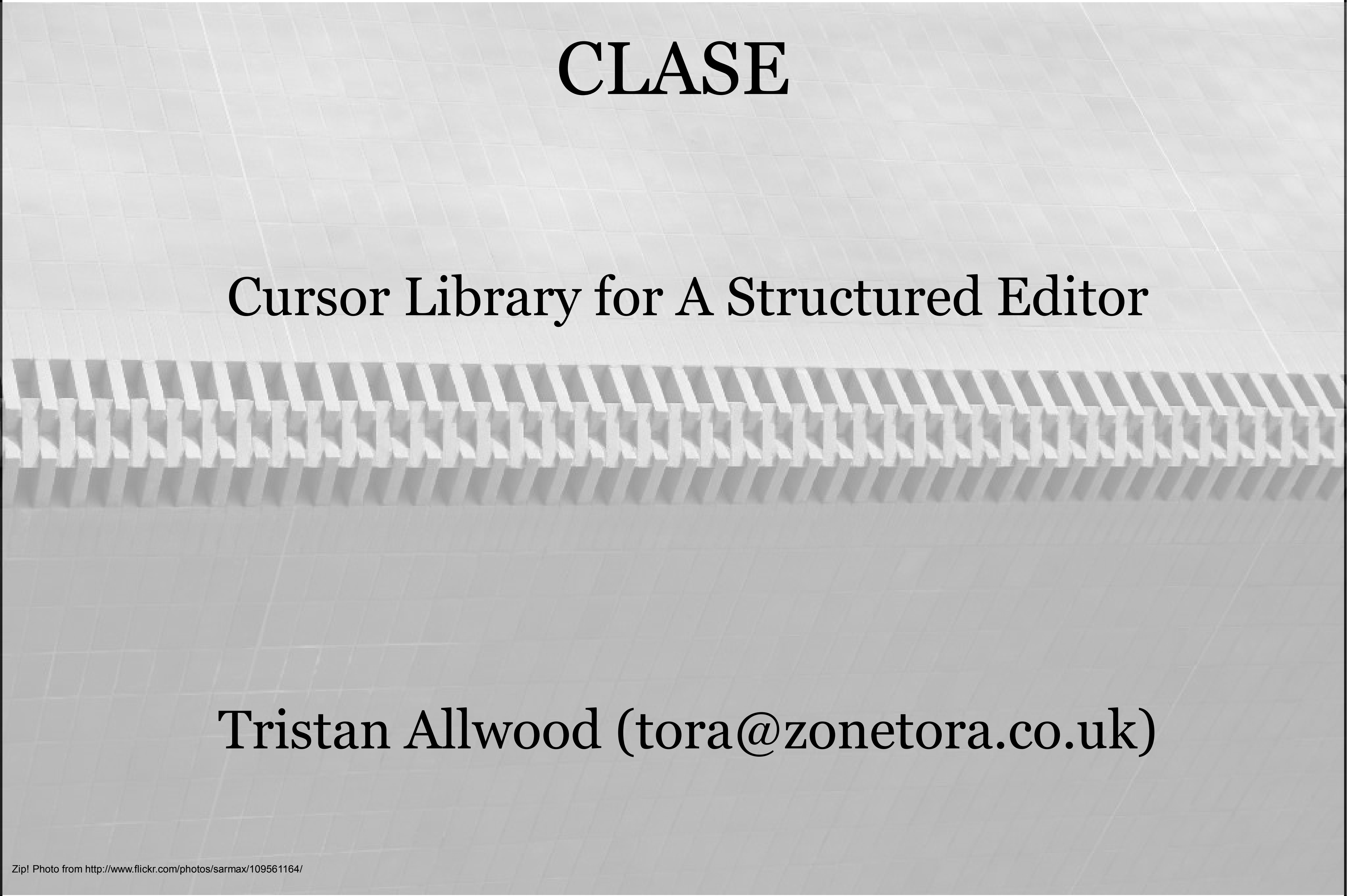


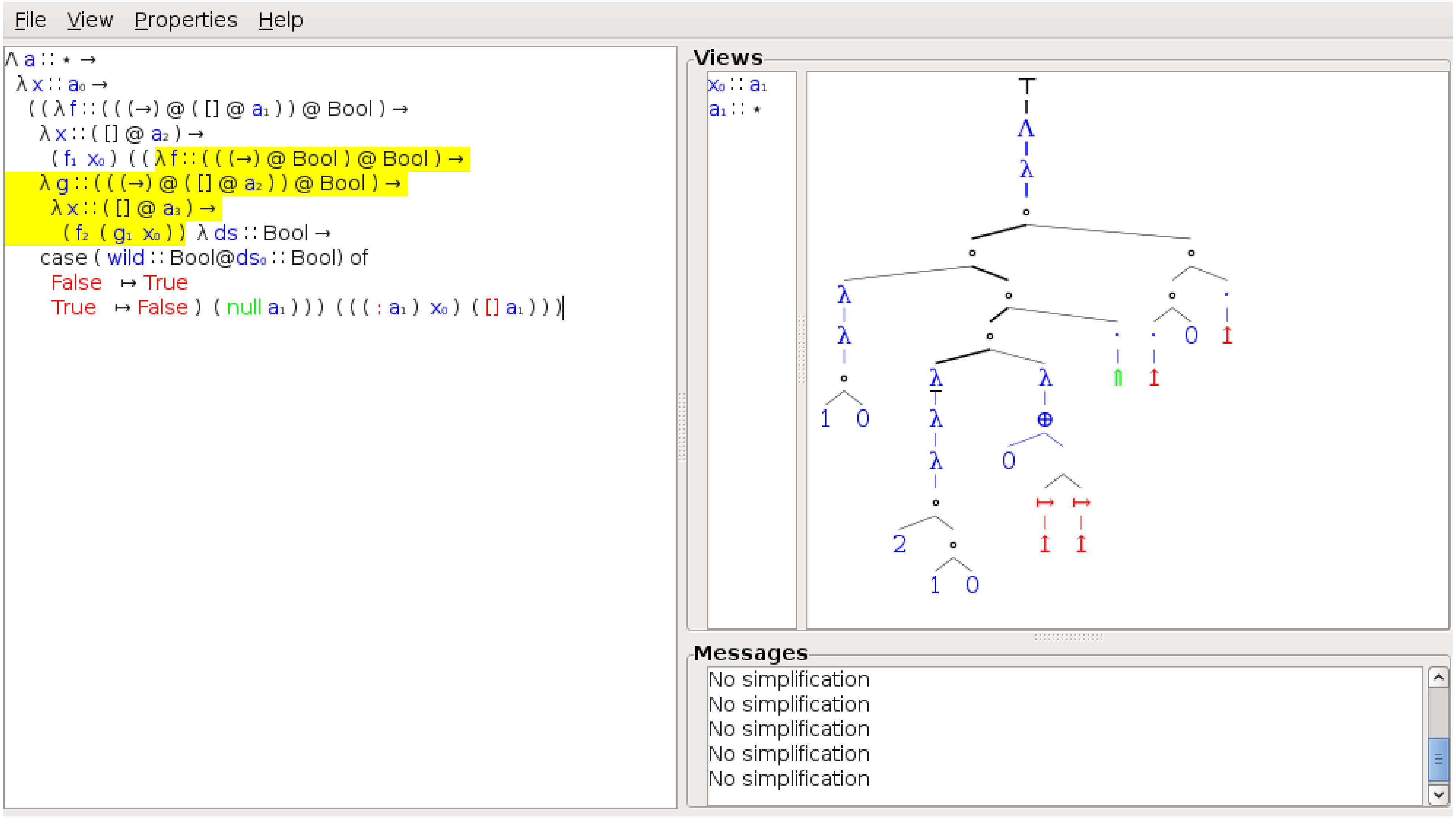
CLASE

Cursor Library for A Structured Editor

Tristan Allwood (tora@zonetora.co.uk)

Zip! Photo from http://www.flickr.com/photos/sarmax/109561164/

Motivation



```
Λ a :: ⋆ →
 λ x :: a₀ →
  ( ( λ f :: ( ( (→) @ ( [] @ a₁ ) ) @ Bool ) →
   λ x :: ( [] @ a₂ ) →
    ( f₁  x₀ )  ( ( λ f :: ( ( (→) @ Bool ) @ Bool ) →
   λ g :: ( ( (→) @ ( [] @ a₂ ) ) @ Bool ) →
    λ x :: ( [] @ a₃ ) →
     ( f₂  ( g₁  x₀ ) )  λ ds :: Bool →
   case ( wild :: Bool@ds₀ :: Bool) of
    False   ↦ True
    True   ↦ False )  ( null a₁ ) ) )  ( ( ( : a₁ )  x₀ )  ( [] a₁ ) ) )
```

```
x₀ :: a₁
a₁ :: ⋆
```

```
No simplification
No simplification
No simplification
No simplification
No simplification
```

Outline

Preliminary - GADTs

```haskell
data Tree a = Leaf | Branch (Tree a) a (Tree a)

data Tree a where
   Leaf :: Tree a
   Branch :: Tree a → a → Tree a → Tree a
```

```haskell
data Tree a where
   Leaf :: Tree a
   Branch :: Tree a → a → Tree a → Tree a
   IntLeaf :: Int → Tree Int
```

```haskell
flatten :: Tree a → [a]
flatten (IntLeaf int) = [int]
...
```

Polite Notice

This talk will feature code snippets!

Code a user has to write "Blue User"	Code that is in the CLASE library "Green Library"	Code that can be autogenerated with T.H. scripts "Generated Orange"

http://www.flickr.com/photos/alkalinezoo/2374201026/

http://www.flickr.com/photos/cambridgelib/2343211287/

http://www.flickr.com/photos/webel/76665500/

Towards Clase Zippers

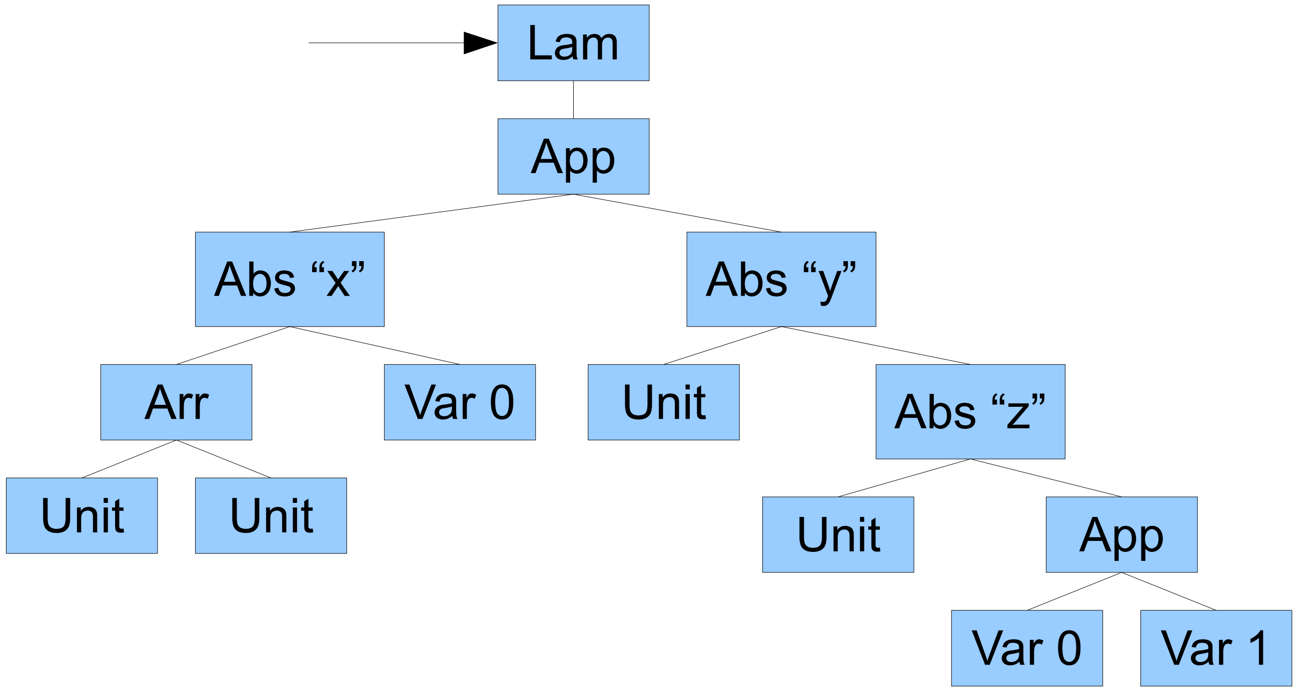
```
data Lam = Lam Exp

data Exp
  = Abs String Type Exp
  | App Exp Exp
  | Var Integer

data Type
  = Unit
  | Arr Type Type
```

Towards CLASE Zippers

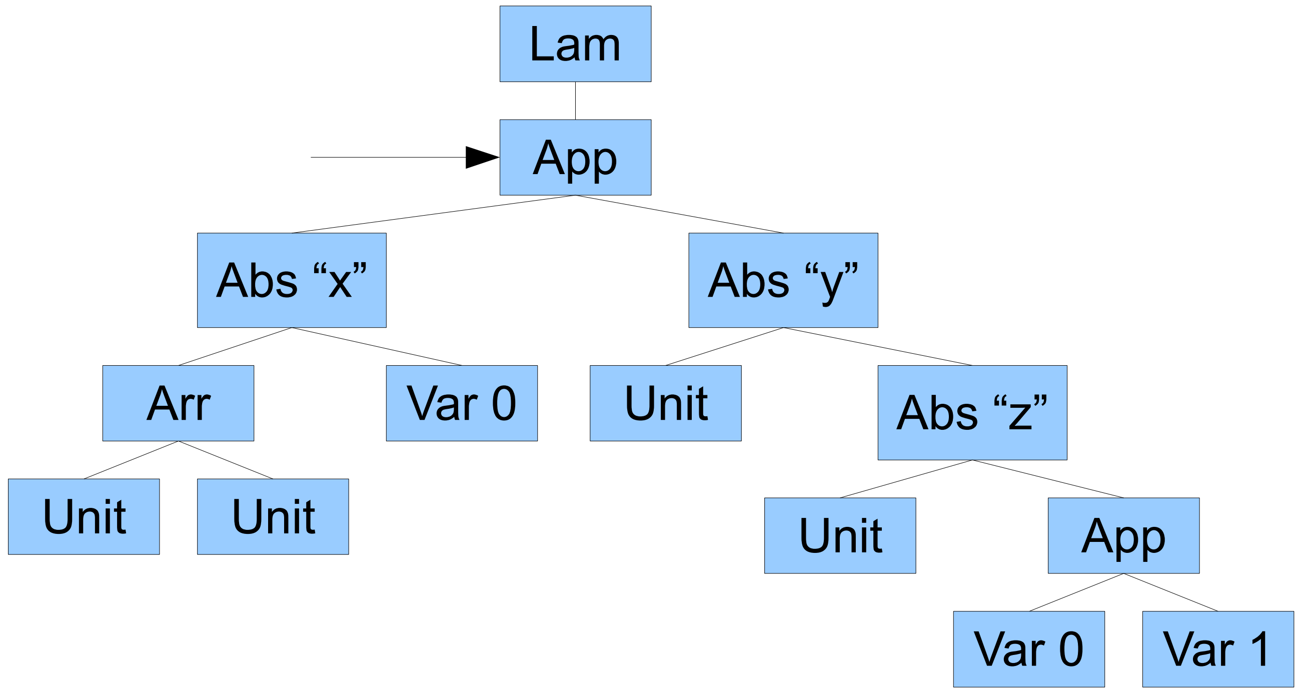
```
sample = Lam (
            App (Abs “x” (Unit `Arr` Unit) (Var 0))
                 (Abs “y” Unit
                    (Abs “z” Unit
                       (App (Var 0)
                            (Var 1)))))
```

