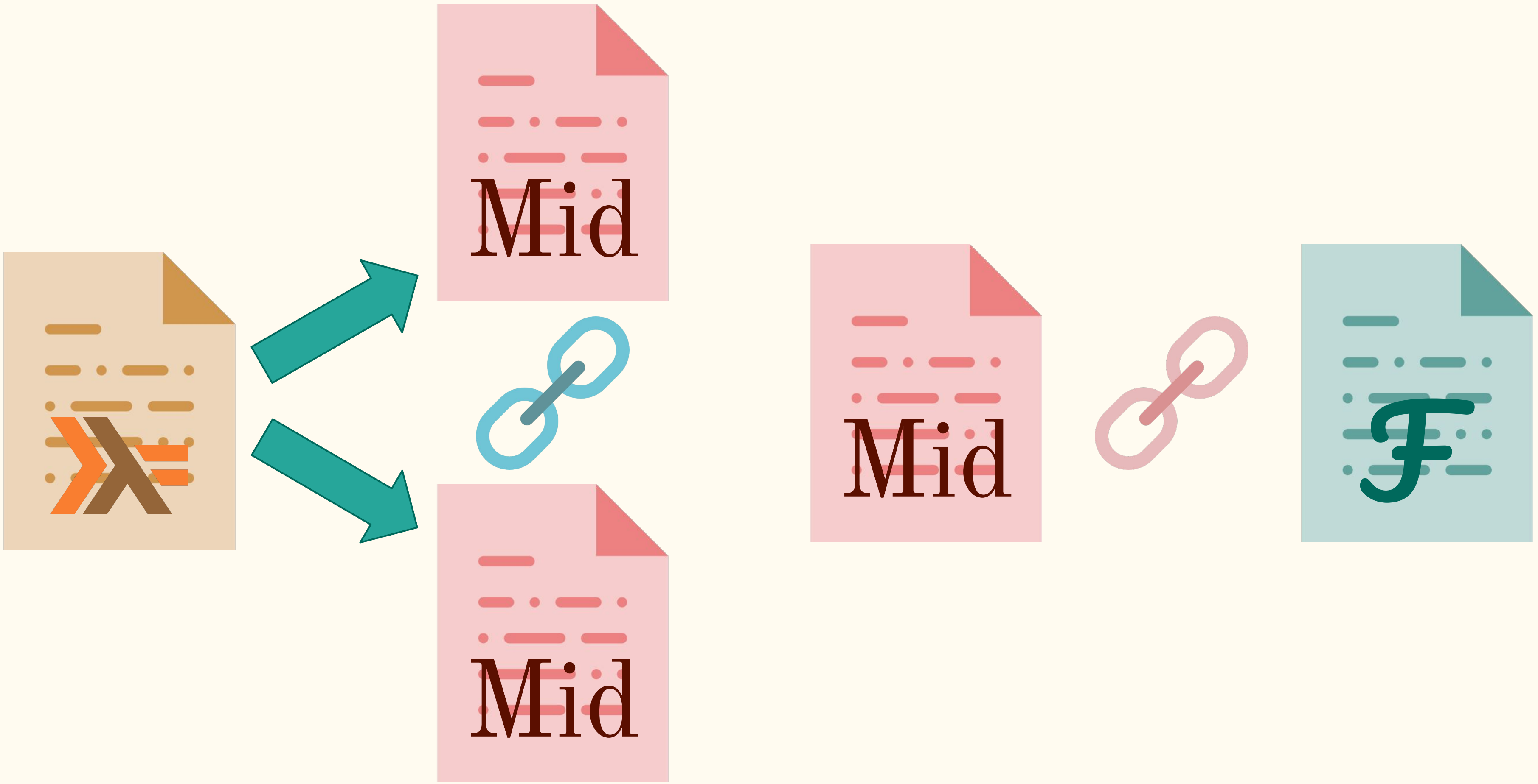

Mid
Mid
42


Mid
Mid
42

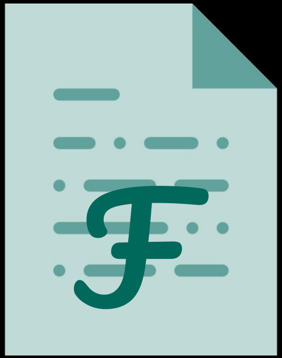

Mid
?
Mid
42

Intermediate Language

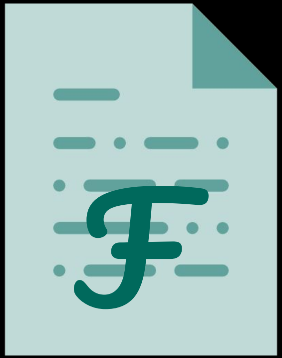

Mid
Mid


Mid
Mid
Mid
ℱ


Mid



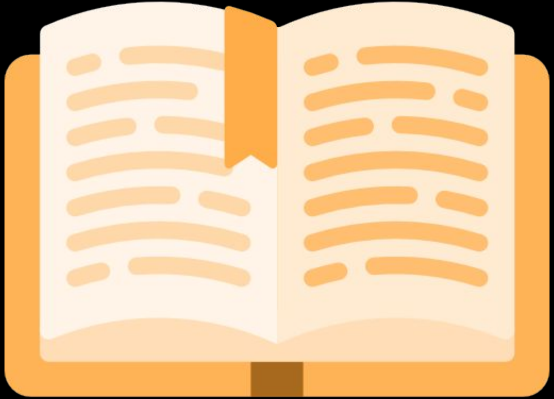

Mid


$\mathcal{F}$

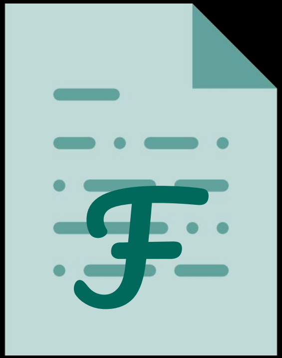

Dict

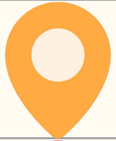

Mid

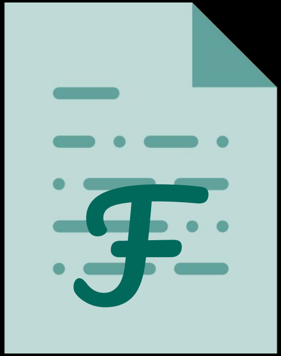

$\mathcal{F}$




Mid

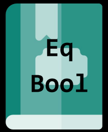


Pointer	Implem.
	...
	...
	...



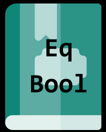
