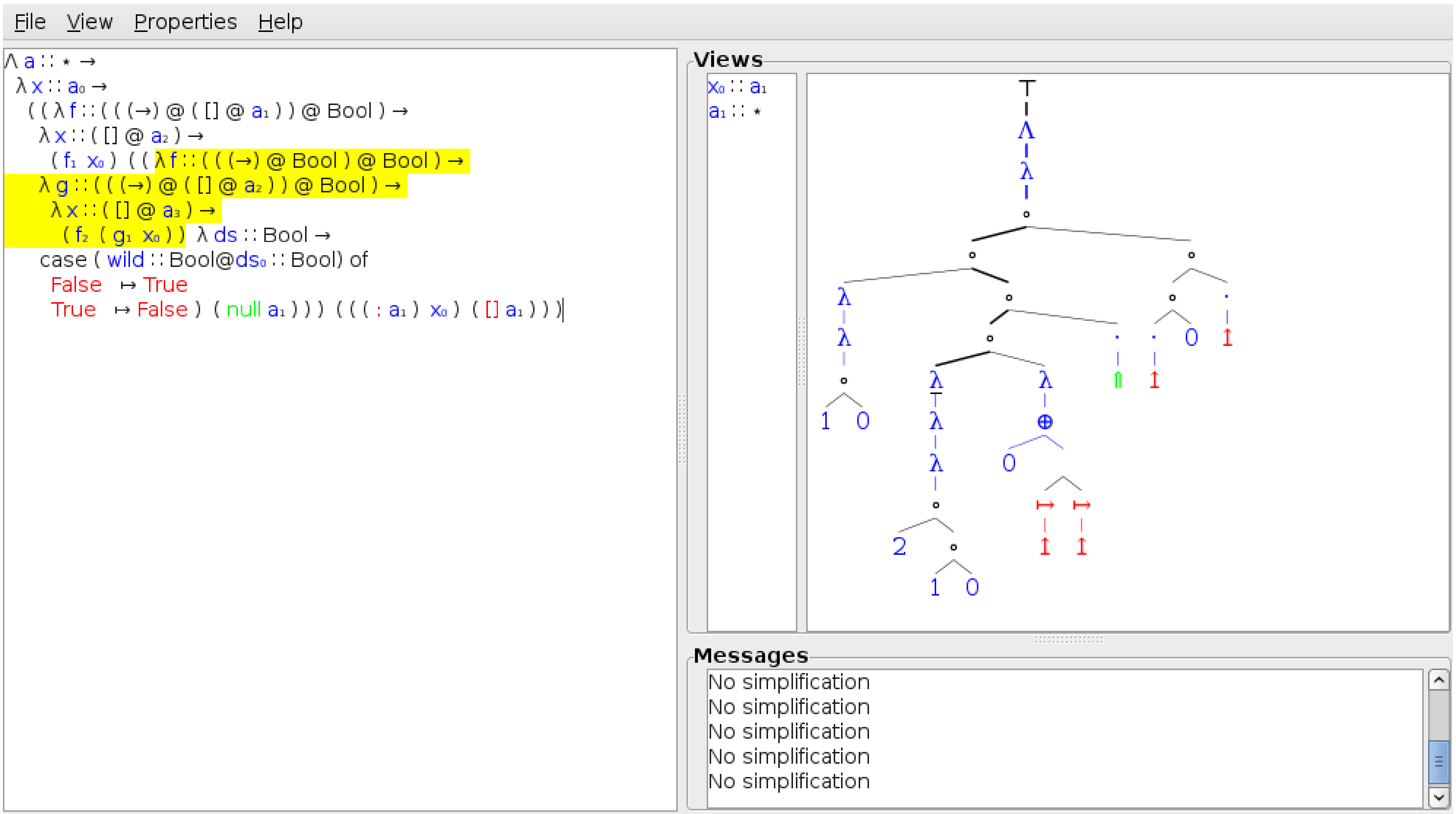
CLASE

Cursor Library for A Structured Editor

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

Motivation


File View Properties Help
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)))
Views
x₀ :: a₁
a₁ :: ⋆
Messages
No simplification
No simplification
No simplification
No simplification
No simplification

Outline

Preliminaries

An example language

Making a simple cursor data structure

Moving that cursor around

Generalizing slightly

Rendering problem

Rendering solution

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/

Preliminary - GADTs

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

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

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

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

Preliminary - GADTs

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

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

Towards Clase Zippers

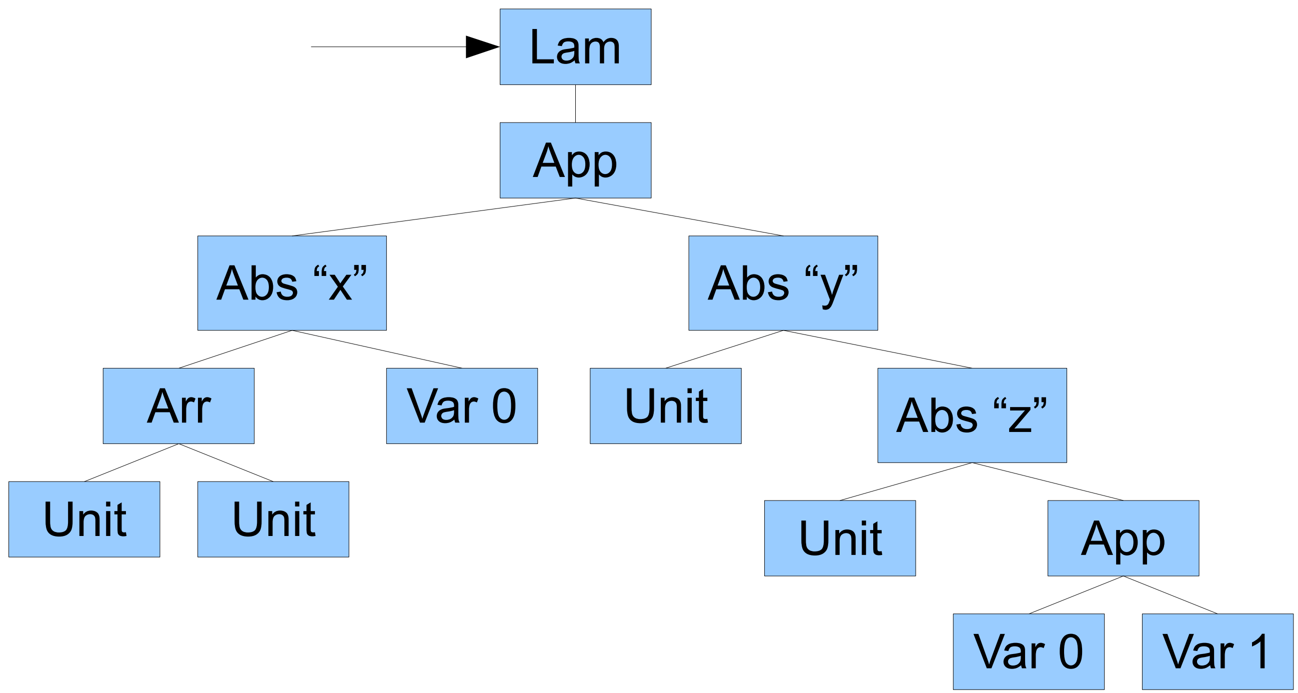
```
data Lam = Lam Exp

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

data Type
  = Unit
  | Arr Type Type
```