$(\lambda x{:}\tau \rightarrow \tau . x)(\lambda y{:}\tau . \lambda z{:}\tau . (z\ y))$

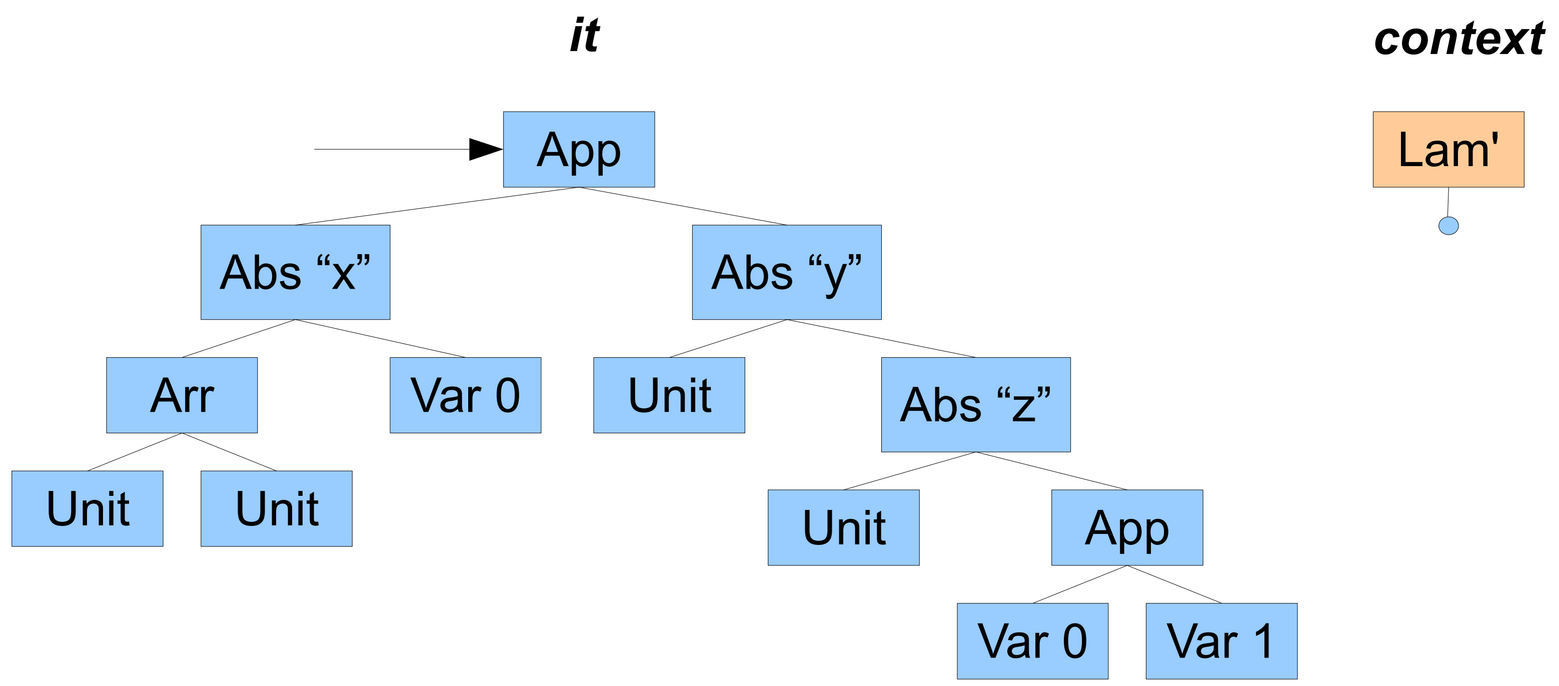
Towards CLASE Zippers



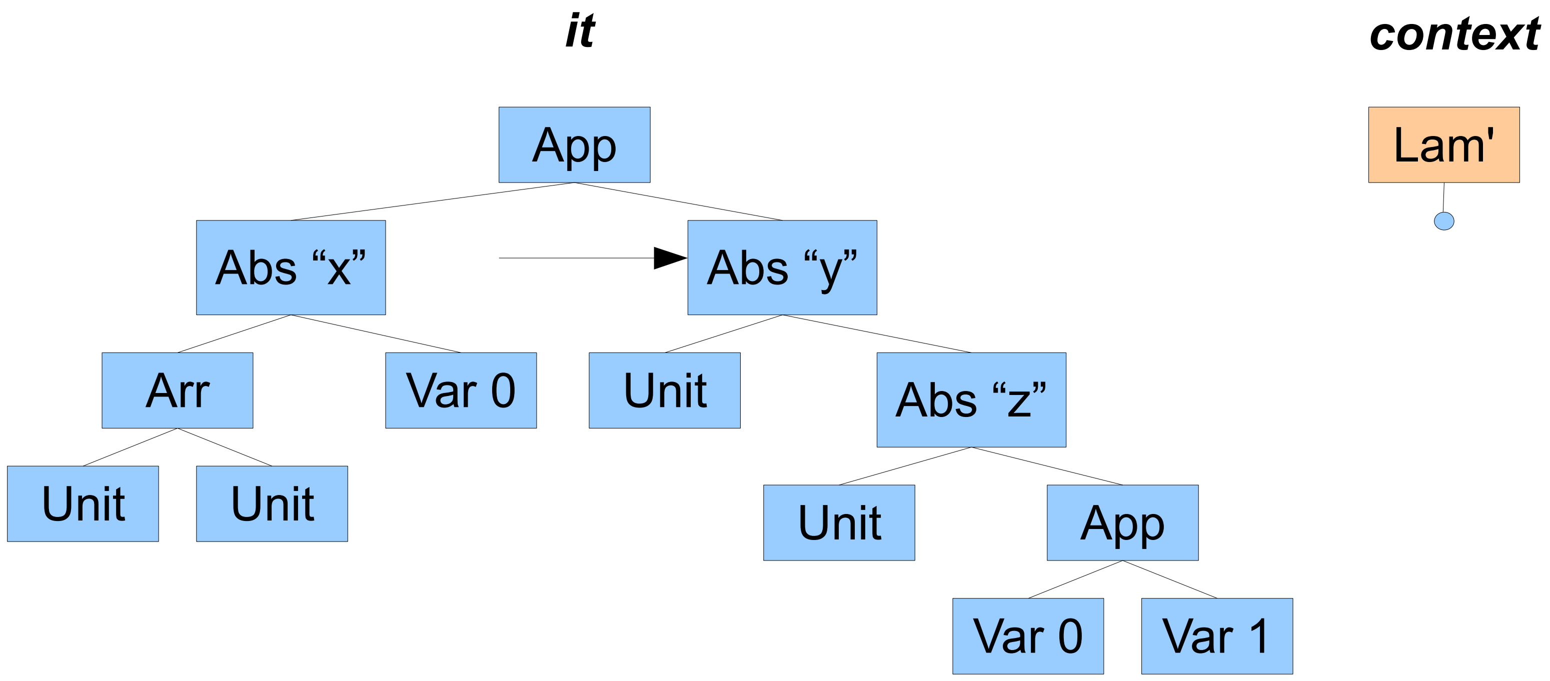
Towards CLASE Zippers



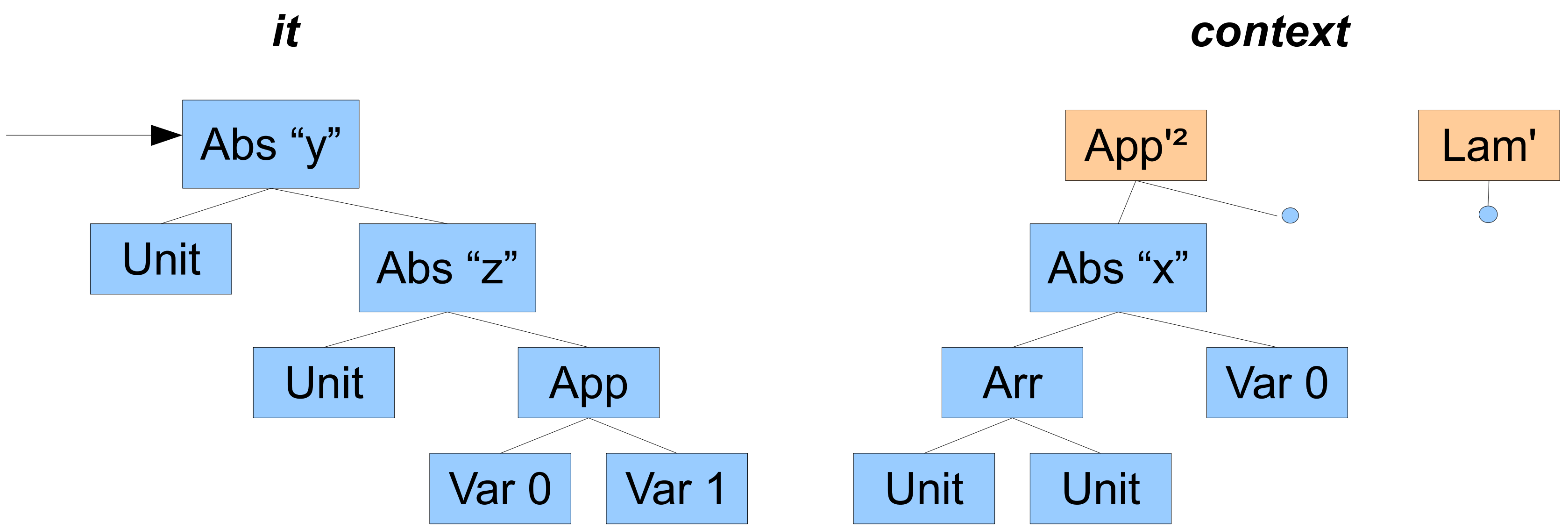
Towards CLASE Zippers



Towards CLASE Zippers

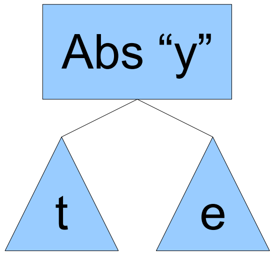


Towards CLASE Zippers

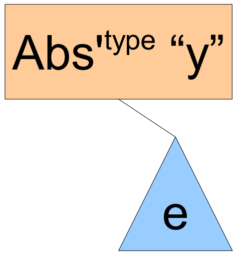


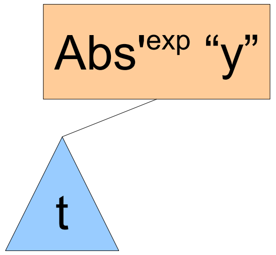
Single Contexts

```
data Exp
   = Abs String Type Exp
    ...
```

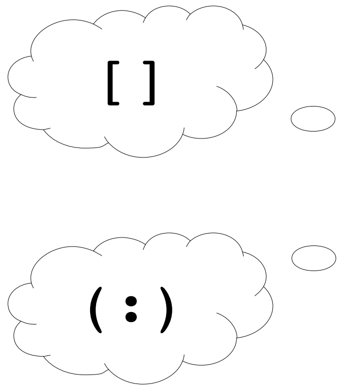


```
data ContextI from to where
   TypeToAbs :: String →          Exp  → ContextI Type Exp
   ExpToAbs  :: String → Type          → ContextI Exp Exp
    ...
```

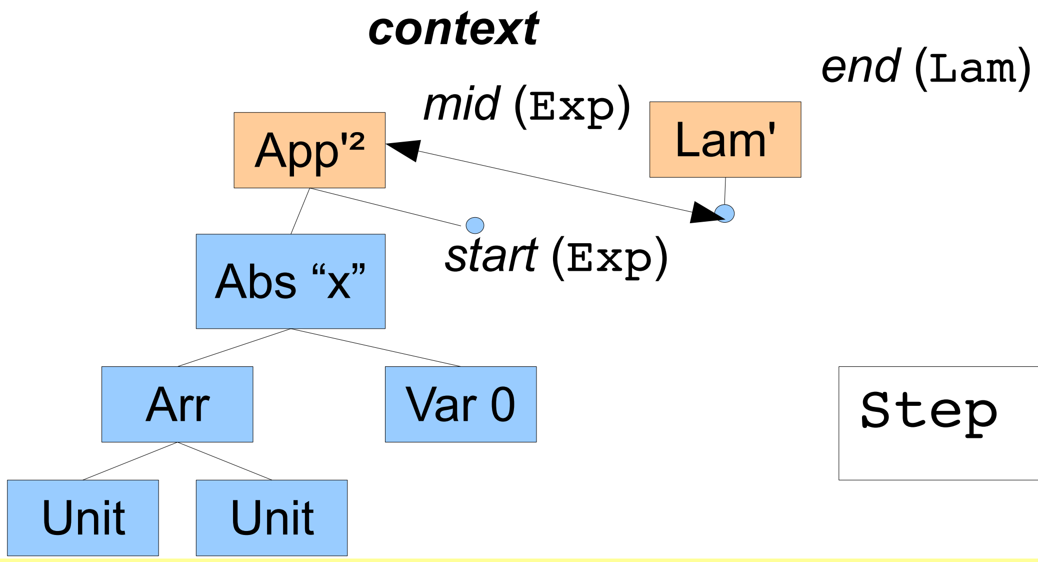




Chaining Contexts



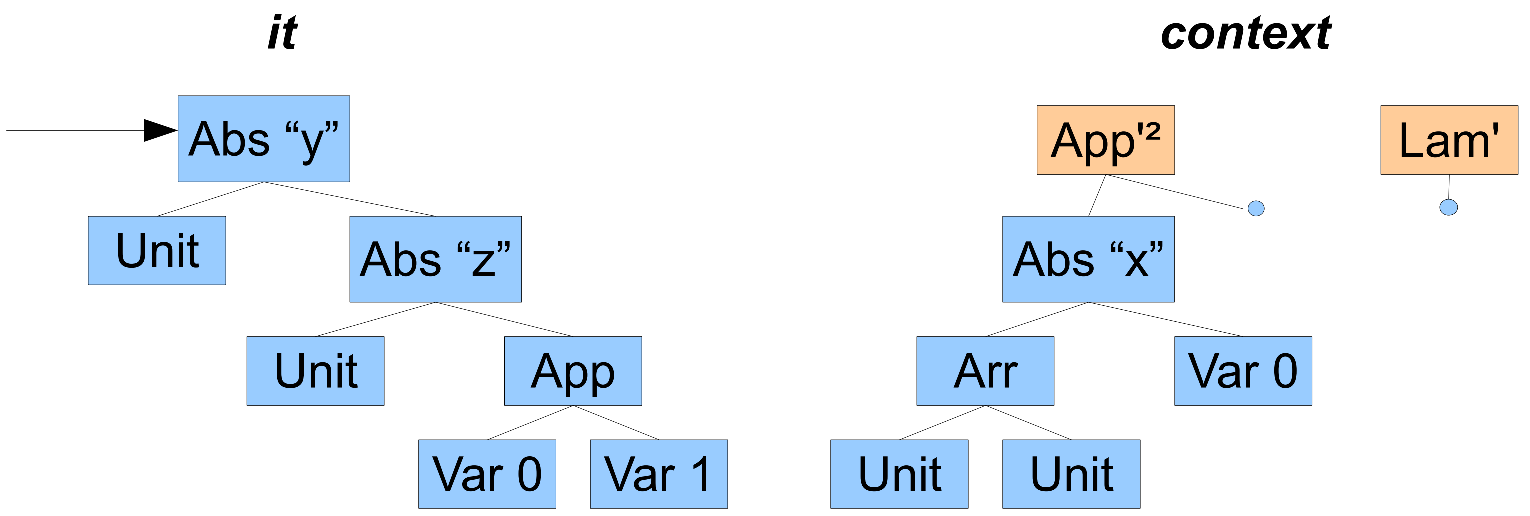
```
data Path start end where
  Stop :: Path here here
  Step :: ContextI start mid →
                Path mid end →
                Path start end
```