Fundamental Theorem of Software Engineering

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

(==) 

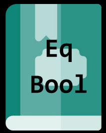
True True

`(==)`  `True True`

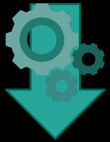
`(==)` 

`True True`

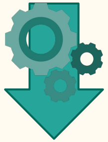
(==)  True True

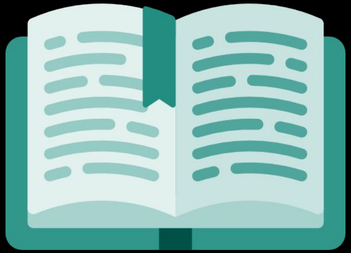
(==) 

True True



(==)  True True

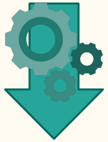


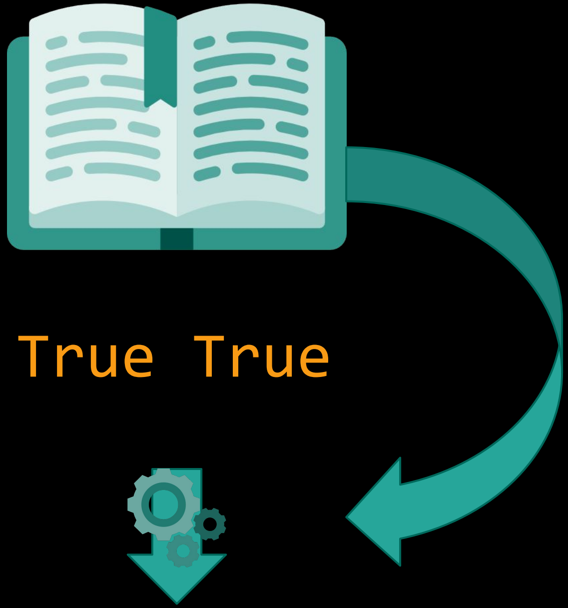


(==) 

True True

(==)  True True





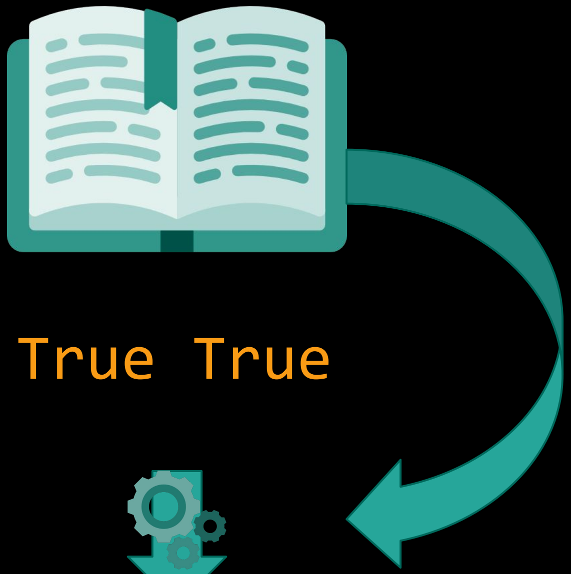
(==) 