Towards CLASE Zippers

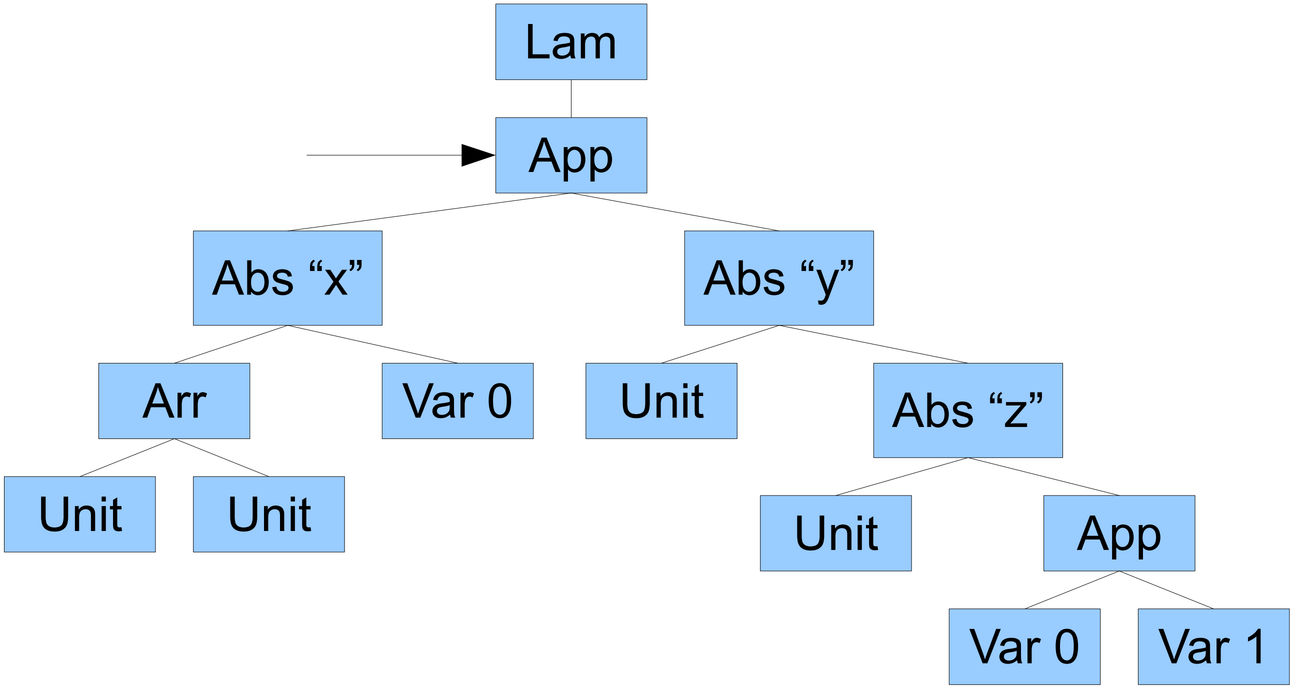
```
sample = Lam (
            App (Abs “x” (Unit `Arr` Unit) (Var 0))
                 (Abs “y” Unit
                    (Abs “z” Unit
                       (App (Var 0)
                            (Var 1)))))
```

(λx:τ→τ.x)(λy:τ.λz:τ.(z y))