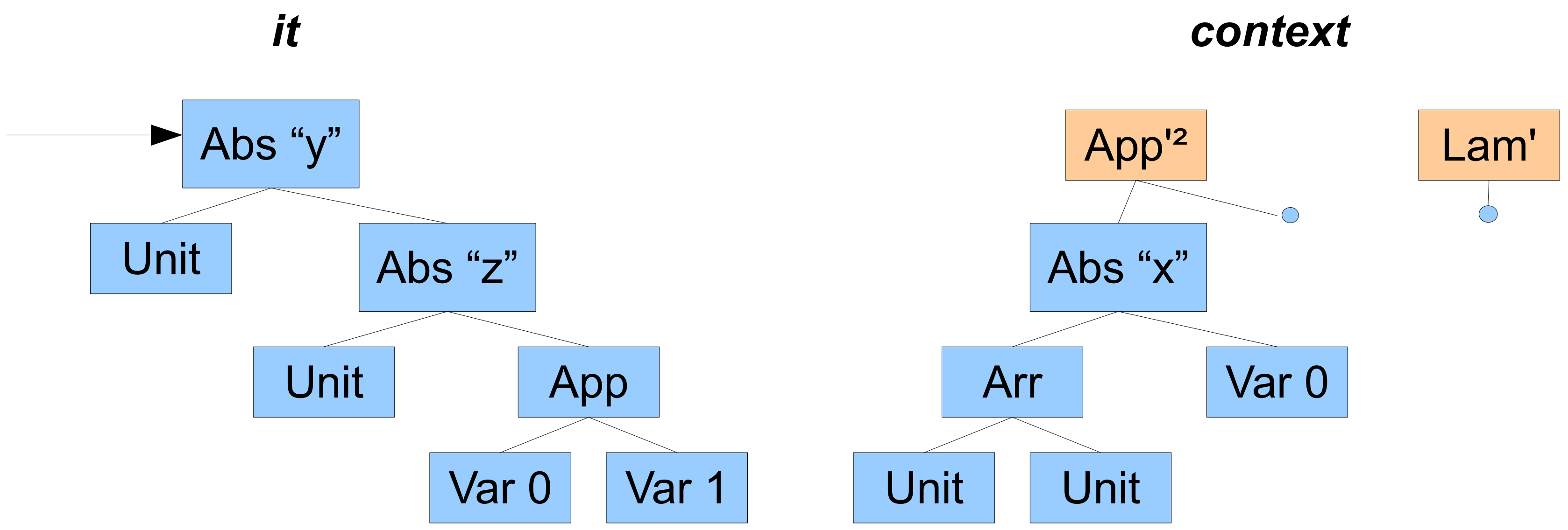
```
Step (App'² ...) (Step Lam' Stop)
```

A Cursor

```
data Cursor a = Cursor {
    it :: a,
    ctx :: Path a Lam
  }
```

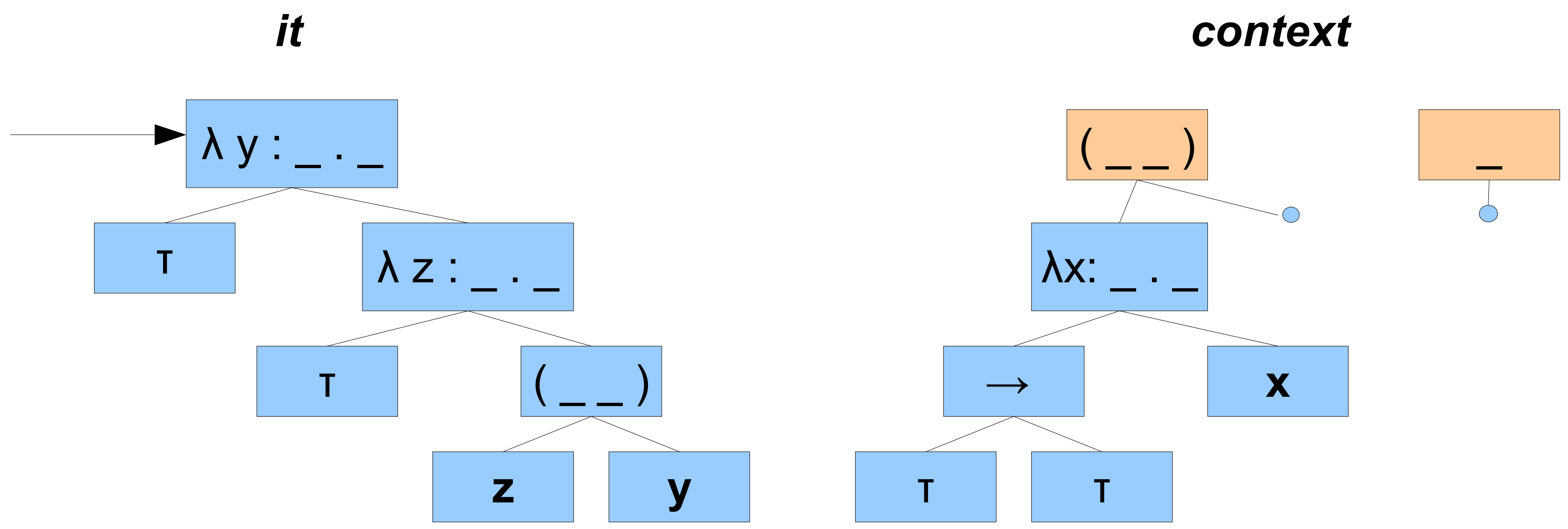


Rendering Problem



(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem



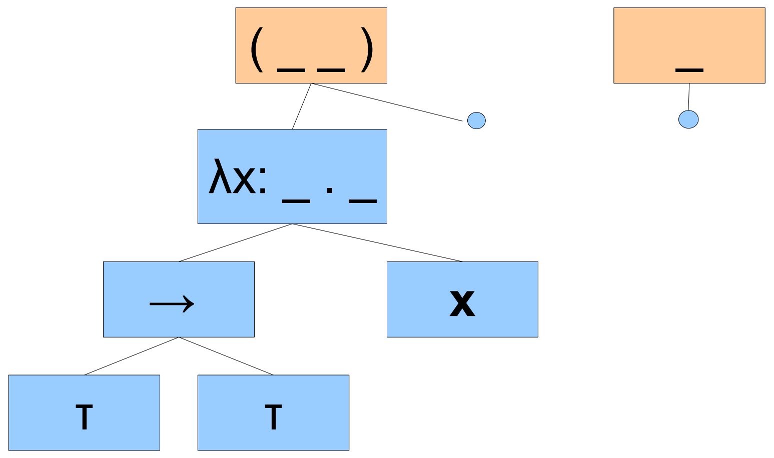
(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

it

$\longrightarrow$ λy:τ.λz:τ.(z y)

context



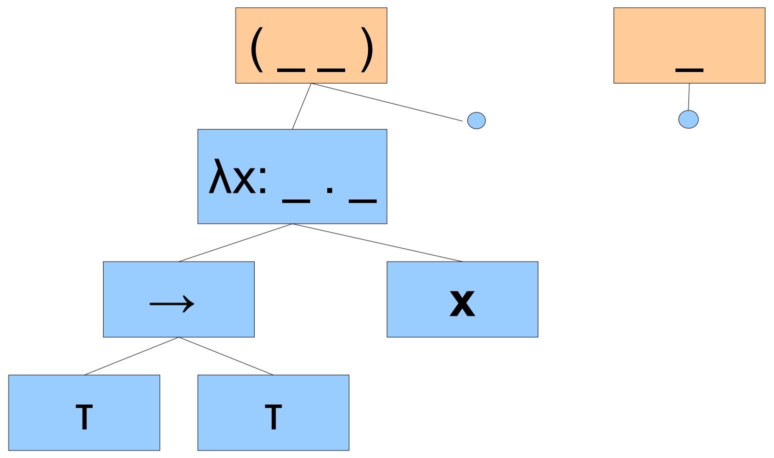
(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

it

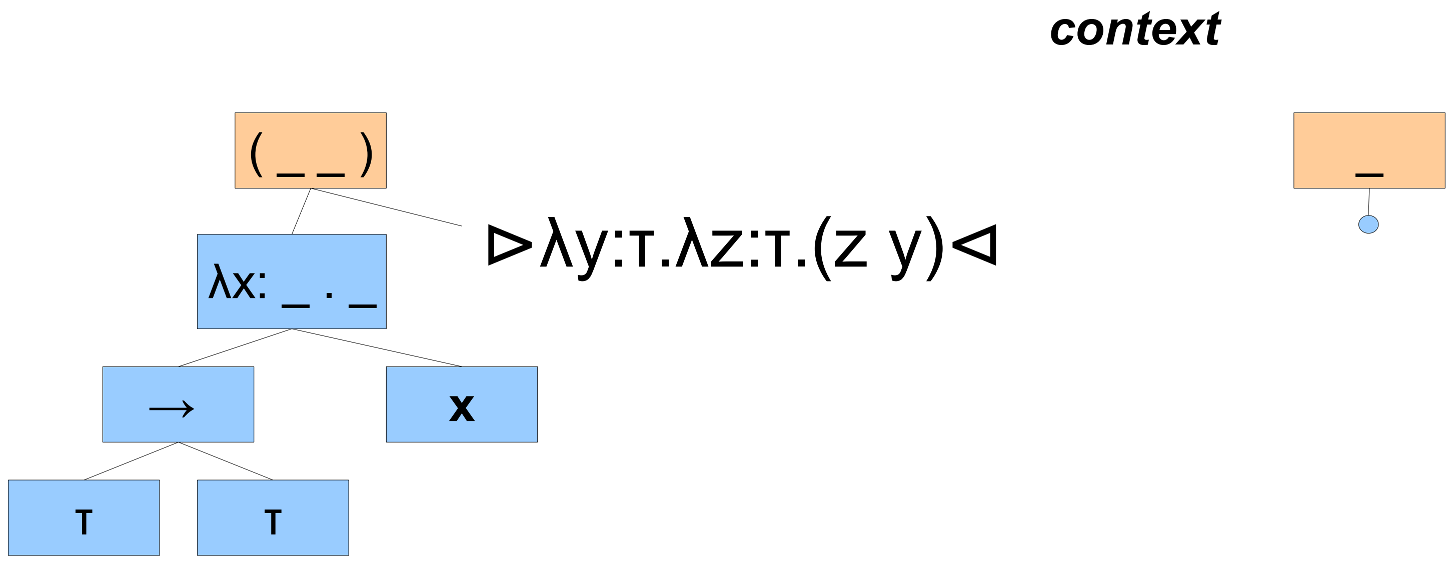
→ ⊳λy:τ.λz:τ.(z y)⊲

context



(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

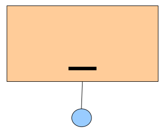


(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

context

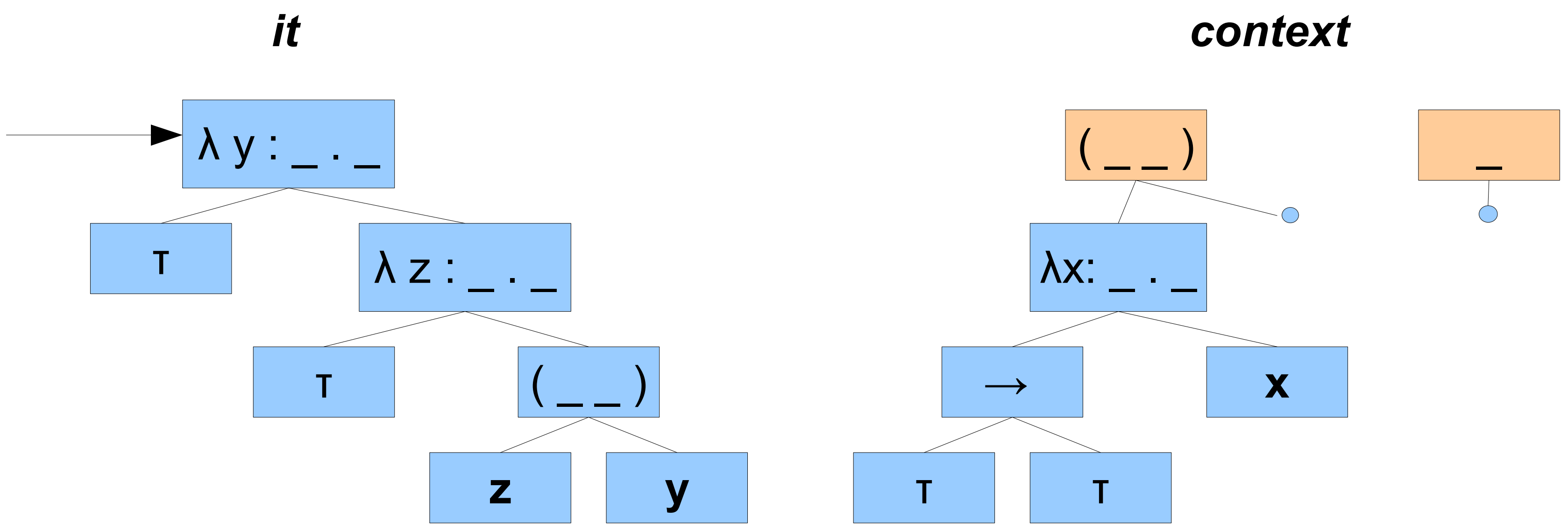
$(\lambda x{:}\tau\rightarrow\tau.x \rhd \lambda y{:}\tau.\lambda z{:}\tau.(z\ y)\lhd)$