True True

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

Pointer	Implem.
(pin)	Eq Bool : …

(==) (pin) True True



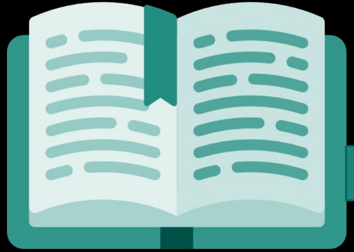
(==) Eq Bool True True

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

Pointer	Implem.
(pin icon)	Eq Bool : …

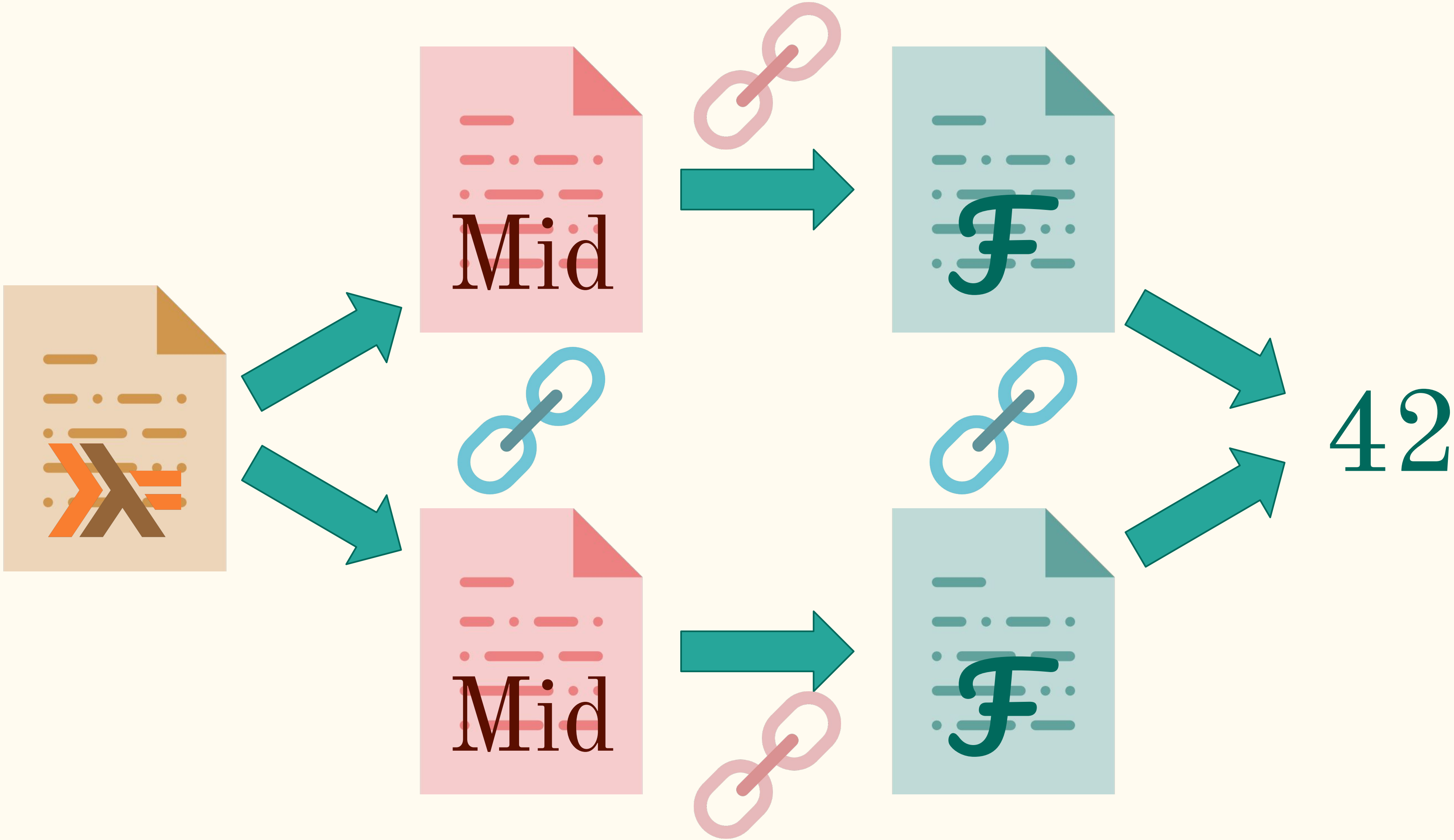
```
(==) 📍 True True
```

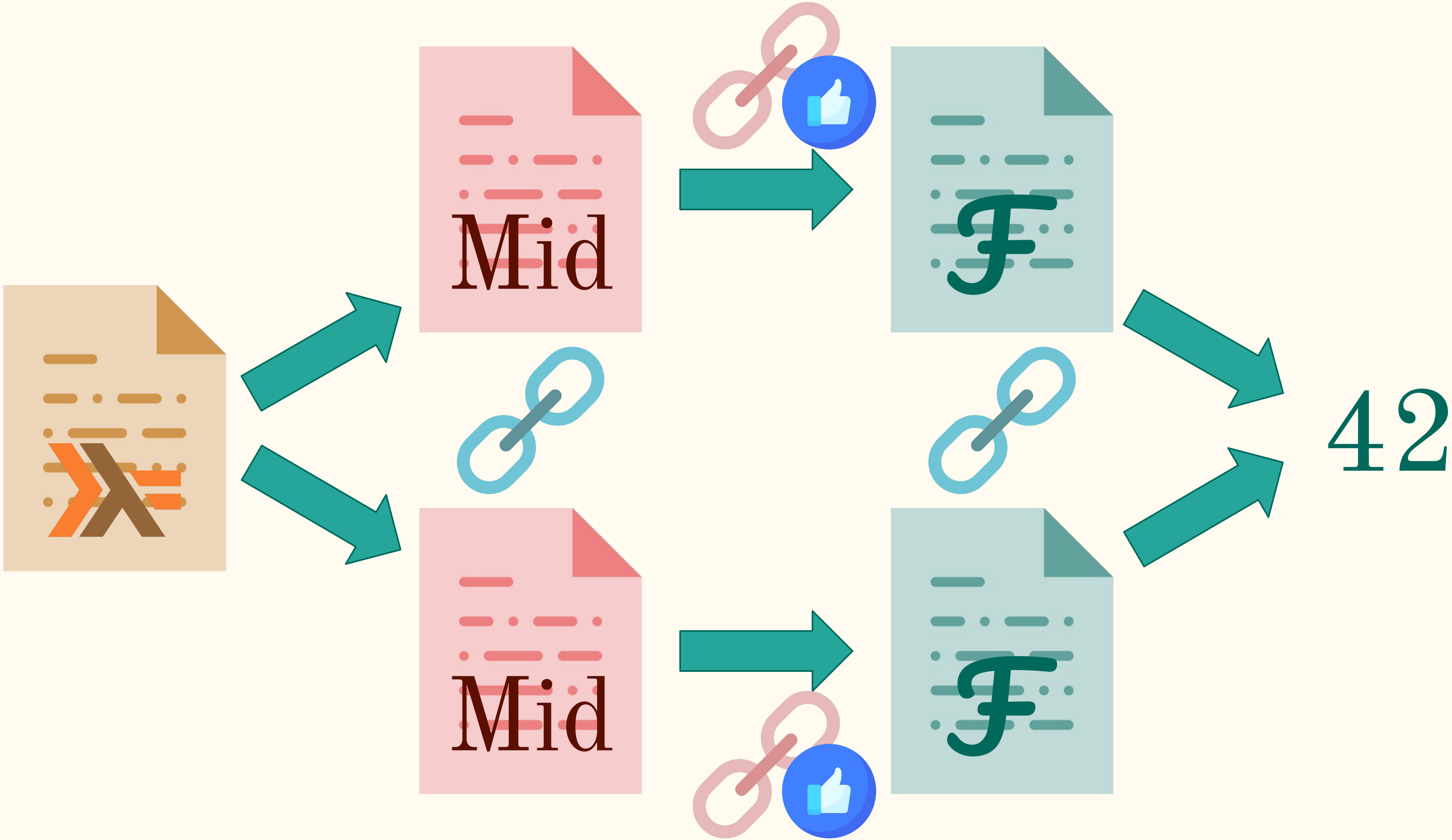
```
(\x : Bool. \y : Bool. case x,y of
  True ,True  -> True
  False,False -> True
  _    ,_     -> False)
True True
```



```
(==) [Eq Bool] True True
```

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


Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42


Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42

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

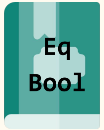
```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


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

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


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

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

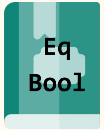