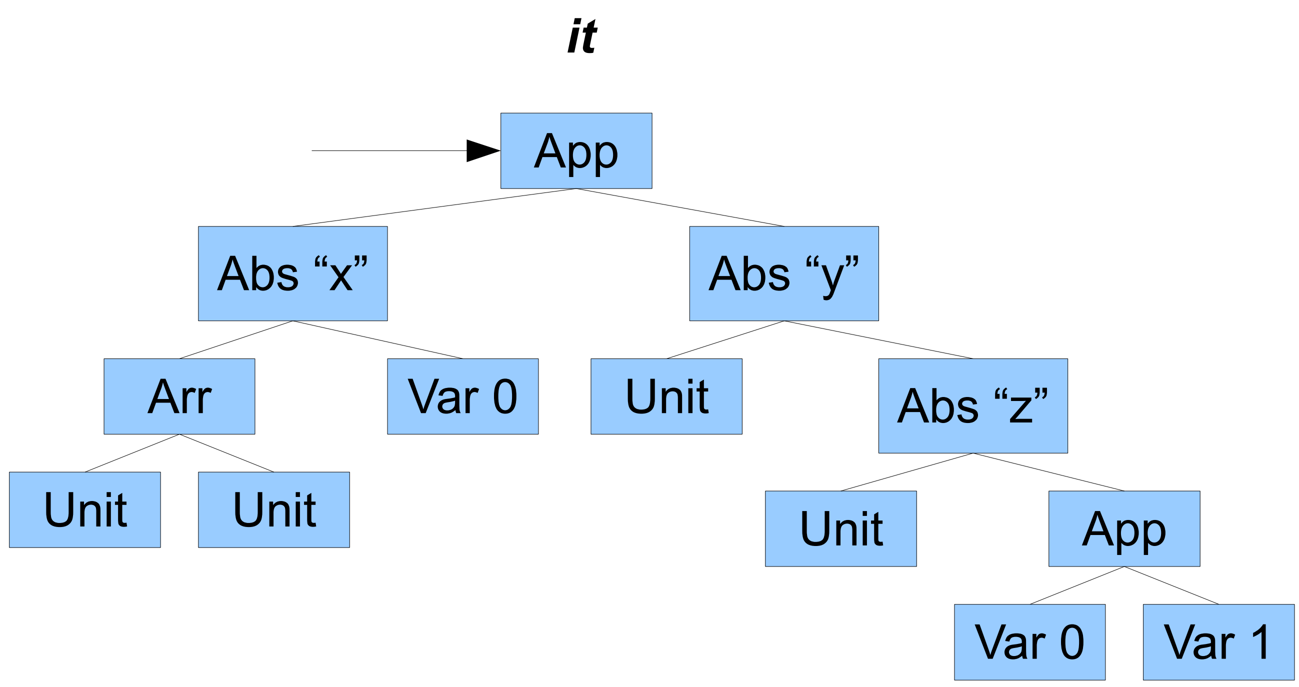
Towards CLASE Zippers



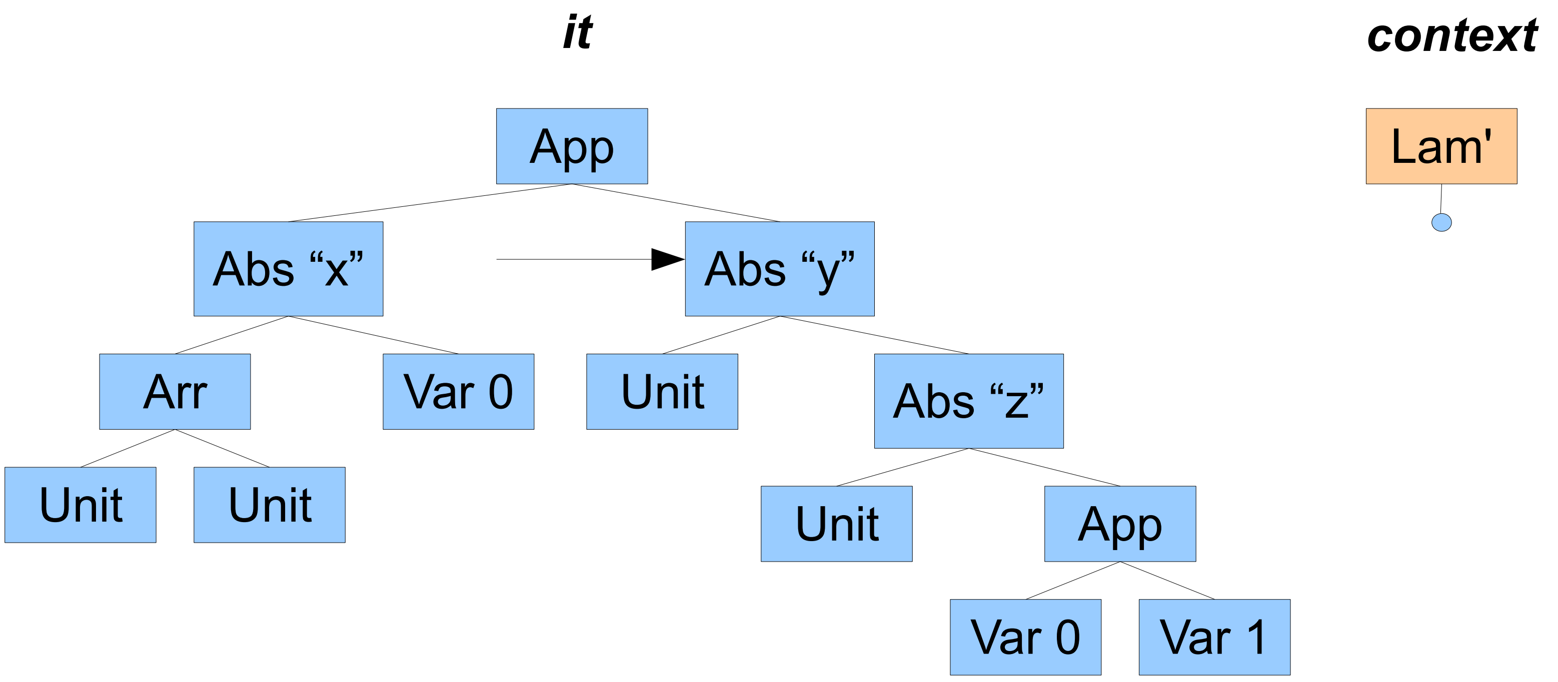
Towards CLASE Zippers



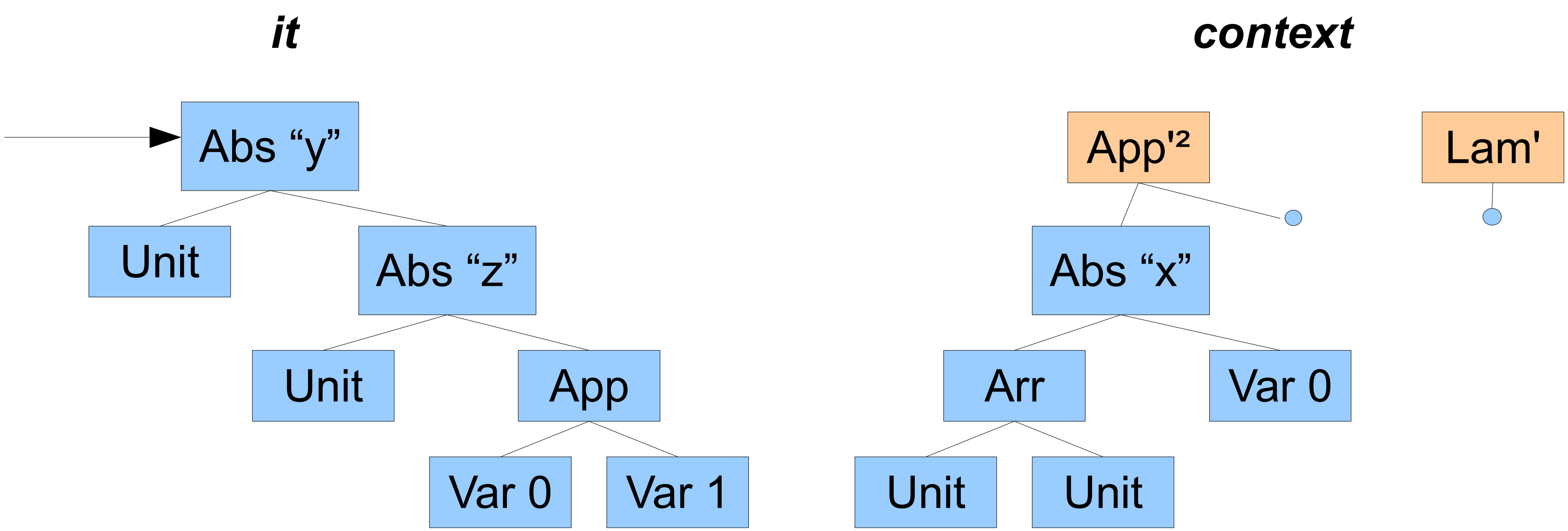
Towards CLASE Zippers



Towards CLASE Zippers

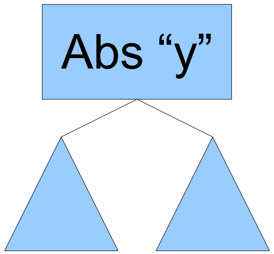


Towards CLASE Zippers

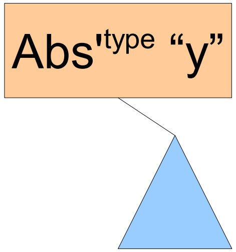


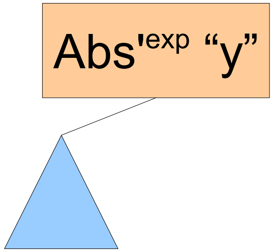
Single Contexts

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




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





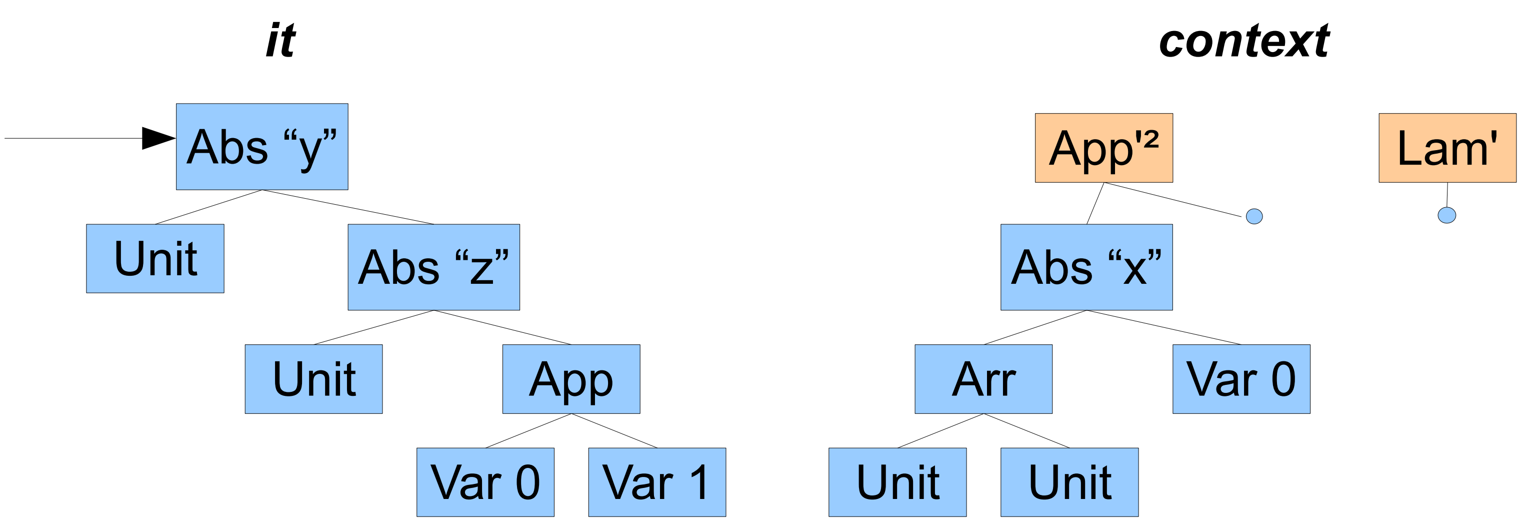
Chaining Contexts

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

```
[]
:
```