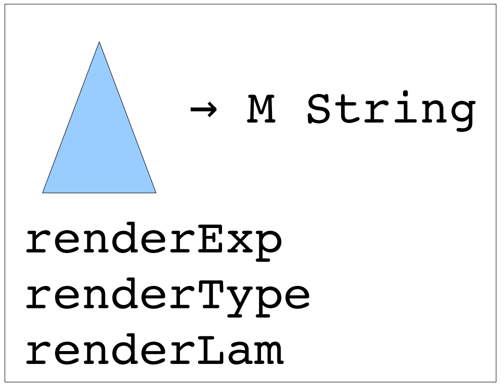
Rendering Problem

(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

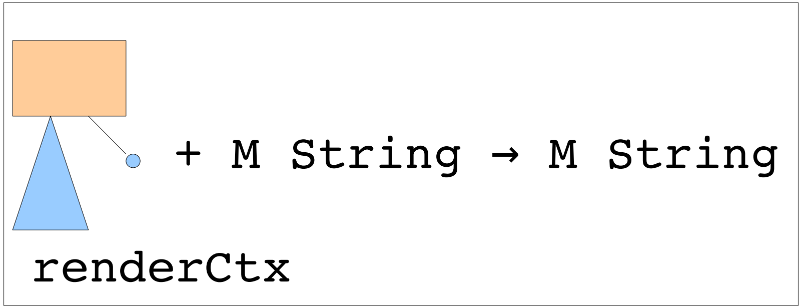


(λx:τ→τ.x ⊳λy:τ.λz:τ.(z y)⊲)



renderExp
renderType
renderLam

M String → M String
renderCursor



renderCtx

Rendering...

```
renderExp :: Exp → M String
renderExp (Abs str ty exp) = do
   tys ← renderType typ
   rhs ← addBinding str (renderExp exp)
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
...
```

```
renderCtx :: Context Lam from to → M String → M String
renderCtx (TypeToAbs str exp) rec = do
   tys ← rec
   rhs ← addBinding str (renderExp exp)
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
renderCtx (ExpToAbs str ty) rec = do
   tys ← renderType ty
   rhs ← addBinding str rec
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
...
```

Rendering...

```
renderExp :: Exp → M String
renderExp (Abs str ty exp) = do
   tys ← renderType typ
   rhs ← addBinding str (renderExp exp)
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
...
```

```
renderCtx :: Context Lam from to → M String → M String
renderCtx (TypeToAbs str exp) rec = do
   tys ← rec
   rhs ← addBinding str (renderExp exp)
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
renderCtx (ExpToAbs str ty) rec = do
   tys ← renderType ty
   rhs ← addBinding str rec
   return (“λ “ ++ str ++ “: “ ++ tys ++ “ . “ ++ rhs)
...
```

Rendering...

```
renderExp :: Exp → M String
renderExp (Abs str ty exp) = do
   tys ← renderType typ
   rhs ← addBinding str (renderExp exp)
   return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
```

```
renderCtx :: Context Lam from to → M String → M String
renderCtx (TypeToAbs str exp) rec = do
   tys ← rec
   rhs ← addBinding str (renderExp exp)
   return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
renderCtx (ExpToAbs str ty) rec = do
   tys ← renderType ty
   rhs ← addBinding str rec
   return ("λ " ++ str ++ ": " ++ tys ++ " . " ++ rhs)
...
```

2 Duplication Problems

- Calculating Traversal Results
- Expressing Binding Transforms

CLASE

```
data Lam ...
data Exp ...
data Type ...
```

```
data ContextI from to where
  TypeToAbs :: String → Exp →
               ContextI Type Exp
  ExpToAbs  :: String → Type →
               ContextI Exp Exp
  ...


instance Language Lam where
  data Context Lam from to
    = CW (ContextI from to)
  ...
```

```
class Language l where
  data Context l :: * → * → *
  ...
```

```
data Cursor l x a
 = (...) ⇒ Cursor {
  it :: a,
  ctx :: Path l (Context l) a l,
  ...
}
```

```
genericMoveUp :: (Language l) ⇒
 Cursor l x a →
 Maybe (CursorWithMovement l Up x a)

genericMoveLeft  :: (Language l) ⇒
  Cursor l x a →
  Maybe (ExistsR l (Cursor l x))


...
```

The CLASE Solution

Declare your “language”

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
data Lam = Lam Exp

data Exp
  = Abs String Type Exp
  | App Exp Exp
  | Var Integer

data Type
  = Unit
  | Arr Type Type
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```haskell
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

The CLASE Solution

- Declare your "language"
- **Generate boilerplate**
- Describe lexical binding
- Implement traversals
- Hook into application

```
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```haskell
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
{-# LANGUAGE TemplateHaskell #-}
module Lam.Gen where

import Lam.Lam
import Data.Cursor.CLASE.Gen.Language
import Data.Cursor.CLASE.Gen.Adapters
import Data.Cursor.CLASE.Gen.Persistence

$(languageGen ["Lam","Language"] ''Lam
              [''Lam, ''Exp, ''Type])

$(adapterGen ["Lam", "Adapters"] ''Lam
             [''Lam, ''Exp, ''Type] "Lam.Language")

$(persistenceGen ["Lam", "Persistence"] ''Lam
                 [''Lam, ''Exp, ''Type] "Lam.Language")

main :: IO ()
main = return ()
```

ASE Solution

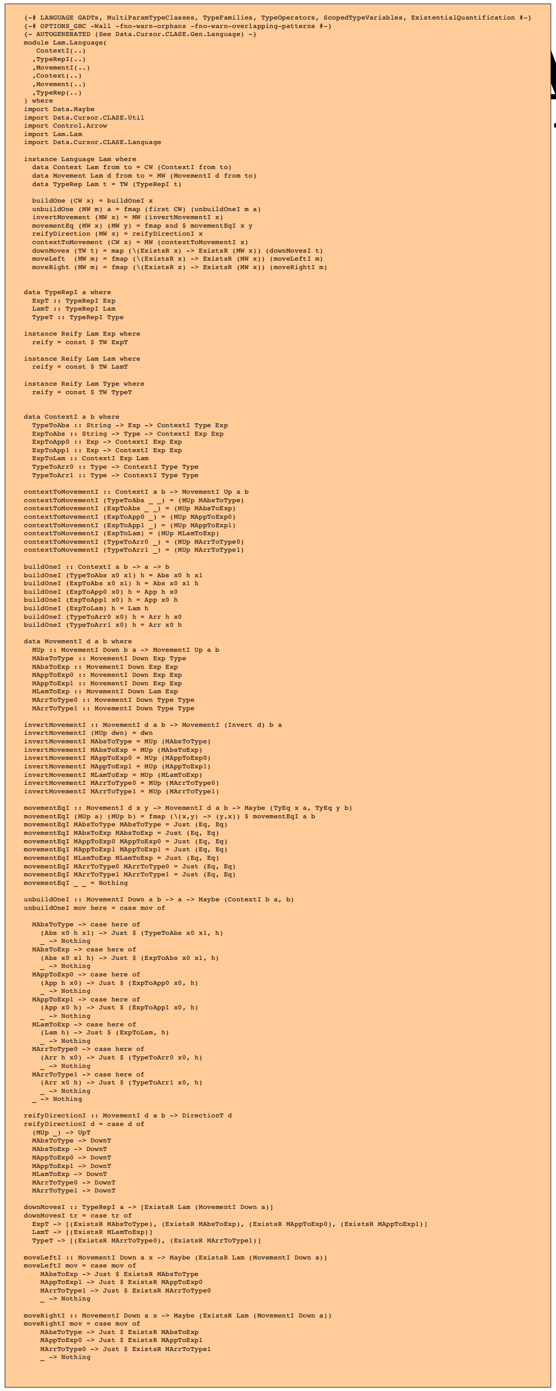
Declare your "language"

Generate boilerplate

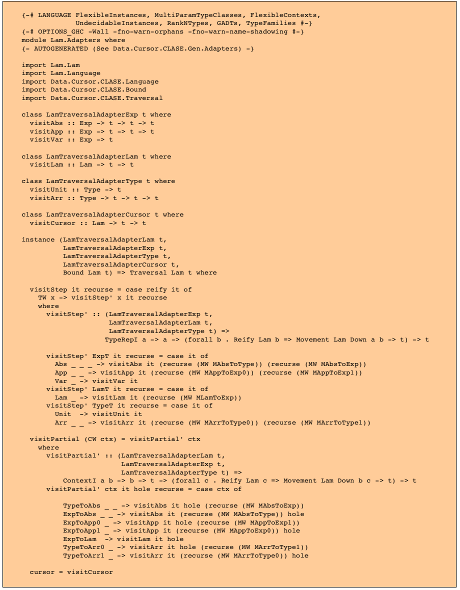
Describe lexical binding

Implement traversals

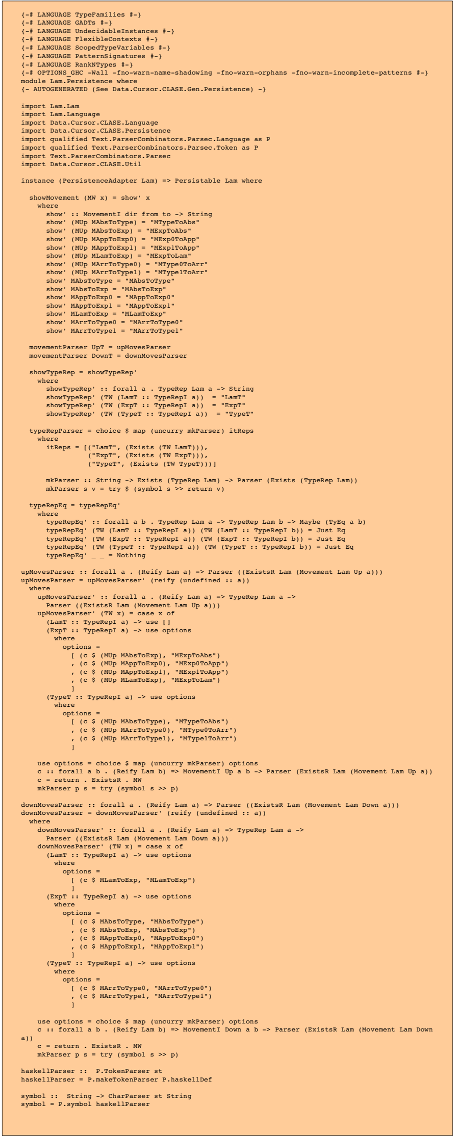
Hook into application



```haskell
{-# LANGUAGE GADTs, MultiParamTypeClasses, TypeFamilies, TypeOperators, ScopedTypeVariables, ExistentialQuantification #-}
{-# OPTIONS_GHC -Wall -fno-warn-orphans -fno-warn-overlapping-patterns #-}
{- AUTOGENERATED (See Data.Cursor.CLASE.Gen.Language) -}
module Lam.Language(
   ContextI(..)
  ,TypeRepI(..)
  ,MovementI(..)
  ,Context(..)
  ,Movement(..)
  ,TypeRep(..)
) where
import Data.Maybe
import Data.Cursor.CLASE.Util
import Control.Arrow
import Lam.Lam
import Data.Cursor.CLASE.Language

instance Language Lam where
  data Context Lam from to = CW (ContextI from to)
  data Movement Lam d from to = MW (MovementI d from to)
  data TypeRep Lam t = TW (TypeRepI t)

  buildOne (CW x) = buildOneI x
  unbuildOne (MW m) a = fmap (first CW) (unbuildOneI m a)
  invertMovement (MW x) = MW (invertMovementI x)
  movementEq (MW x) (MW y) = fmap snd $ movementEqI x y
  reifyDirection (MW x) = reifyDirectionI x
  contextToMovement (CW x) = MW (contextToMovementI x)
  downMoves (TW t) = map (\(ExistsR x) -> ExistsR (MW x)) (downMovesI t)
  moveLeft  (MW m) = fmap (\(ExistsR x) -> ExistsR (MW x)) (moveLeftI m)
  moveRight (MW m) = fmap (\(ExistsR x) -> ExistsR (MW x)) (moveRightI m)


data TypeRepI a where
  ExpT :: TypeRepI Exp
  LamT :: TypeRepI Lam
  TypeT :: TypeRepI Type

instance Reify Lam Exp where
  reify = const $ TW ExpT

instance Reify Lam Lam where
  reify = const $ TW LamT

instance Reify Lam Type where
  reify = const $ TW TypeT


data ContextI a b where
  TypeToAbs :: String -> Exp -> ContextI Type Exp
  ExpToAbs :: String -> Type -> ContextI Exp Exp
  ExpToApp0 :: Exp -> ContextI Exp Exp
  ExpToApp1 :: Exp -> ContextI Exp Exp
  ExpToLam :: ContextI Exp Lam
  TypeToArr0 :: Type -> ContextI Type Type
  TypeToArr1 :: Type -> ContextI Type Type

contextToMovementI :: ContextI a b -> MovementI Up a b
contextToMovementI (TypeToAbs _ _) = (MUp MAbsToType)
contextToMovementI (ExpToAbs _ _) = (MUp MAbsToExp)
contextToMovementI (ExpToApp0 _) = (MUp MAppToExp0)
contextToMovementI (ExpToApp1 _) = (MUp MAppToExp1)
contextToMovementI (ExpToLam) = (MUp MLamToExp)
contextToMovementI (TypeToArr0 _) = (MUp MArrToType0)
contextToMovementI (TypeToArr1 _) = (MUp MArrToType1)

buildOneI :: ContextI a b -> a -> b
buildOneI (TypeToAbs x0 x1) h = Abs x0 h x1
buildOneI (ExpToAbs x0 x1) h = Abs x0 x1 h
buildOneI (ExpToApp0 x0) h = App h x0
buildOneI (ExpToApp1 x0) h = App x0 h
buildOneI (ExpToLam) h = Lam h
buildOneI (TypeToArr0 x0) h = Arr h x0
buildOneI (TypeToArr1 x0) h = Arr x0 h

data MovementI d a b where
  MUp :: MovementI Down b a -> MovementI Up a b
  MAbsToType :: MovementI Down Exp Type
  MAbsToExp :: MovementI Down Exp Exp
  MAppToExp0 :: MovementI Down Exp Exp
  MAppToExp1 :: MovementI Down Exp Exp
  MLamToExp :: MovementI Down Lam Exp
  MArrToType0 :: MovementI Down Type Type
  MArrToType1 :: MovementI Down Type Type

invertMovementI :: MovementI d a b -> MovementI (Invert d) b a
invertMovementI (MUp dwn) = dwn
invertMovementI MAbsToType = MUp (MAbsToType)
invertMovementI MAbsToExp = MUp (MAbsToExp)
invertMovementI MAppToExp0 = MUp (MAppToExp0)
invertMovementI MAppToExp1 = MUp (MAppToExp1)
invertMovementI MLamToExp = MUp (MLamToExp)
invertMovementI MArrToType0 = MUp (MArrToType0)
invertMovementI MArrToType1 = MUp (MArrToType1)

movementEqI :: MovementI d x y -> MovementI d a b -> Maybe (TyEq x a, TyEq y b)
movementEqI (MUp a) (MUp b) = fmap (\(x,y) -> (y,x)) $ movementEqI a b
movementEqI MAbsToType MAbsToType = Just (Eq, Eq)
movementEqI MAbsToExp MAbsToExp = Just (Eq, Eq)
movementEqI MAppToExp0 MAppToExp0 = Just (Eq, Eq)
movementEqI MAppToExp1 MAppToExp1 = Just (Eq, Eq)
movementEqI MLamToExp MLamToExp = Just (Eq, Eq)
movementEqI MArrToType0 MArrToType0 = Just (Eq, Eq)
movementEqI MArrToType1 MArrToType1 = Just (Eq, Eq)
movementEqI _ _ = Nothing

unbuildOneI :: MovementI Down a b -> a -> Maybe (ContextI b a, b)
unbuildOneI mov here = case mov of

  MAbsToType -> case here of
    (Abs x0 h x1) -> Just $ (TypeToAbs x0 x1, h)
    _ -> Nothing
  MAbsToExp -> case here of
    (Abs x0 x1 h) -> Just $ (ExpToAbs x0 x1, h)
    _ -> Nothing
  MAppToExp0 -> case here of
    (App h x0) -> Just $ (ExpToApp0 x0, h)
    _ -> Nothing
  MAppToExp1 -> case here of
    (App x0 h) -> Just $ (ExpToApp1 x0, h)
    _ -> Nothing
  MLamToExp -> case here of
    (Lam h) -> Just $ (ExpToLam, h)
    _ -> Nothing
  MArrToType0 -> case here of
    (Arr h x0) -> Just $ (TypeToArr0 x0, h)
    _ -> Nothing
  MArrToType1 -> case here of
    (Arr x0 h) -> Just $ (TypeToArr1 x0, h)
    _ -> Nothing
  _ -> Nothing

reifyDirectionI :: MovementI d a b -> DirectionT d
reifyDirectionI d = case d of
  (MUp _) -> UpT
  MAbsToType -> DownT
  MAbsToExp -> DownT
  MAppToExp0 -> DownT
  MAppToExp1 -> DownT
  MLamToExp -> DownT
  MArrToType0 -> DownT
  MArrToType1 -> DownT

downMovesI :: TypeRepI a -> [ExistsR Lam (MovementI Down a)]
downMovesI tr = case tr of
  ExpT -> [(ExistsR MAbsToType), (ExistsR MAbsToExp), (ExistsR MAppToExp0), (ExistsR MAppToExp1)]
  LamT -> [(ExistsR MLamToExp)]
  TypeT -> [(ExistsR MArrToType0), (ExistsR MArrToType1)]

moveLeftI :: MovementI Down a x -> Maybe (ExistsR Lam (MovementI Down a))
moveLeftI mov = case mov of
    MAbsToExp -> Just $ ExistsR MAbsToType
    MAppToExp1 -> Just $ ExistsR MAppToExp0
    MArrToType1 -> Just $ ExistsR MArrToType0
    _ -> Nothing

moveRightI :: MovementI Down a x -> Maybe (ExistsR Lam (MovementI Down a))
moveRightI mov = case mov of
    MAbsToType -> Just $ ExistsR MAbsToExp
    MAppToExp0 -> Just $ ExistsR MAppToExp1
    MArrToType0 -> Just $ ExistsR MArrToType1
    _ -> Nothing
```