```
not :: Eq Bool => Bool -> Bool
not b = b == False
```

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




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

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



¯_(ツ)_/¯


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

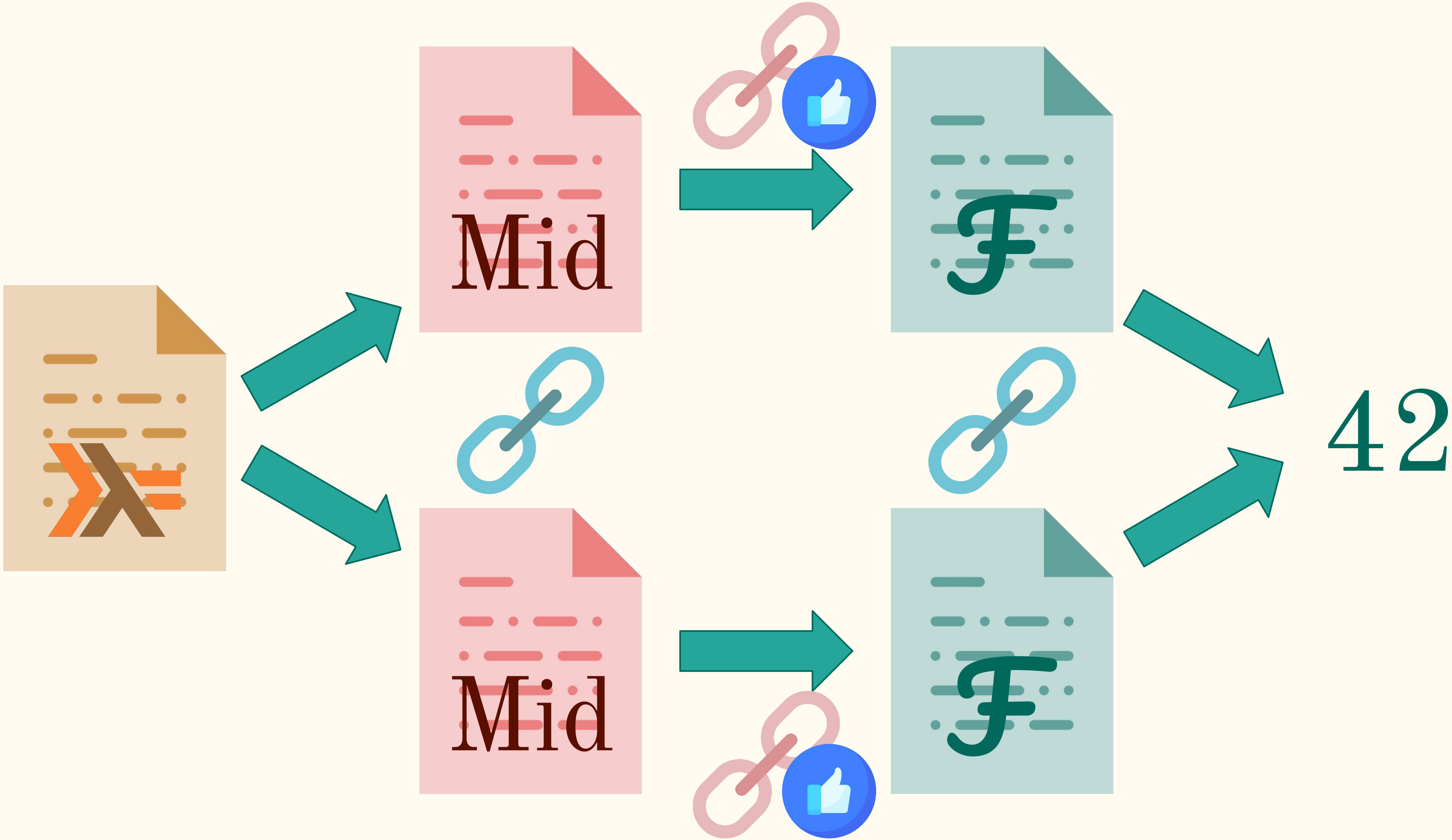
```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