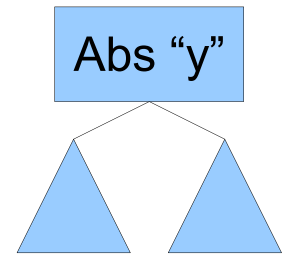
A Cursor

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



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 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
   ...
```

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 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)
```

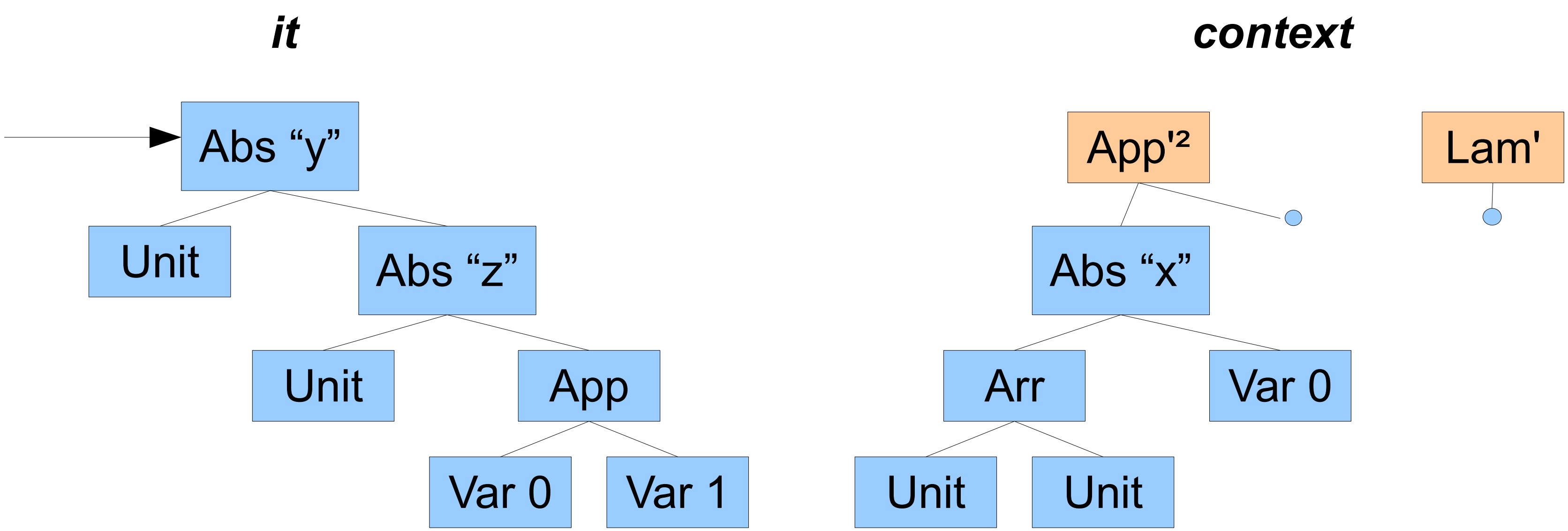
Generalizing

```
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

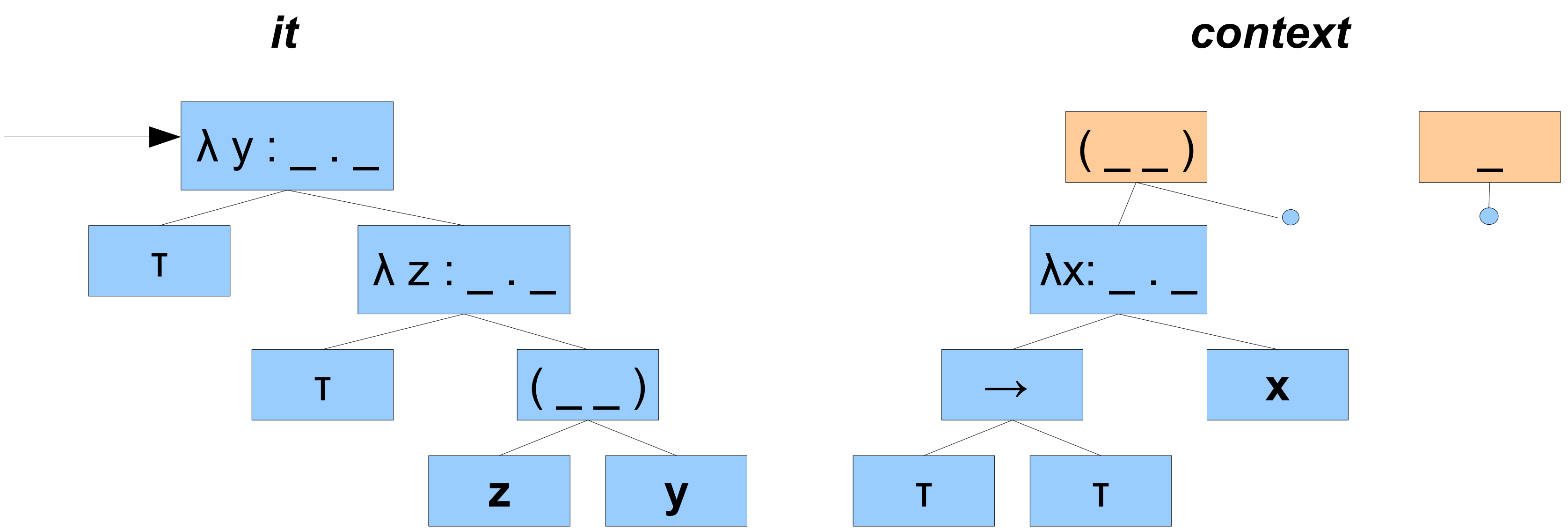
```
instance Language Lam where
  data Context Lam from to = CW (ContextI from to)
  data Movement Lam d from to = MW (MovementI d from to)
  ...