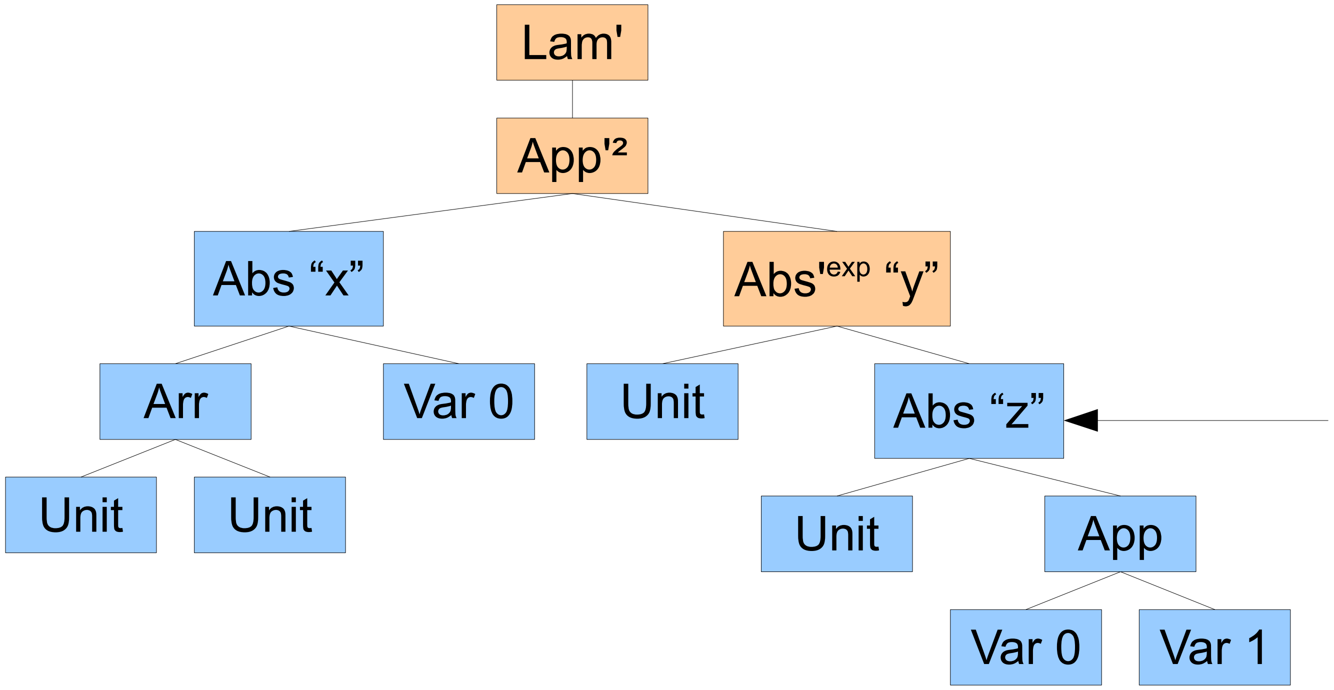
```haskell
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, FlexibleContexts,
             UndecidableInstances, RankNTypes, GADTs, TypeFamilies #-}
{-# OPTIONS_GHC -Wall -fno-warn-orphans -fno-warn-name-shadowing #-}
module Lam.Adapters where
{- AUTOGENERATED (See Data.Cursor.CLASE.Gen.Adapters) -}

import Lam.Lam
import Lam.Language
import Data.Cursor.CLASE.Language
import Data.Cursor.CLASE.Bound
import Data.Cursor.CLASE.Traversal

class LamTraversalAdapterExp t where
  visitAbs :: Exp -> t -> t -> t
  visitApp :: Exp -> t -> t -> t
  visitVar :: Exp -> t

class LamTraversalAdapterLam t where
  visitLam :: Lam -> t -> t

class LamTraversalAdapterType t where
  visitUnit :: Type -> t
  visitArr :: Type -> t -> t -> t

class LamTraversalAdapterCursor t where
  visitCursor :: Lam -> t -> t

instance (LamTraversalAdapterLam t,
          LamTraversalAdapterExp t,
          LamTraversalAdapterType t,
          LamTraversalAdapterCursor t,
          Bound Lam t) => Traversal Lam t where

  visitStep it recurse = case reify it of
    TW x -> visitStep' x it recurse
    where
      visitStep' :: (LamTraversalAdapterExp t,
                     LamTraversalAdapterLam t,
                     LamTraversalAdapterType t) =>
                    TypeRepI a -> a -> (forall b . Reify Lam b => Movement Lam Down a b -> t) -> t

      visitStep' ExpT it recurse = case it of
        Abs _ _ _ -> visitAbs it (recurse (MW MAbsToType)) (recurse (MW MAbsToExp))
        App _ _ -> visitApp it (recurse (MW MAppToExp0)) (recurse (MW MAppToExp1))
        Var _ -> visitVar it
      visitStep' LamT it recurse = case it of
        Lam _ -> visitLam it (recurse (MW MLamToExp))
      visitStep' TypeT it recurse = case it of
        Unit  -> visitUnit it
        Arr _ _ -> visitArr it (recurse (MW MArrToType0)) (recurse (MW MArrToType1))

  visitPartial (CW ctx) = visitPartial' ctx
    where
      visitPartial' :: (LamTraversalAdapterLam t,
                        LamTraversalAdapterExp t,
                        LamTraversalAdapterType t) =>
          ContextI a b -> b -> t -> (forall c . Reify Lam c => Movement Lam Down b c -> t) -> t
      visitPartial' ctx it hole recurse = case ctx of

          TypeToAbs _ _ -> visitAbs it hole (recurse (MW MAbsToExp))
          ExpToAbs _ _ -> visitAbs it (recurse (MW MAbsToType)) hole
          ExpToApp0 _ -> visitApp it hole (recurse (MW MAppToExp1))
          ExpToApp1 _ -> visitApp it (recurse (MW MAppToExp0)) hole
          ExpToLam  -> visitLam it hole
          TypeToArr0 _ -> visitArr it hole (recurse (MW MArrToType1))
          TypeToArr1 _ -> visitArr it (recurse (MW MArrToType0)) hole

  cursor = visitCursor
```



```haskell
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE PatternSignatures #-}
{-# LANGUAGE RankNTypes #-}
{-# OPTIONS_GHC -Wall -fno-warn-name-shadowing -fno-warn-orphans -fno-warn-incomplete-patterns #-}
module Lam.Persistence where
{- AUTOGENERATED (See Data.Cursor.CLASE.Gen.Persistence) -}

import Lam.Lam
import Lam.Language
import Data.Cursor.CLASE.Language
import Data.Cursor.CLASE.Persistence
import qualified Text.ParserCombinators.Parsec.Language as P
import qualified Text.ParserCombinators.Parsec.Token as P
import Text.ParserCombinators.Parsec
import Data.Cursor.CLASE.Util

instance (PersistenceAdapter Lam) => Persistable Lam where

  showMovement (MW x) = show' x
    where
      show' :: MovementI dir from to -> String
      show' (MUp MAbsToType) = "MTypeToAbs"
      show' (MUp MAbsToExp) = "MExpToAbs"
      show' (MUp MAppToExp0) = "MExp0ToApp"
      show' (MUp MAppToExp1) = "MExp1ToApp"
      show' (MUp MLamToExp) = "MExpToLam"
      show' (MUp MArrToType0) = "MType0ToArr"
      show' (MUp MArrToType1) = "MType1ToArr"
      show' MAbsToType = "MAbsToType"
      show' MAbsToExp = "MAbsToExp"
      show' MAppToExp0 = "MAppToExp0"
      show' MAppToExp1 = "MAppToExp1"
      show' MLamToExp = "MLamToExp"
      show' MArrToType0 = "MArrToType0"
      show' MArrToType1 = "MArrToType1"

  movementParser UpT = upMovesParser
  movementParser DownT = downMovesParser

  showTypeRep = showTypeRep'
    where
      showTypeRep' :: forall a . TypeRep Lam a -> String
      showTypeRep' (TW (LamT :: TypeRepI a))  = "LamT"
      showTypeRep' (TW (ExpT :: TypeRepI a))  = "ExpT"
      showTypeRep' (TW (TypeT :: TypeRepI a))  = "TypeT"

  typeRepParser = choice $ map (uncurry mkParser) itReps
    where
      itReps = [("LamT", (Exists (TW LamT))),
                ("ExpT", (Exists (TW ExpT))),
                ("TypeT", (Exists (TW TypeT)))]

      mkParser :: String -> Exists (TypeRep Lam) -> Parser (Exists (TypeRep Lam))
      mkParser s v = try $ (symbol s >> return v)

  typeRepEq = typeRepEq'
    where
      typeRepEq' :: forall a b . TypeRep Lam a -> TypeRep Lam b -> Maybe (TyEq a b)
      typeRepEq' (TW (LamT :: TypeRepI a)) (TW (LamT :: TypeRepI b)) = Just Eq
      typeRepEq' (TW (ExpT :: TypeRepI a)) (TW (ExpT :: TypeRepI b)) = Just Eq
      typeRepEq' (TW (TypeT :: TypeRepI a)) (TW (TypeT :: TypeRepI b)) = Just Eq
      typeRepEq' _ _ = Nothing

upMovesParser :: forall a . (Reify Lam a) => Parser ((ExistsR Lam (Movement Lam Up a)))
upMovesParser = upMovesParser' (reify (undefined :: a))
  where
    upMovesParser' :: forall a . (Reify Lam a) => TypeRep Lam a ->
      Parser ((ExistsR Lam (Movement Lam Up a)))
    upMovesParser' (TW x) = case x of
      (LamT :: TypeRepI a) -> use []
      (ExpT :: TypeRepI a) -> use options
        where
          options =
            [ (c $ (MUp MAbsToExp), "MExpToAbs")
            , (c $ (MUp MAppToExp0), "MExp0ToApp")
            , (c $ (MUp MAppToExp1), "MExp1ToApp")
            , (c $ (MUp MLamToExp), "MExpToLam")
            ]
      (TypeT :: TypeRepI a) -> use options
        where
          options =
            [ (c $ (MUp MAbsToType), "MTypeToAbs")
            , (c $ (MUp MArrToType0), "MType0ToArr")
            , (c $ (MUp MArrToType1), "MType1ToArr")
            ]

    use options = choice $ map (uncurry mkParser) options
    c :: forall a b . (Reify Lam b) => MovementI Up a b -> Parser (ExistsR Lam (Movement Lam Up a))
    c = return . ExistsR . MW
    mkParser p s = try (symbol s >> p)

downMovesParser :: forall a . (Reify Lam a) => Parser ((ExistsR Lam (Movement Lam Down a)))
downMovesParser = downMovesParser' (reify (undefined :: a))
  where
    downMovesParser' :: forall a . (Reify Lam a) => TypeRep Lam a ->
      Parser ((ExistsR Lam (Movement Lam Down a)))
    downMovesParser' (TW x) = case x of
      (LamT :: TypeRepI a) -> use options
        where
          options =
            [ (c $ MLamToExp, "MLamToExp")
            ]
      (ExpT :: TypeRepI a) -> use options
        where
          options =
            [ (c $ MAbsToType, "MAbsToType")
            , (c $ MAbsToExp, "MAbsToExp")
            , (c $ MAppToExp0, "MAppToExp0")
            , (c $ MAppToExp1, "MAppToExp1")
            ]
      (TypeT :: TypeRepI a) -> use options
        where
          options =
            [ (c $ MArrToType0, "MArrToType0")
            , (c $ MArrToType1, "MArrToType1")
            ]

    use options = choice $ map (uncurry mkParser) options
    c :: forall a b . (Reify Lam b) => MovementI Down a b -> Parser (ExistsR Lam (Movement Lam Down
a))
    c = return . ExistsR . MW
    mkParser p s = try (symbol s >> p)

haskellParser ::  P.TokenParser st
haskellParser = P.makeTokenParser P.haskellDef

symbol ::  String -> CharParser st String
symbol = P.symbol haskellParser
```