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

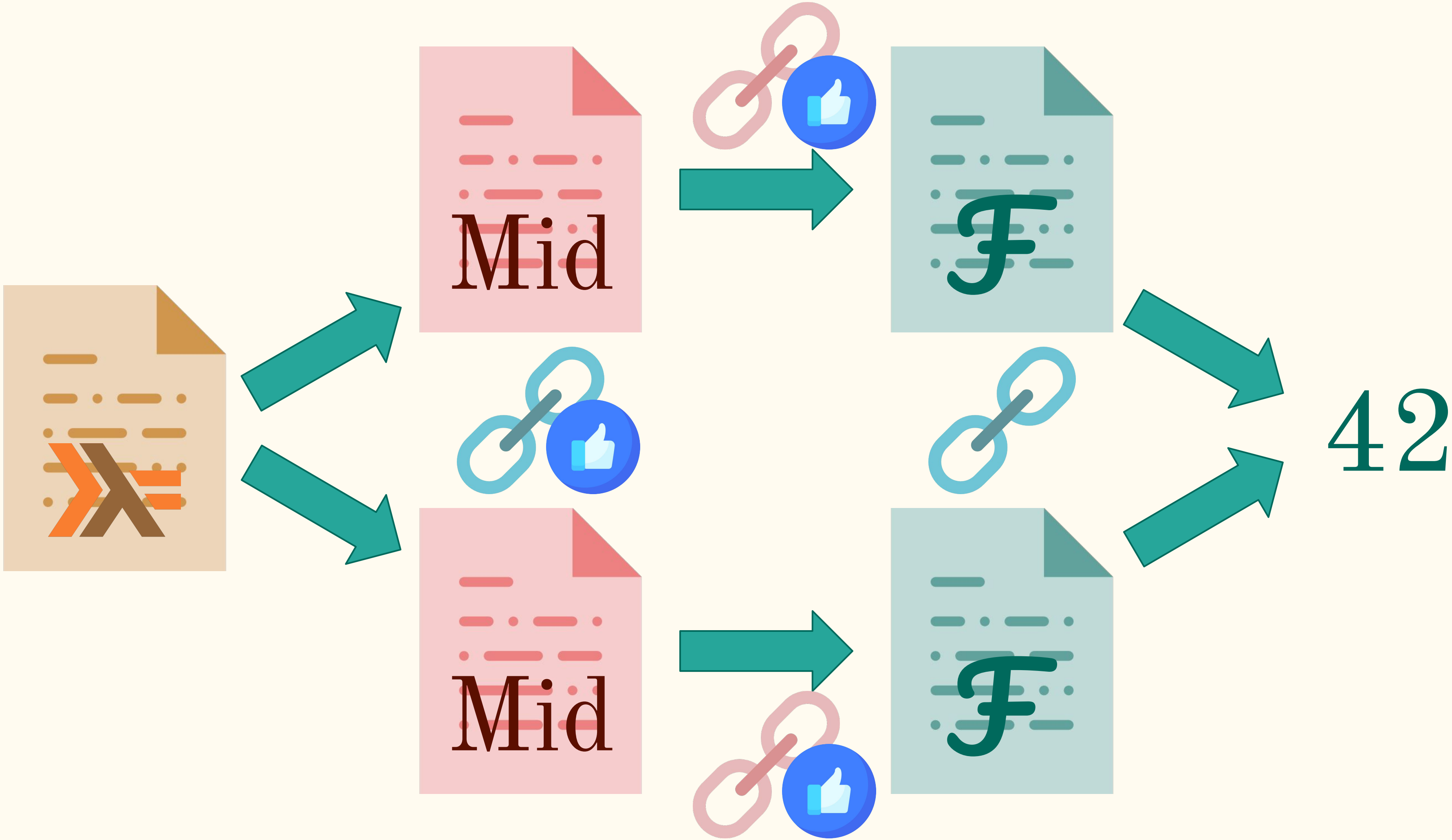
```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False


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

Pointer	Implem.
	Eq Bool : ...

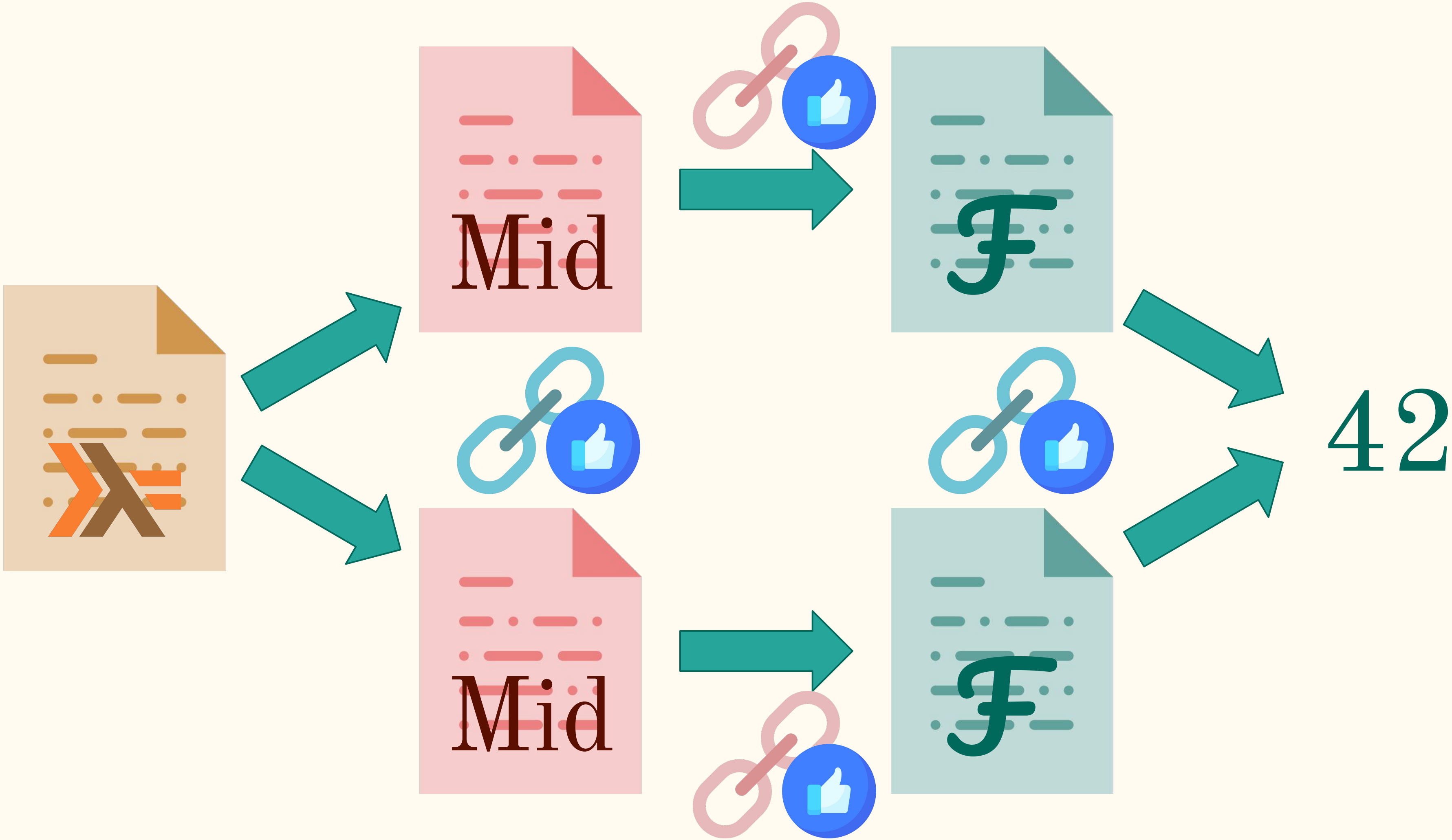
```
inst Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