  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)
  ...
```

Rendering Problem



(λx:τ→τ.x)(⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem



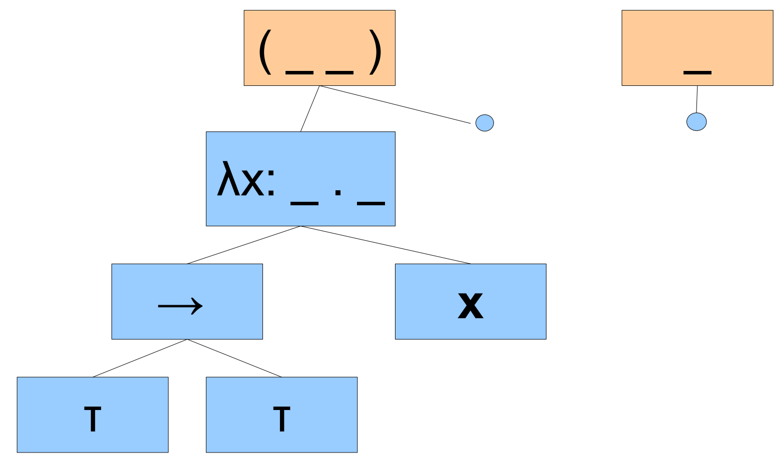
(λx:τ→τ.x)(⊳λy:τ.λz:τ.(z y)⊲)

Rendering Problem

it

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

context



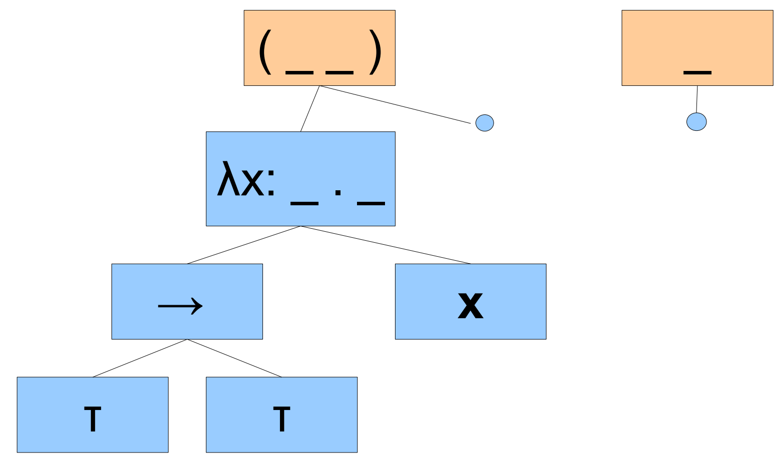
$(\lambda x{:}\tau\rightarrow\tau.x)(\rhd\lambda y{:}\tau.\lambda z{:}\tau.(z\ y)\lhd)$

Rendering Problem

it

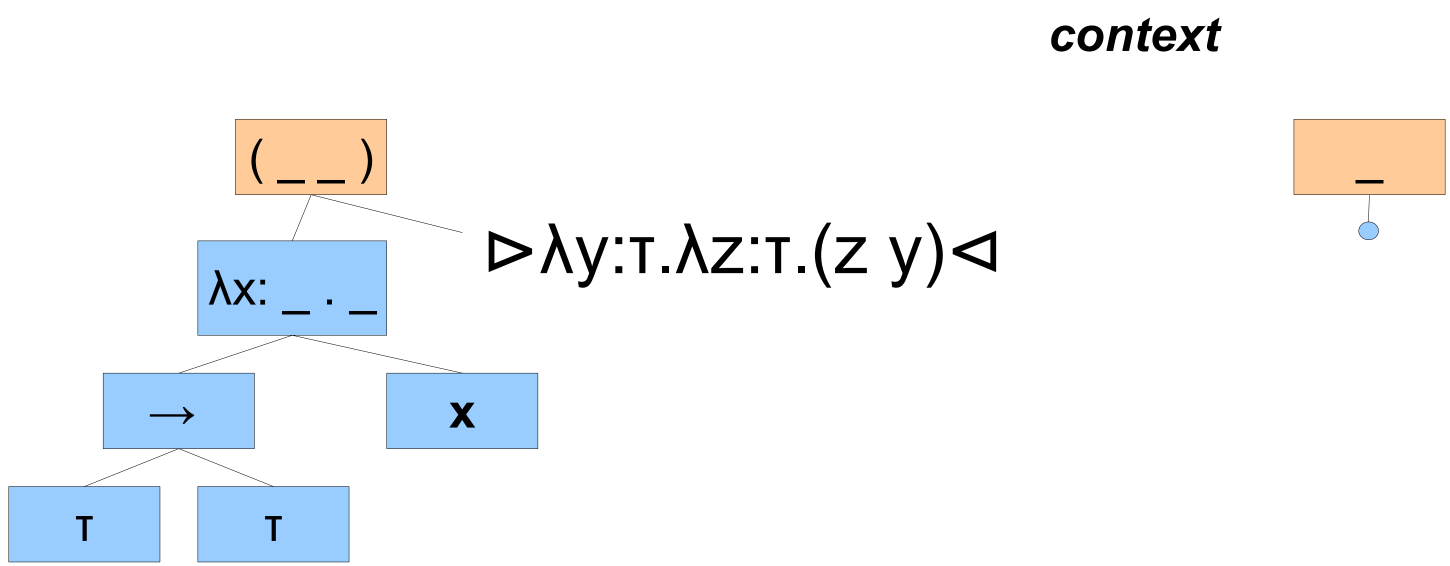
$\rhd \lambda y{:}\tau.\lambda z{:}\tau.(z\ y) \lhd$

context



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

Rendering Problem

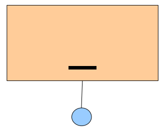


$$(\lambda x{:}\tau\to\tau.x)(\rhd\lambda y{:}\tau.\lambda z{:}\tau.(z\ y)\lhd)$$

Rendering Problem

context

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



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)
...
```

Binding...

```
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

...
```

Rendering...

```
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
```

Rendering...