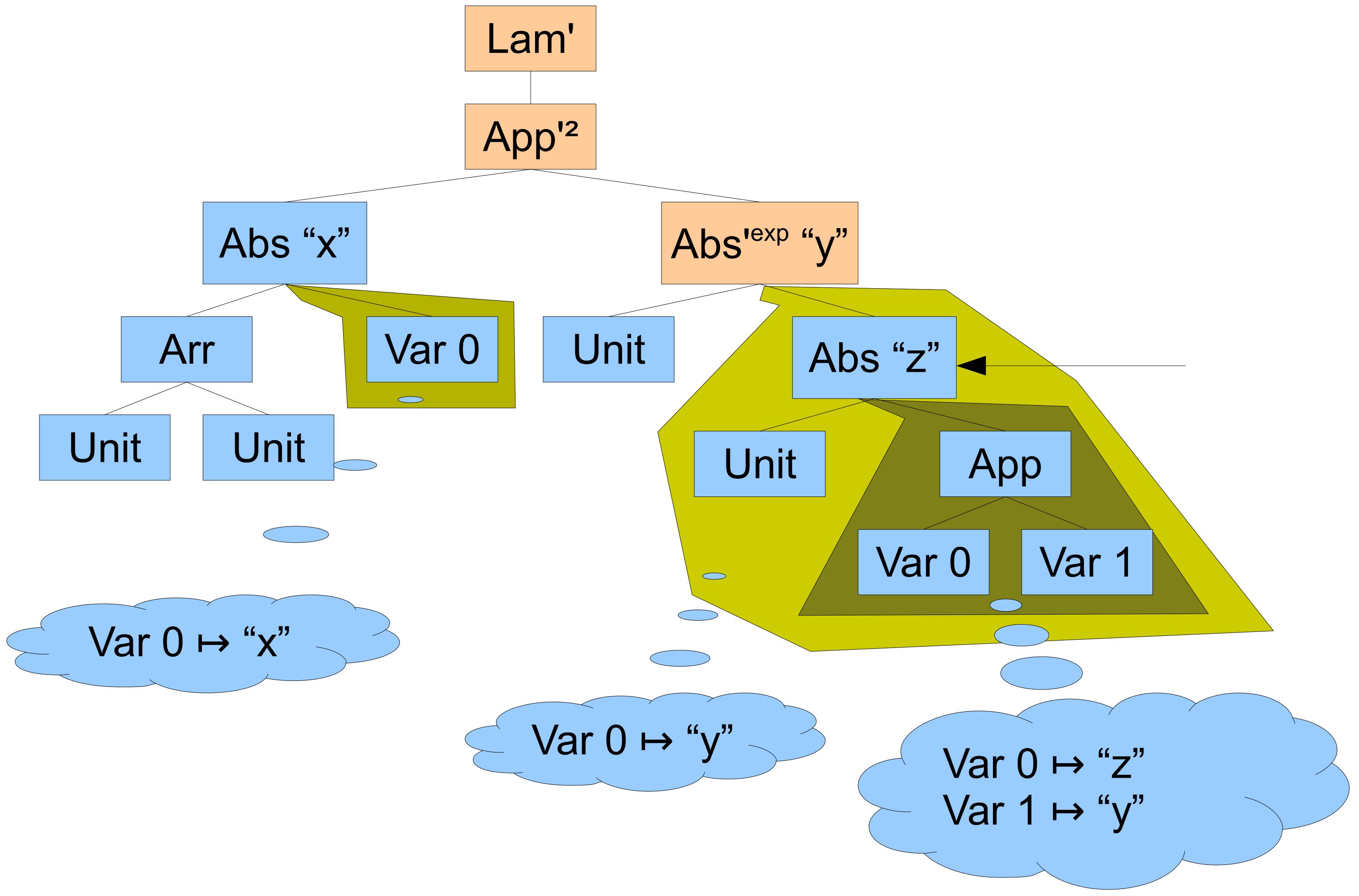
Binding

- Declare your "language"
- Generate boilerplate
- **Describe lexical binding**
- Implement traversals
- Hook into application



Binding

- Declare your "language"
- Generate boilerplate
- **Describe lexical binding**
- Implement traversals
- Hook into application



Binding

Declare your "language"

Generate boilerplate

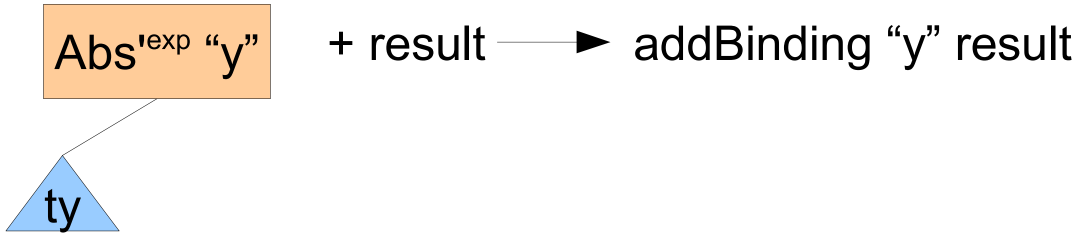
Describe lexical binding

Implement traversals

Hook into application

```
class (Language l) => Bound l t where
  bindingHook :: Context l from to -> t -> t
```

```
instance Bound Lam (M a) where
  bindingHook (ExpToAbs str _) hole
     = addBinding str hole
  bindingHook _ hole = hole
```



The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
completeTraversal
  :: ∀ l t x a . (Traversal l t) ⇒ Cursor l x a → t
```

```
class (Bound l t) ⇒ Traversal l t where

  cursor :: l → t → t

  visitStep :: (Reify l a) ⇒ a →
               (∀ b . Reify l b ⇒
                      Movement l Down a b → t) →
               t

  visitPartial :: Context l a b → b → t →
                  (∀ c . Reify l c ⇒
                         Movement l Down b c → t) →
                  t
```

Traversal Adapters...

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
class LamTraversalAdapterExp t where
  visitAbs :: Exp → t → t → t
  visitApp :: Exp → t → t → t
  visitVar :: Exp → t

class LamTraversalAdapterLam t where
  visitLam :: Lam → t → t

class LamTraversalAdapterType t where
  visitUnit :: Type → t
  visitArr :: Type → t → t → t

class LamTraversalAdapterCursor t where
  visitCursor :: Lam → t → t
```

Traversal Adapters...

```
class LamTraversalAdapterExp t where
  visitAbs :: Exp → t → t → t
  ...
```

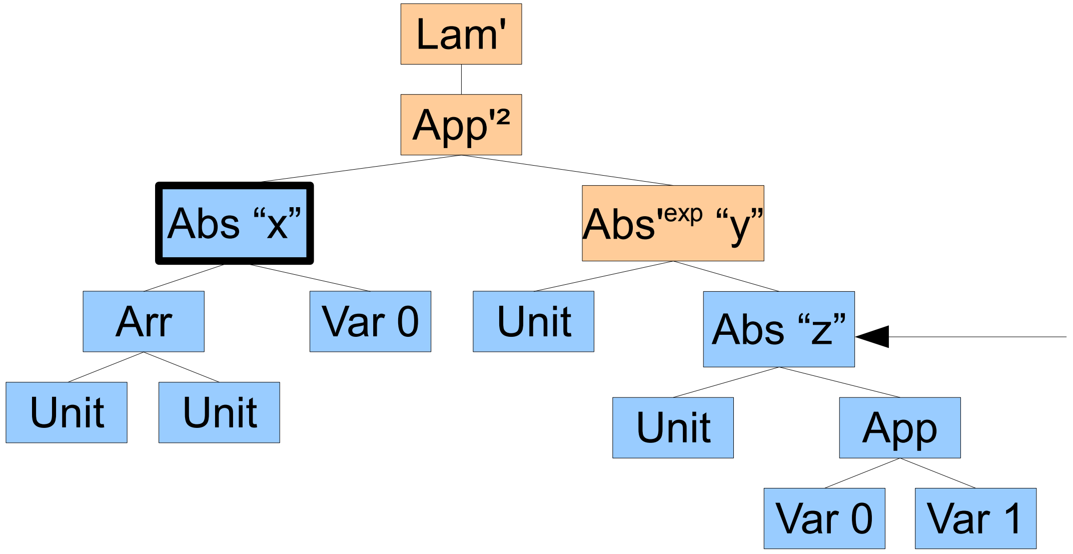
Declare your "language"

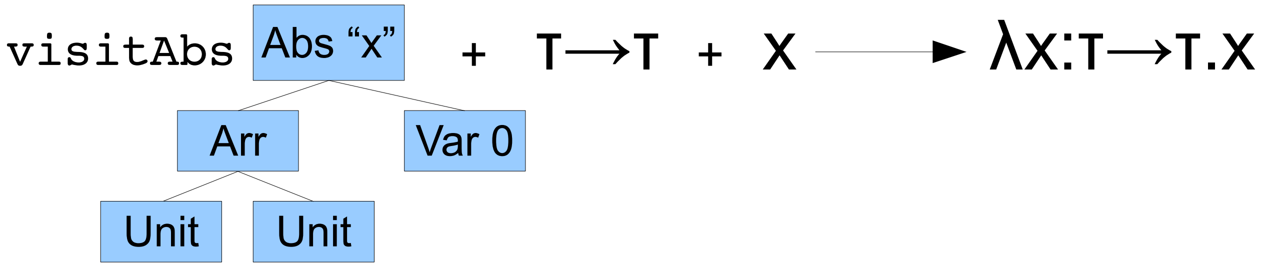
Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application





Traversal Adapters...

```
class LamTraversalAdapterExp t where
  visitAbs :: Exp → t → t → t
  ...
```

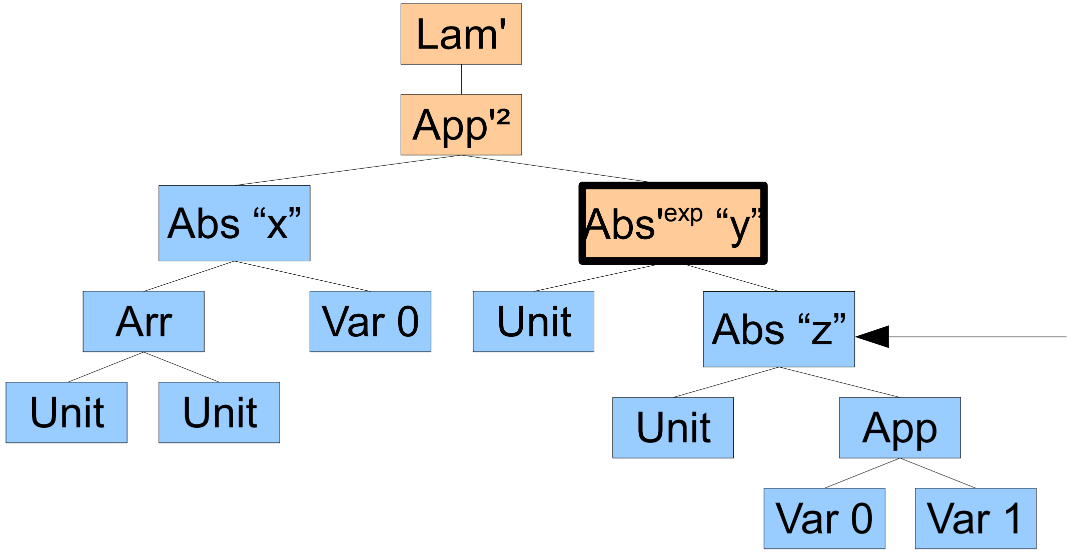
Declare your "language"

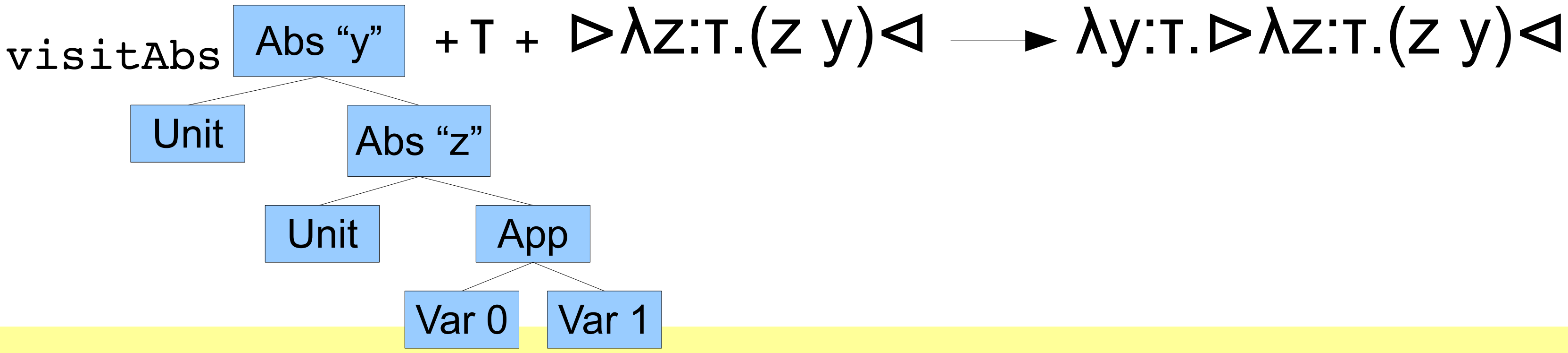
Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application





$+\,\mathrm{T}\,+\ \rhd\lambda z{:}\tau.(z\ y)\lhd \longrightarrow \lambda y{:}\tau.\rhd\lambda z{:}\tau.(z\ y)\lhd$

Rendering...

Declare your "language"

Generate boilerplate

Describe lexical binding

Implement traversals

Hook into application

```
instance LamTraversalAdapterExp (M String) where
  visitAbs (Abs str _ _) ty exp = do
    tys ← ty
    exps ← exp
    return ("λ " ++ str ++ " : "
                 ++ tys ++ " . " ++ exps)

instance LamTraversalAdapterCursor (M String) where
  visitCursor _ ins = do
    str ← ins
    return ("▷" ++ str ++ "◁")
```

The CLASE Solution

Declare your "language"

```
completeTraversal
  :: ∀ l t x a . (Traversal l t) ⇒ Cursor l x a → t
```

Generate boilerplate

Describe lexical binding

```
instance (LamTraversalAdapterLam t,
          LamTraversalAdapterExp t,
          LamTraversalAdapterType t,
          LamTraversalAdapterCursor t,
          Bound Lam t) ⇒ Traversal Lam t where
```

Implement traversals

```
instance LamTraversalAdapterExp (M String) where ...
instance LamTraversalAdapterLam (M String) where ...
instance LamTraversalAdapterType (M String) where ..
instance Bound Lam (M a) where ...
```

Hook into application

```
render :: Cursor Lam x a -> String
render = runM . completeTraversal
```

The CLASE Solution

Declare your "language"

Generate boilerplate

Describe lexical binding

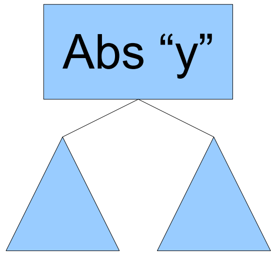
Implement traversals

Hook into application

Quick Demo

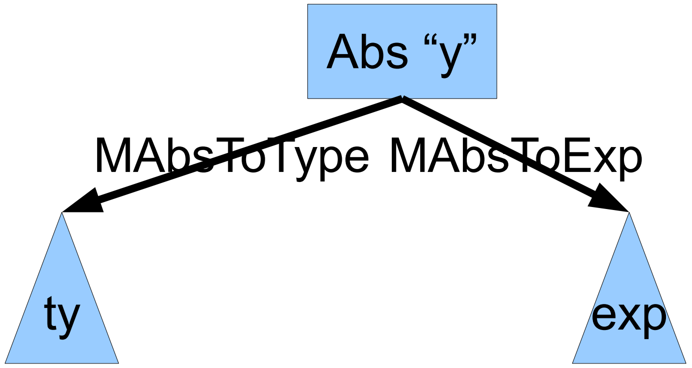
Moving around

```
data Exp
  = Abs String Type Exp
   ...
```