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

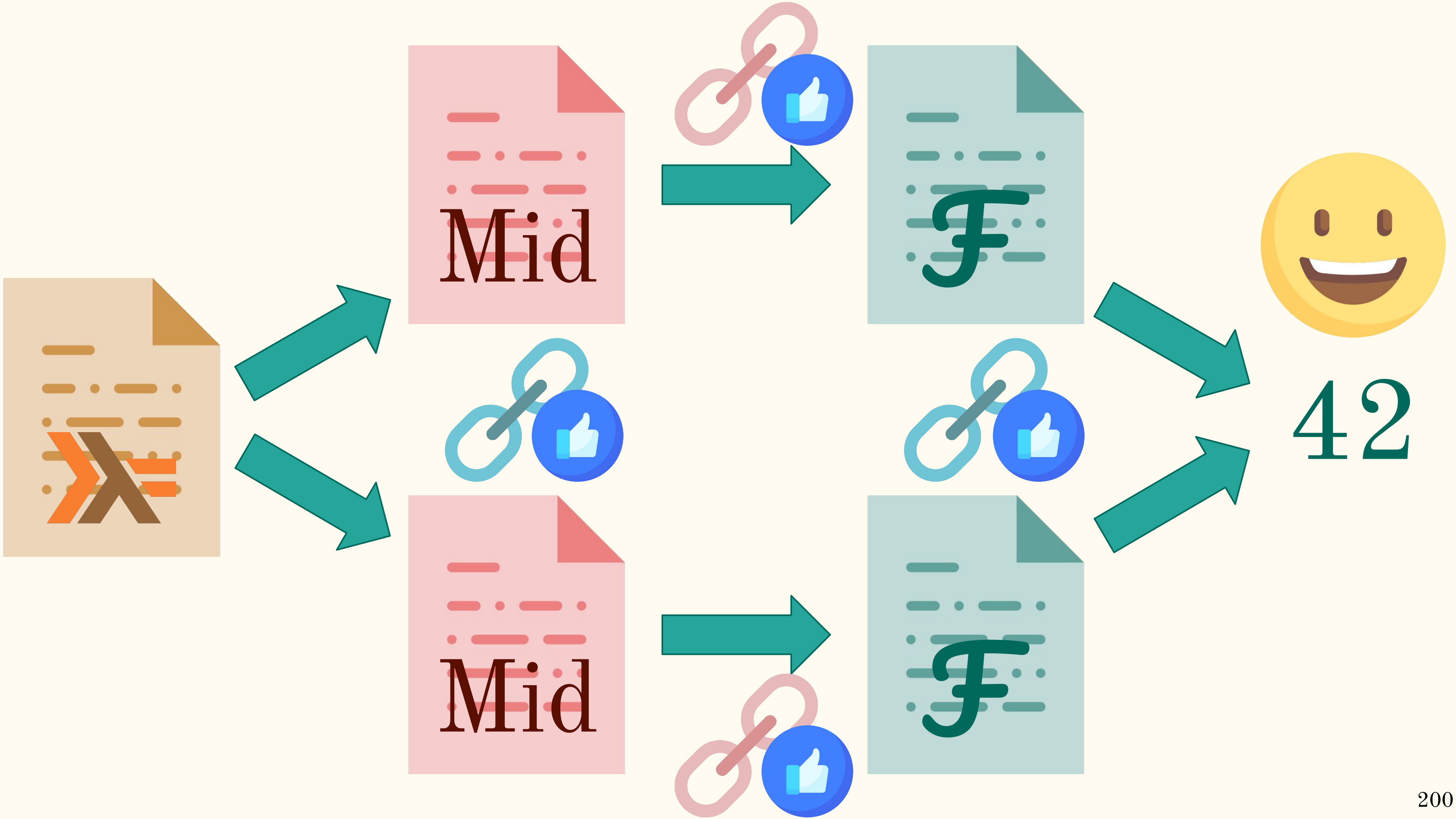
Pointer	Implem.
	Eq Bool : ...


Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42



Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42


Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42


Mid
Mid
$\mathcal{F}$
$\mathcal{F}$
42

Future Work

Image by Arkarti from DeviantArt

From the rule premise:

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

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

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

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:

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

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

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

By Preservation (Theorem 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

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

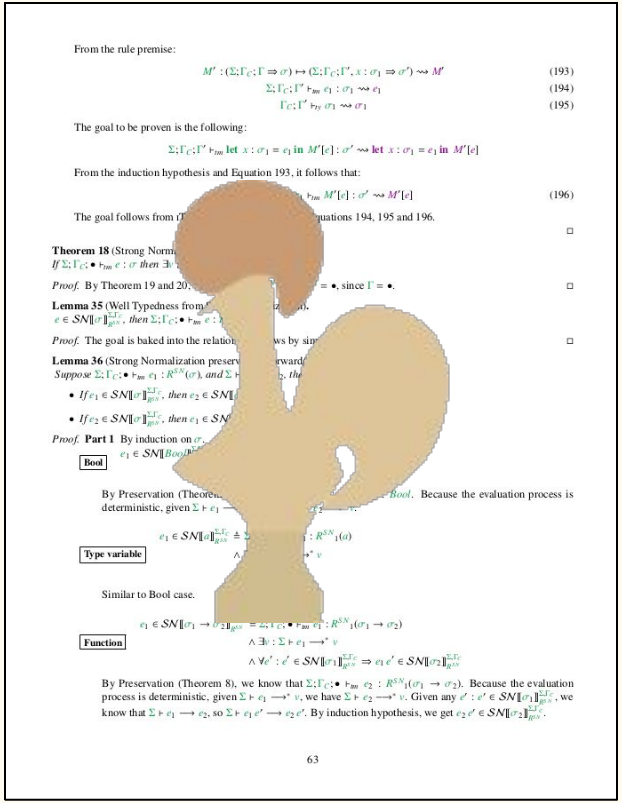
Similar to Bool case.