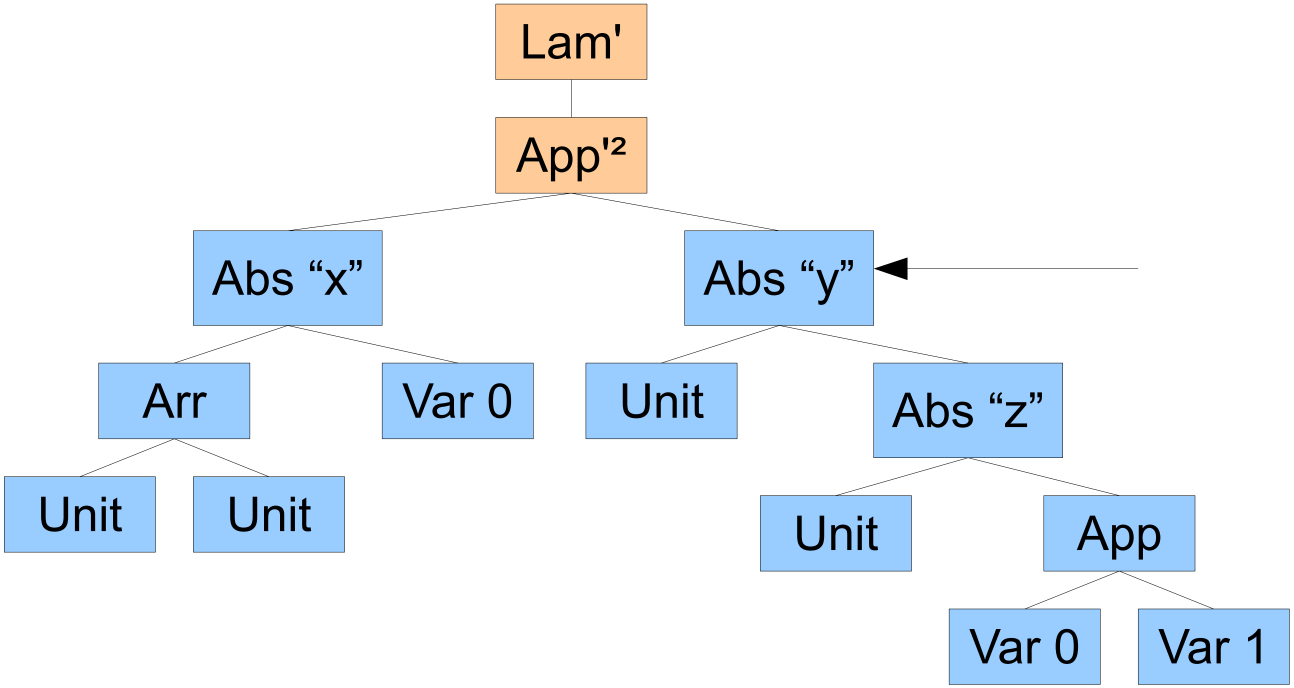
```
instance LamTraversalAdapterExp (M String)
where
  visitAbs (Abs str _ _) ty exp = do
    tys ← ty
    exps ← exp
    return (“λ “ ++ str ++ “ : “
                 ++ tys ++ “ . “ ++ exps)

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

Rendering...

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

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

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

  cursor :: l → t → t


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

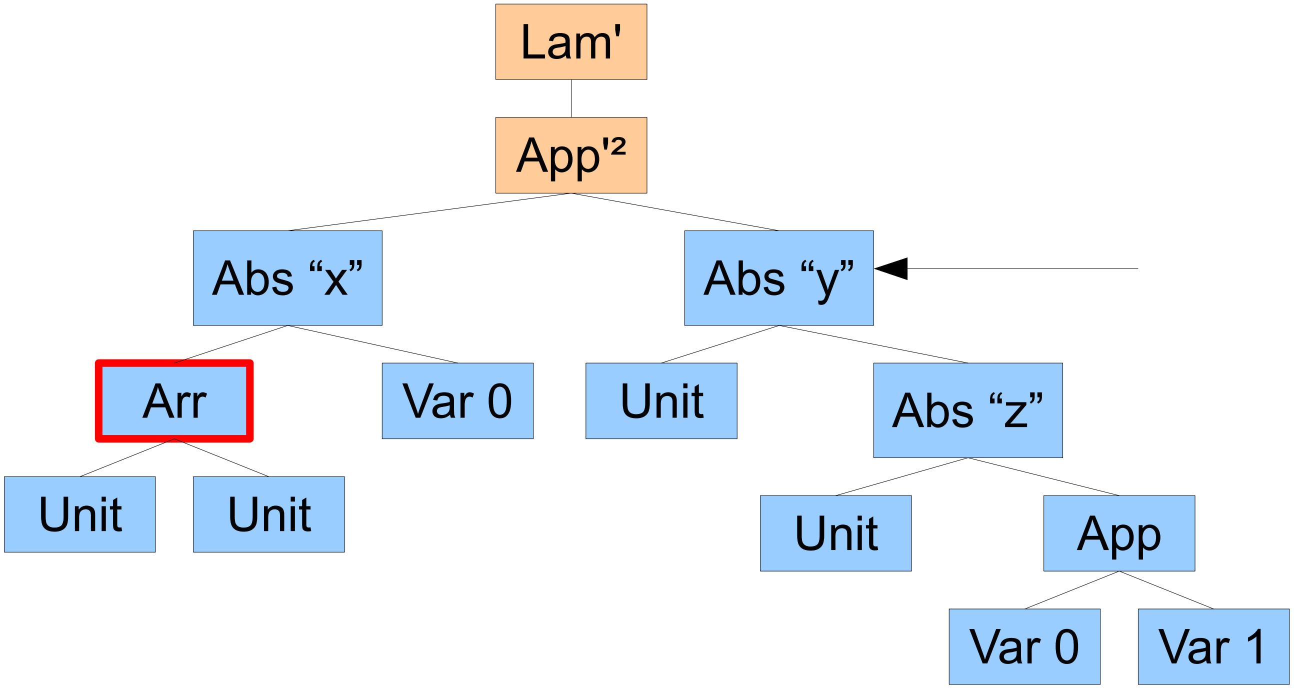
```
instance (LamTraversalAdapterLam t,
          LamTraversalAdapterExp t,