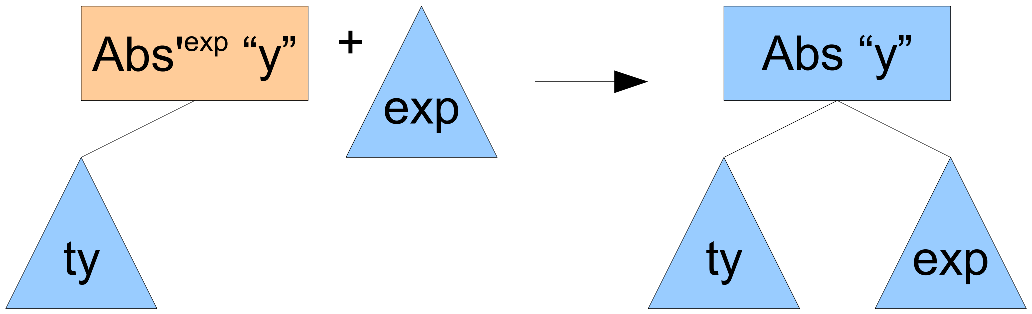
```
data Up
data Down
```

```
data MovementI direction from to where
  MAbsToType :: MovementI Down Exp Type
  MAbsToExp  :: MovementI Down Exp Exp
   ...
  MUp :: MovementI Down to from → MovementI Up from to
```



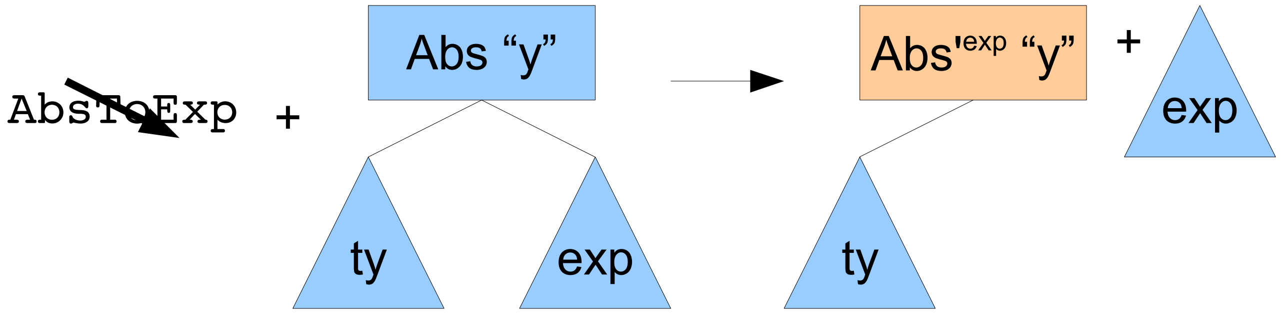
Moving Up

```
buildOneI :: ContextI a b -> a -> b
buildOneI (TypeToAbs x0 x1) h = Abs x0 h x1
buildOneI (ExpToAbs x0 x1) h = Abs x0 x1 h
...
```

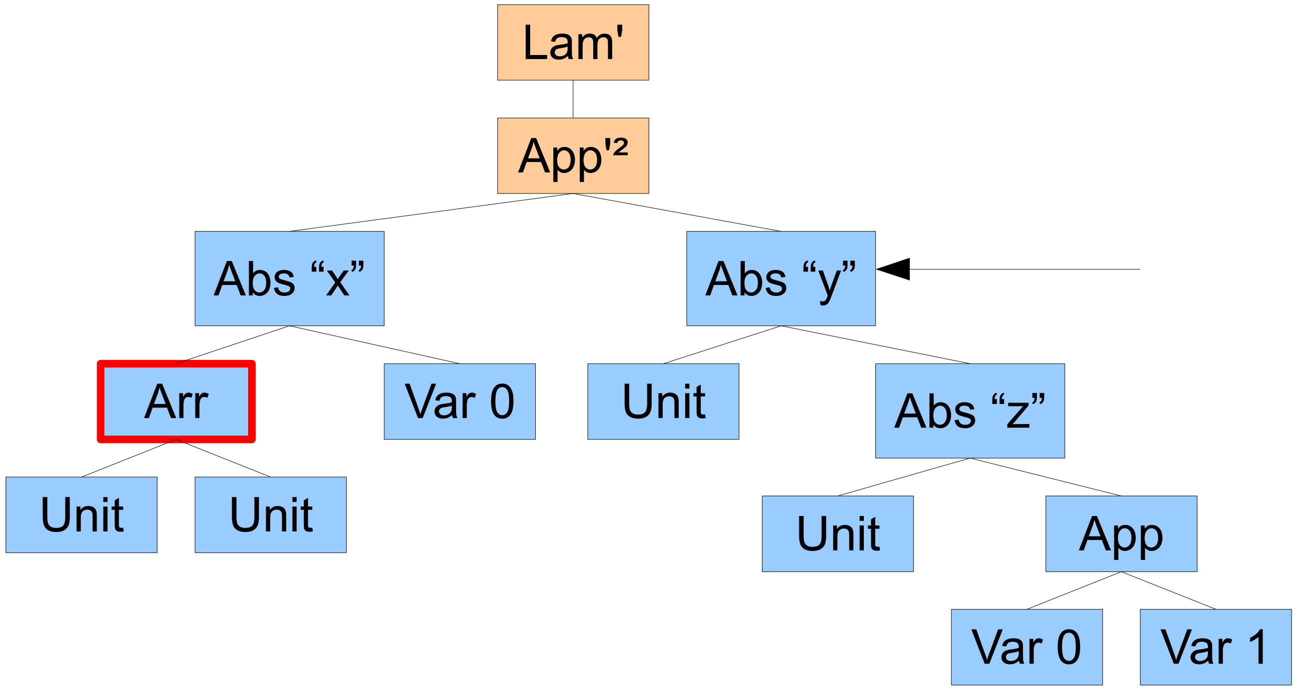


Moving Down

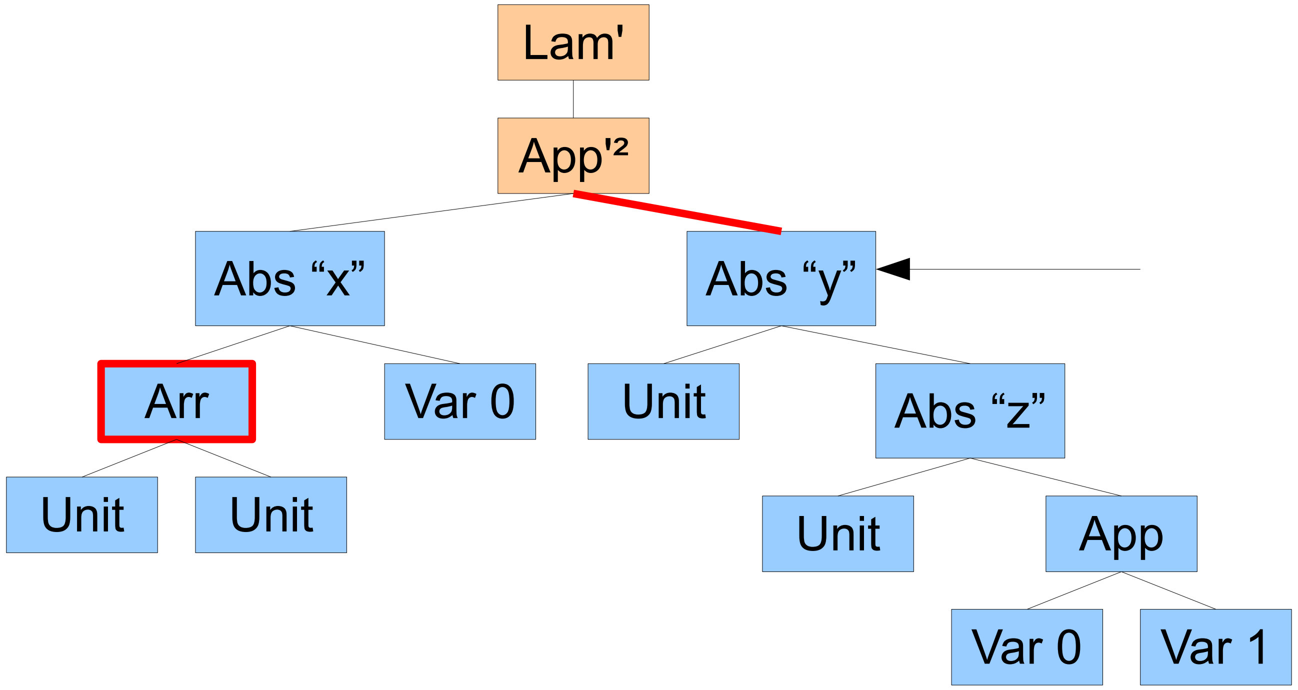
```
unbuildOneI :: MovementI Down a b → a →
                  Maybe (ContextI b a, b)

unbuildOneI mov here = case mov of
  MAbsToType → case here of
    (Abs x0 h x1) → Just (TypeToAbs x0 x1, h)
    _ → Nothing
  MAbsToExp → case here of
    (Abs x0 x1 h) → Just (ExpToAbs x0 x1, h)
    _ → Nothing
   ...
```