Function

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

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 Norm...
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 relation ... ws by sim... □

Lemma 36 (Strong Normalization preserv... rward...
Suppose $\Sigma;\Gamma_C;\bullet \vdash_{tm} e_1 : R^{SN}(\sigma)$*, and* $\Sigma \vdash$ ... *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 (Theorem ... $Bool$. Because the evaluation process is deterministic, given $\Sigma \vdash e_1 \longrightarrow$ ...

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

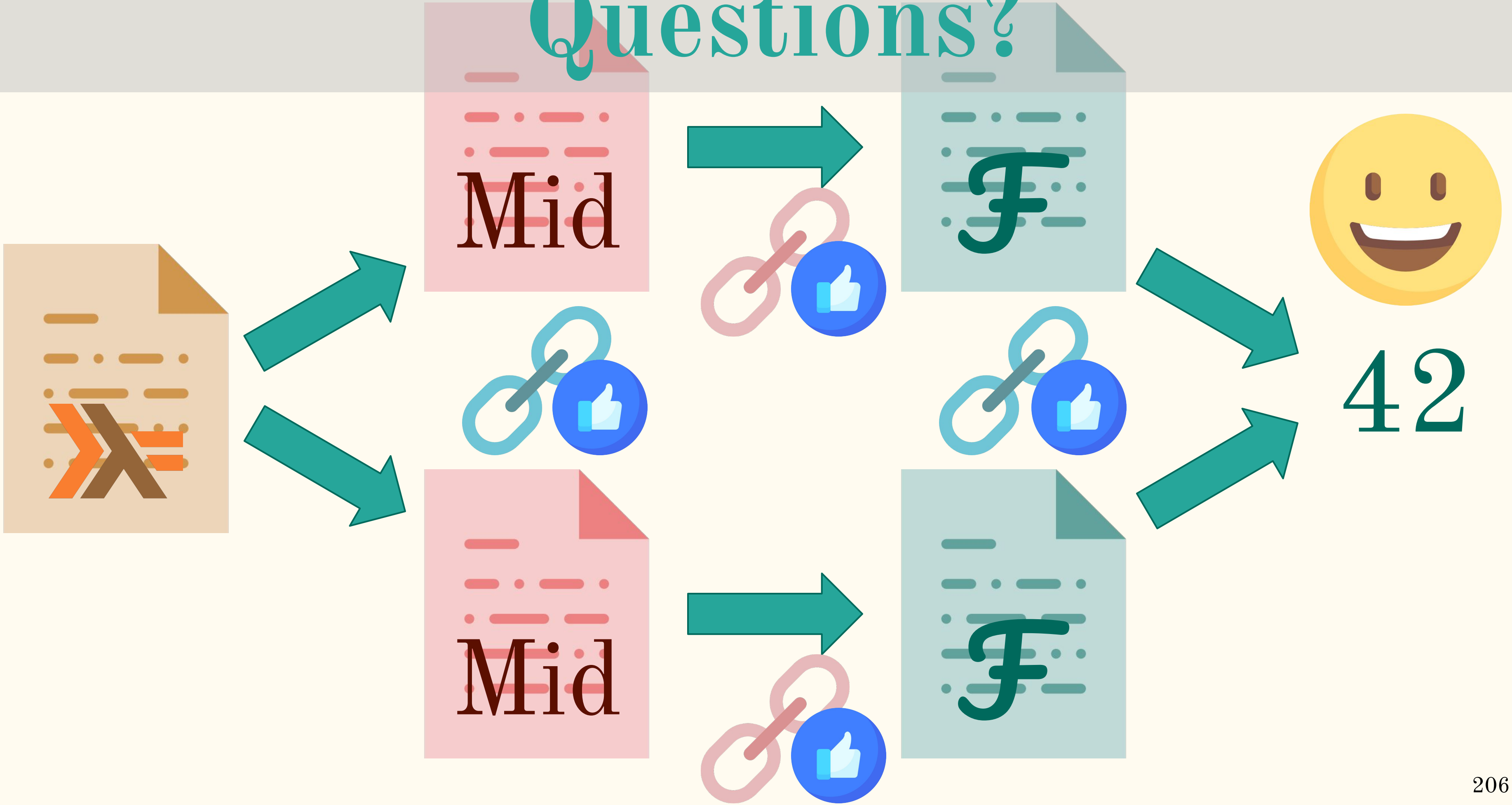
Similar to Bool case.

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, Pixel perfect, Smashicons & Vectors Market from www.flaticon.com

Logical relations example edited from Types and Programming Languages, B. Pierce, MIT Press, 2002