          LamTraversalAdapterType t,
          LamTraversalAdapterCursor t,
          Bound Lam t) => Traversal Lam t where
```

Bookmarks



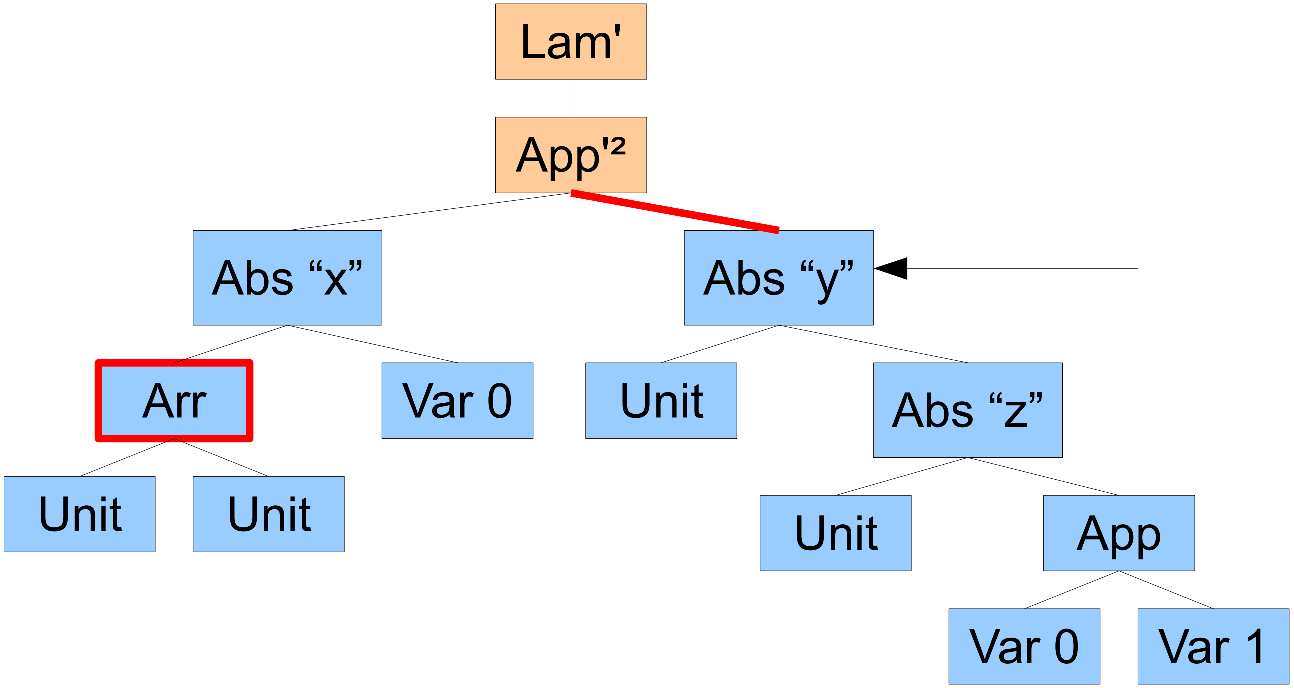
Lam'
App'²
Abs "x"
Abs "y"
Arr
Var 0
Unit
Abs "z"
Unit
Unit
Unit
App
Var 0
Var 1

Bookmarks



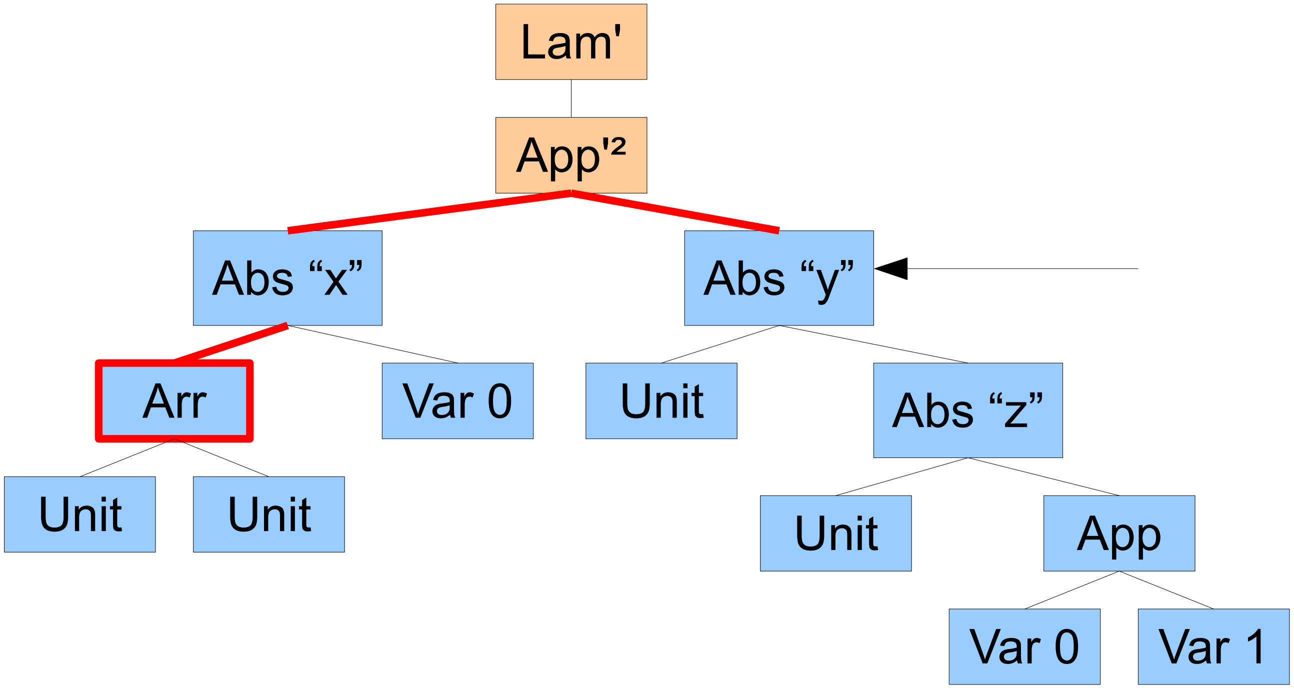
Lam'
App'²
Abs “x”
Abs “y”
Arr
Var 0
Unit
Abs “z”
Unit
Unit
Unit
App
Var 0
Var 1

Bookmarks



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

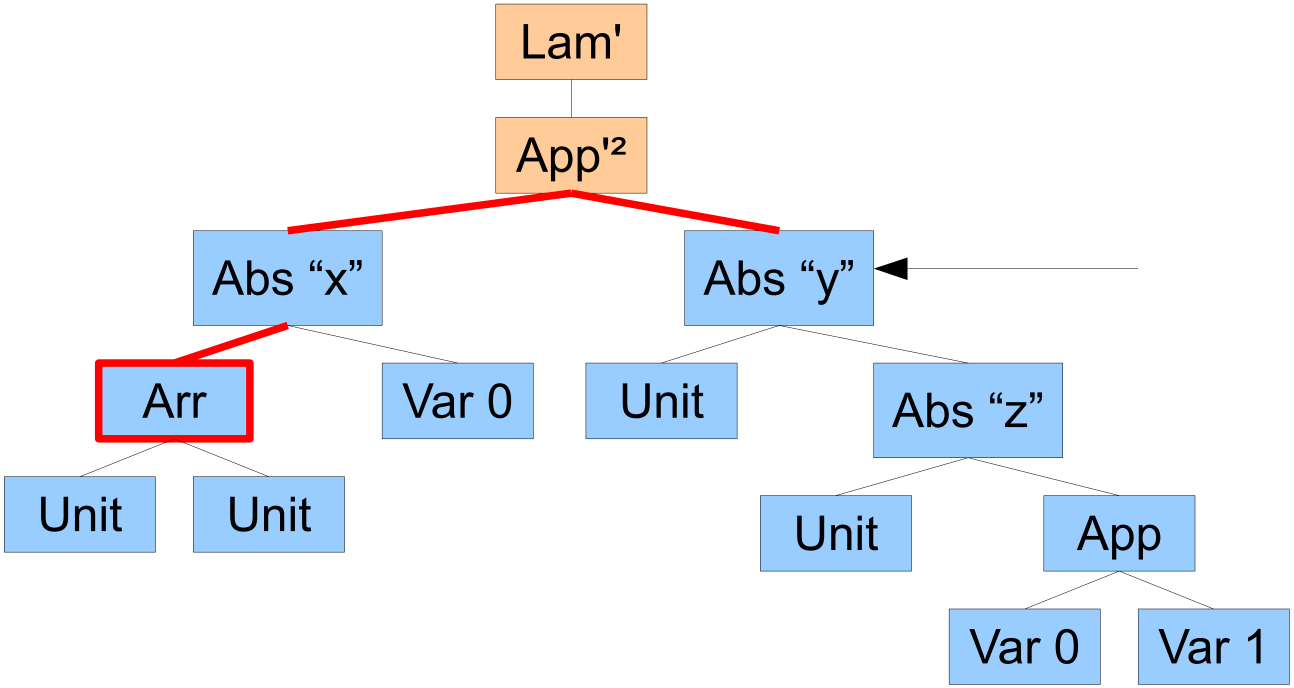
Bookmarks



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

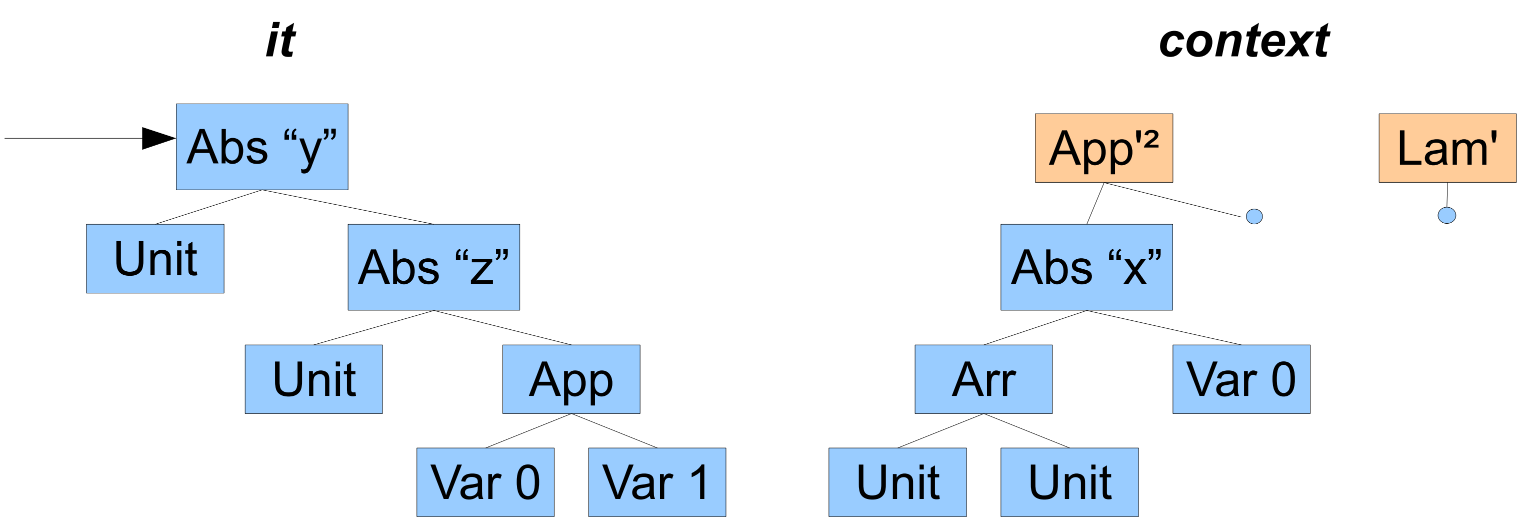
Bookmarks

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