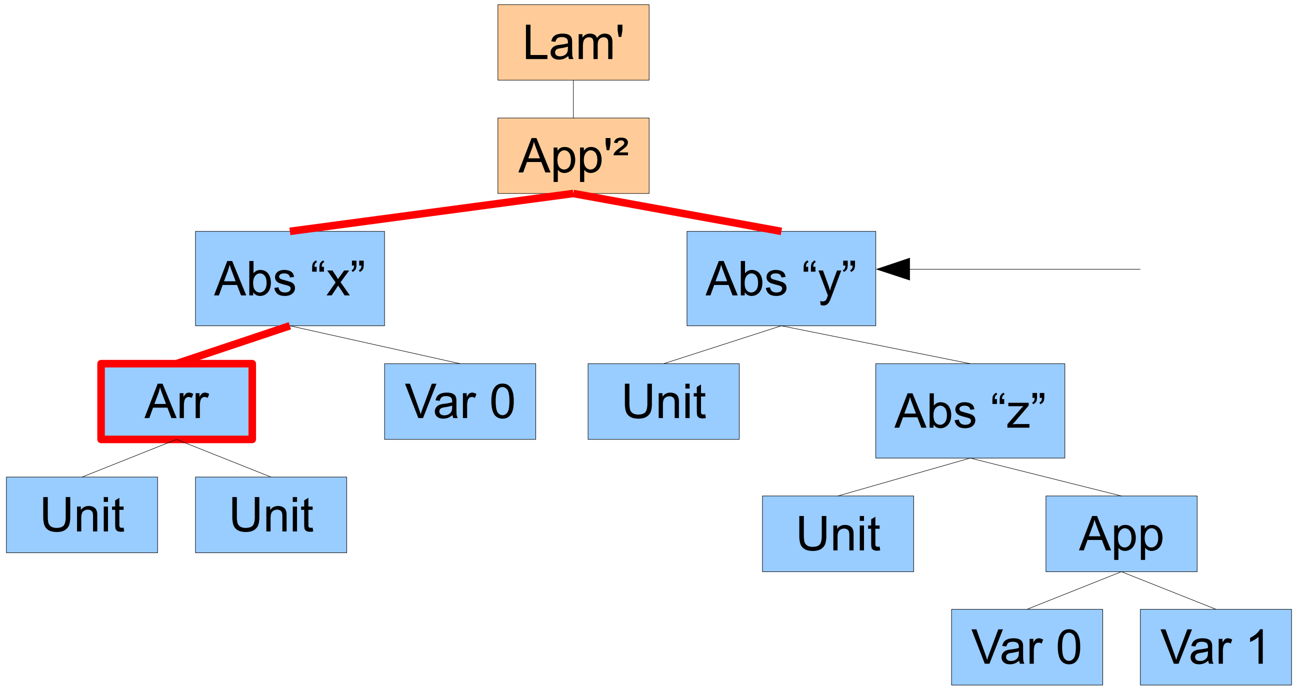
Bookmarks


Lam'
App'²
Abs “x”
Abs “y”
Arr
Var 0
Unit
Abs “z”
Unit
Unit
Unit
App
Var 0
Var 1

Bookmarks

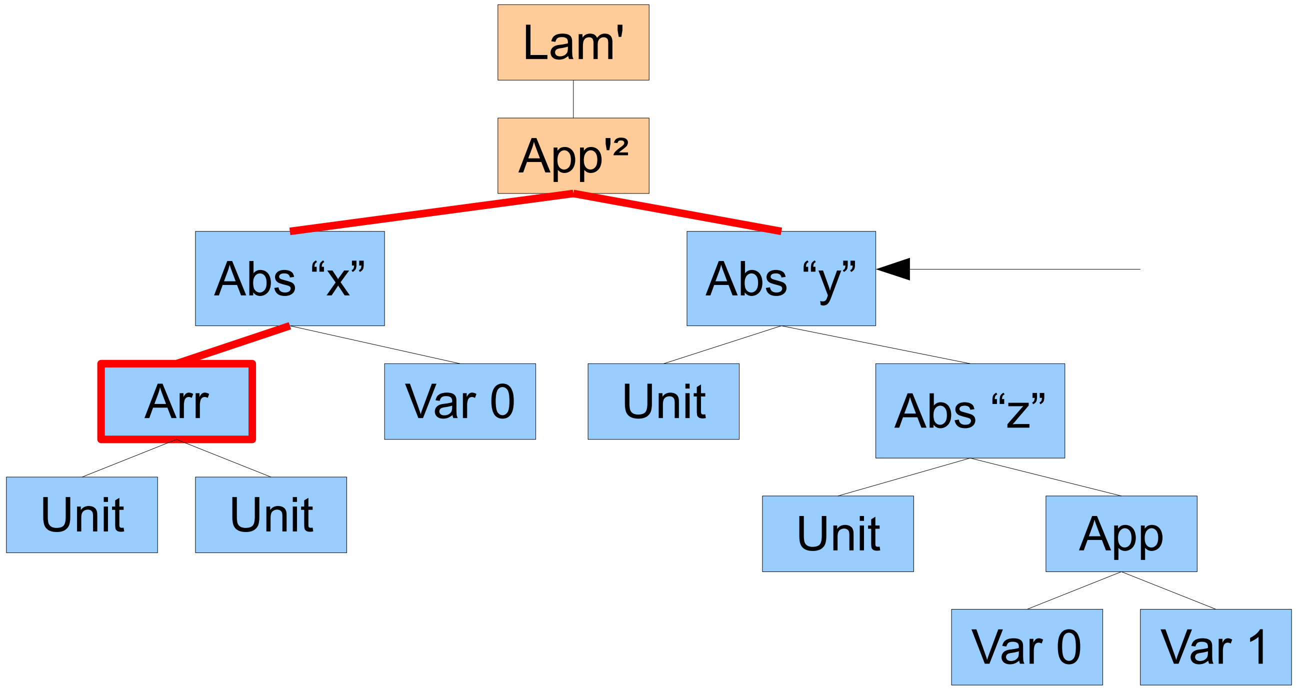

Lam'
App'²
Abs "x"
Abs "y"
Arr
Var 0
Unit
Abs "z"
Unit
Unit
Unit
App
Var 0
Var 1

Bookmarks


Lam'
App'²
Abs "x"
Abs "y"
Arr
Var 0
Unit
Abs "z"
Unit
Unit
Unit
App
Var 0
Var 1

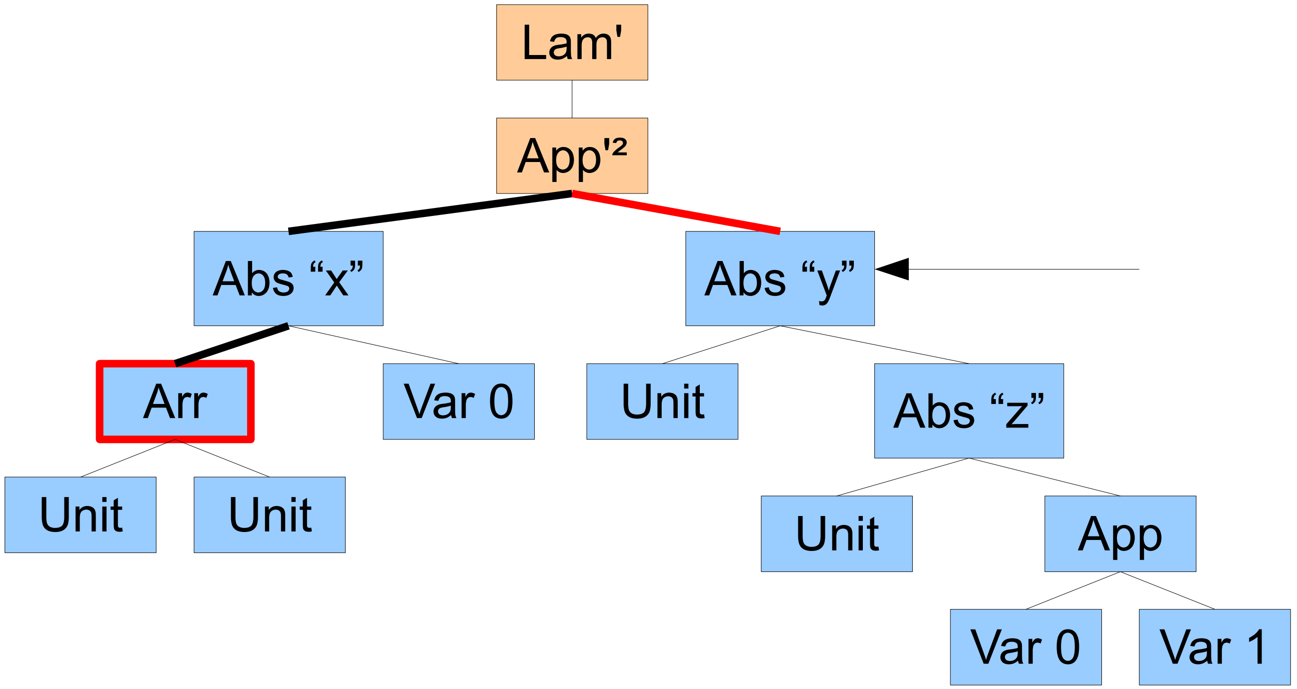
Bookmarks

```
data Route l from to where
  Route :: (...) =>
               Path l (Movement l Up) from mid →
               Path l (Movement l Down) mid to →
               Route l from to
```



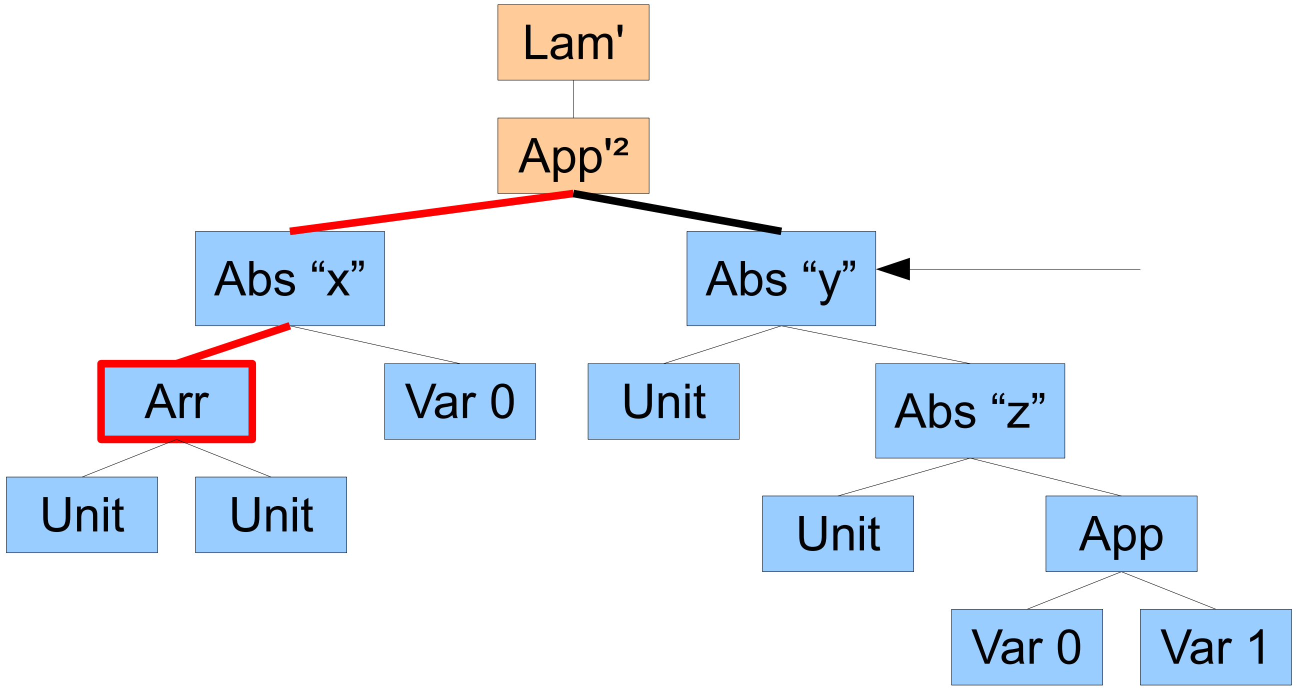
Bookmarks

```
data Route l from to where
  Route :: (...) =>
              Path l (Movement l Up) from mid →
              Path l (Movement l Down) mid to →
              Route l from to
```



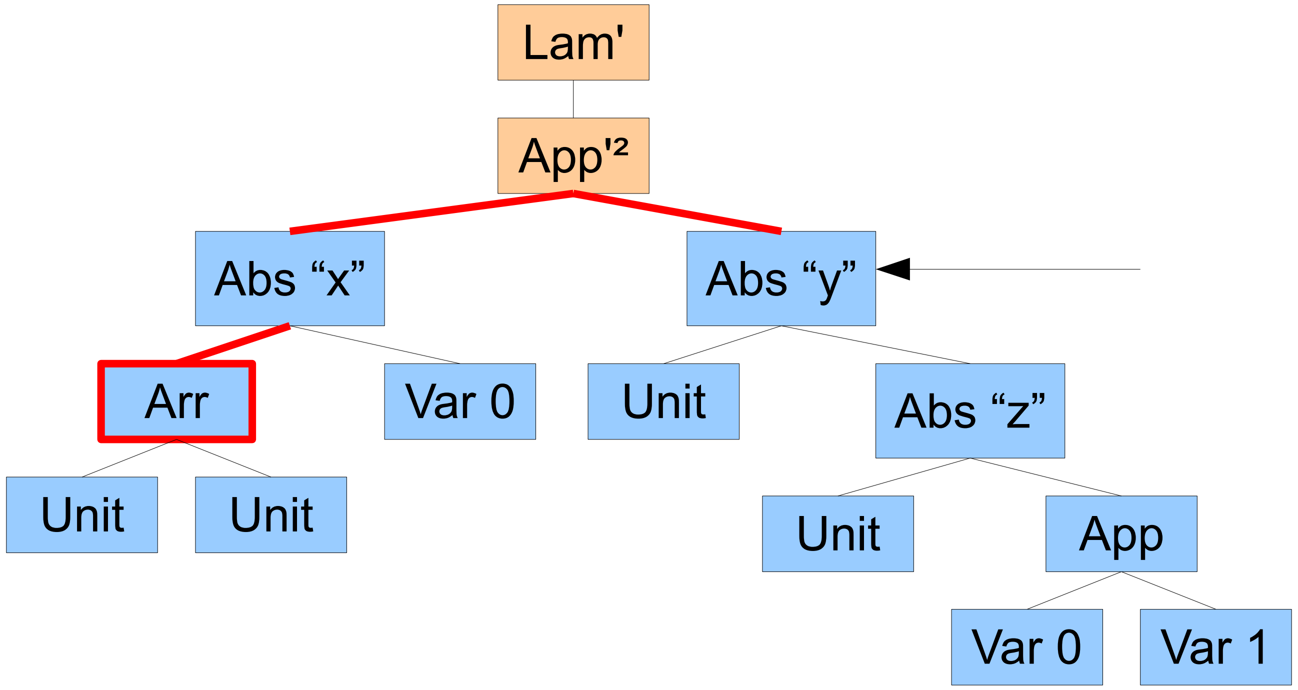
Bookmarks

```
data Route l from to where
  Route :: (...) =>
            Path l (Movement l Up) from mid →
            Path l (Movement l Down) mid to →
            Route l from to
```



Cursors with Bookmarks

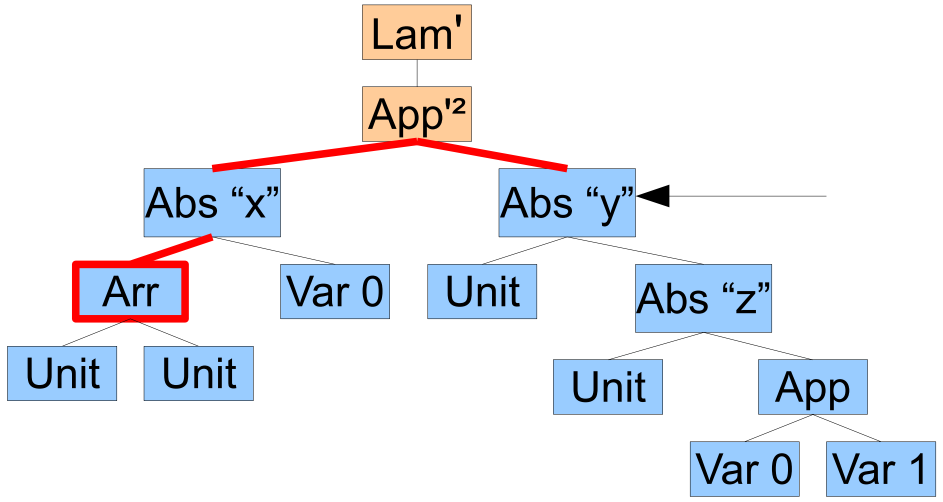
```
data Cursor l x a = (Reify l a) => Cursor {
    it :: a,
    ctx :: Path l (Context l) a l,
    log :: Route l a x
}
```

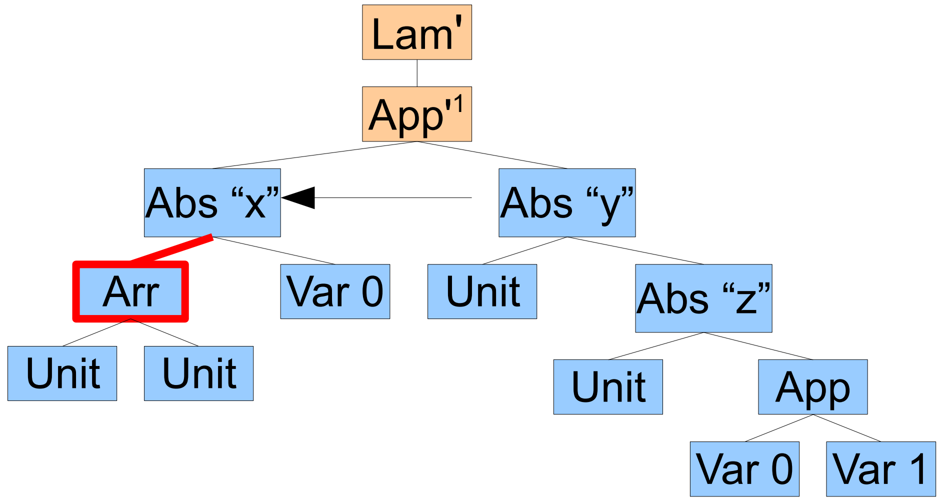


Cursors with Bookmarks

```
data Cursor l x a = (Reify l a) => Cursor {
    it :: a,
    ctx :: Path l (Context l) a l,
    log :: Route l a x
}
```

```
genericMoveLeft  :: (Language l) ⇒
  Cursor l x a →
  Maybe (∃ b . Cursor l x b)
```





Summary

- Heterogeneous underlying “Language”
- Scripts to autogenerate the boilerplate
- Simply specify lexical binding
- Complete (and other) traversals made easy
- Bookmarks

Thank you for listening!

Preliminary - GADTs

```haskell
data Exists a where
  Exists :: a b -> Exists a

data TyEq a b where
  Eq :: TyEq a a
```

Moving around

```
applyMovement :: MovementI dir from to →
                  Cursor from → Maybe (Cursor to)
applyMovement mov (Cursor it ctx)
   = case (reifyDirectionI mov) of
  UpT   →  case ctx of
   Step up ups -> case (up `contextMovementEq` mov) of
      Just Eq -> Just $ Cursor (buildOne up it) ups
      Nothing -> Nothing
    Stop -> Nothing
  DownT -> case (unbuildOne mov it) of
   Just (ctx', it') → Cursor it' (Step ctx' ctx)
   Nothing → Nothing
```

```
data DirectionT dir where
   UpT   :: DirectionT Up
   DownT :: DirectionT Down
```

```
buildOneI :: ContextI a b → a → b

unbuildOneI :: MovementI Down a b → a →
                     Maybe (ContextI b a, b)

reifyDirectionI :: MovementI dir a b → DirectionT dir

contextMovementEq :: ContextI a b → MovementI Up a c → Maybe (TyEq b c)
```

Generalizing

```
class Language l where
  data Context l :: * → * → *
  data Movement l :: * → * → * → *
  ...

  buildOne :: Context l a b → a → b

  unbuildOne :: Movement l Down a b → a →
                Maybe (Context l b a, b)

  reifyDirection :: Movement l d a b → DirectionT d

  contextToMovement :: Context l a b →
                       Movement l Up a b

  movementEq :: Movement l d a b → Movement l d a c →
                Maybe (TyEq b c)

  ...
```

Generalizing

```
instance Language Lam where
  data Context Lam from to = CW (ContextI from to)
  data Movement Lam d from to = MW (MovementI d from to)
  ...

  buildOne (CW x) = buildOneI x
  unbuildOne (MW m) a = fmap (first CW) (unbuildOneI m a)
  reifyDirection (MW x) = reifyDirectionI x
  movementEq (MW x) (MW y) = movementEqI x y
  contextToMovement (CW x) = MW (contextToMovementI x)
  ...
```