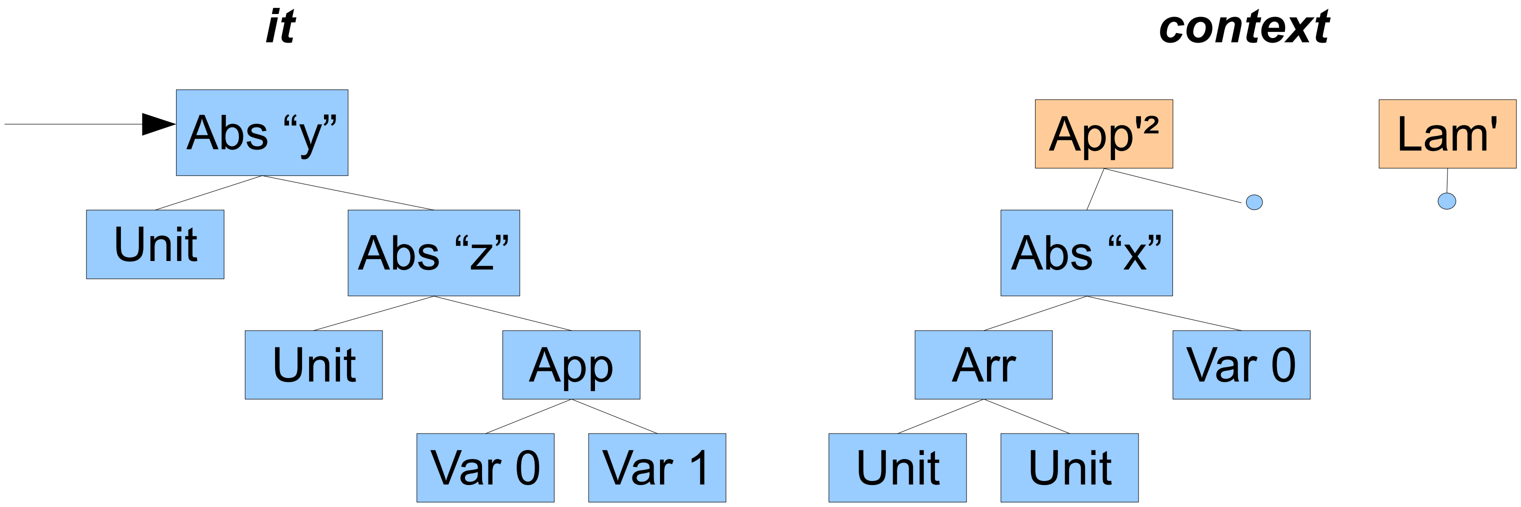
Cursors with Bookmarks

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



Cursors with Bookmarks

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



Moving (redux)

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

genericMoveDown :: (Language l) ⇒
  Cursor l x a → Maybe (CursorWithMovement l Down x a)

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

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

```
data CursorWithMovement l d x from where
  CWM :: (Reify l to) ⇒ Cursor l x to → Movement l d from to →
         CursorWithMovement l d x from
```

Demo

Thank you for listening!