Strengthening the Zipper

CLASE - Cursor Library for A Structured Editor

Tristan Allwood (tora@doc.ic.ac.uk)
Susan Eisenbach (s.eisenbach@imperial.ac.uk)
Motivation

\[ \lambda a :: \ast \rightarrow \lambda x :: a \rightarrow ( ( \lambda f :: ( ( \rightarrow ) @ ( [] @ a ) ) @ \text{Bool}) \rightarrow \lambda x :: ( [] @ a ) \rightarrow ( f _{1} x_{0} ) ( ( \lambda f :: ( ( \rightarrow ) @ \text{Bool}) @ \text{Bool}) \rightarrow \lambda g :: ( ( ( \rightarrow ) @ ( [] @ a ) ) @ \text{Bool}) \rightarrow \lambda x :: ( [] @ a ) \rightarrow ( f_{1} ( g_{1} x_{0} ) ) \lambda ds :: \text{Bool} \rightarrow \text{case ( wild :: \text{Bool}@ds :: \text{Bool}) of} \]

\[ \begin{align*} & \text{False} \mapsto \text{True} \\ & \text{True} \mapsto \text{False} \end{align*} \]

\[ ( \text{null } a ) ( ( [ ] a ) ( x_{0} ) ( [ ] a ))  \]
Motivation

- In place updates
- Side effects
- Pointers
- IO
Motivation

- In place updates
- Side effects
- Pointers
- IO
## Towards Clase Zippers

<table>
<thead>
<tr>
<th>data Root = Root Stat</th>
</tr>
</thead>
<tbody>
<tr>
<td>data Stat</td>
</tr>
<tr>
<td>= Assign Var IExp</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>data BExp</td>
</tr>
<tr>
<td>= LT IExp IExp</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>data IExp</td>
</tr>
<tr>
<td>= Add IExp IExp</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Towards CLASE Zippers

```
Cursor {  
  it = Root ((Assign "x" (Const 3) `Then`  
               (If (LT (IVar "x") (Const 4)) (Assign "y" (Const 4))))) (Assign "x" (Const 3))  
  ctx = Stop  
} :: Cursor Root

data Root = Root Stat

data Stat  
  = Assign Var IExp  
    `Then` Stat  
    `If` BExp Stat  
  `=` Assign Var IExp

data BExp  
  = LT IExp IExp  
    GT IExp IExp

data IExp  
  = Add IExp IExp  
    IVar Var  
    Const Int

data Path tc start end where  
  Stop :: Path here here  
  Step :: tc start mid ->  
        Path tc mid end ->  
        Path tc start end

data Cursor a = Cursor {  
  it :: a,  
  ctx :: Path ContextI a Root  
}
Towards CLASE Zippers

```
data Root = Root Stat
  data Stat
    = Assign Var IExp
    | If BExp Stat
    | Stat `Then` Stat

data BExp
  = LT IExp IExp
  | GT IExp IExp

data IExp
  = Add IExp IExp
  | IVar Var
  | Const Int

data Path tc start end where
  Stop :: Path here here
  Step :: tc start mid →
        Path tc mid end →
        Path tc start end

data Cursor a = Cursor {
  it :: a,
  ctx :: Path ContextI a Root
}```
Towards CLASE Zippers

Data

Root = Root Stat

Stat = Assign Var IExp | If BExp Stat | Stat `Then` Stat

BExp = LT IExp IExp | GT IExp IExp

IExp = Add IExp IExp | IVar Var | Const Int

Path tc start end where
Stop :: Path here here
Step :: tc start mid → Path tc mid end → Path tc start end

Data Cursor a = Cursor {
  it :: a,
  ctx :: Path ContextI a Root
}

Cursor {
  it = ((Assign "x" (Const 3) `Then` (If (LT (IVar "x") (Const 4))) Assign "y" (Const 4))),
  ctx = Step Root` Stop
} :: Cursor Stat
Towards CLASE Zippers

```
Cursor a = Cursor {
  it :: a,
  ctx :: Path ContextI a Root
}
data Cursor a = Cursor {
  it :: a,
  ctx :: Path ContextI a Root
}
data Path tc start end where
  Stop :: Path here here
  Step :: tc start mid →
         Path tc mid end →
         Path tc start end
data Root' :: ContextI Stat Root
data Stat = Assign Var IExp
           | If BExp Stat
            Stat `Then` Stat
data BExp = LT IExp IExp
           | GT IExp IExp
data IExp = Add IExp IExp
           | IVar Var
           | Const Int
```

```
Assign “x”
   Const 3
   IVar “x”
   Const 4
   Const 4

If
   LT
   Assign “y”
   Const 4
   Const 4

Then

ctx

Root' = Root Stat
```
Towards CLASE Zippers

The diagram illustrates a zipper data structure with nodes labeled with logical expressions and operations. The zipper operations include:
- `lt`: less than
- `assign`: assignment
- `const`: constant
- `if`: if-then-else
- `then'1`: a context operation
- `root'`: another context operation
- `stop`: stop operation

The code snippet below defines the operation of the zipper:

```
data Root = Root Stat

data Stat
    = Assign Var IExp
    | LT IExp IExp
    | GT IExp IExp

data IExp
    = Add IExp IExp
    | IVar Var
    | Const Int

data Path tc start end where
    Stop :: Path here here
    Step :: tc start mid \rightarrow tc mid end \rightarrow tc start end

data Cursor a = Cursor {
    it :: a,
    ctx :: Path ContextI a Root
}
```
Towards CLASE Zippers

```plaintext
Cursor {  
  it = If (LT (IVar "x") (Const 4)) (Assign "y") (Const 4)) (Assign "x" (Const 3))) (Stat)  
  ctx = Step (Then'1 (Assign "x" (Const 3))) (Stat)  
} :: Cursor Stat
```

```
data Root = Root Stat  
data Stat = Assign Var IExp BExp Stat Stat `Then` Stat  
  BExp = LT IExp IExp | GT IExp IExp  
data IExp = Add IExp IExp | IVar Var | Const Int  
data Path tc start end where  
  Stop :: Path here here  
  Step :: tc start mid -> Path tc mid end -> Path tc start end
```

```
data Cursor a = Cursor {  
  it :: a,  
  ctx :: Path ContextI a Root  
}```
Towards CLASE Zippers

Cursor {
  it = LT (IVar "x") (Const 4),
  ctx = Step (If' BExp (Assign "y" (Const 4))) (Step "x" (Const 3))) (Step Root' Stop)
} :: Cursor BExp
Bookmarks
data Route from to where
  Route :: Path (MovementI Up) from mid \rightarrow Path (MovementI Down) mid to \rightarrow Route from to

data Path tc start end where
  Stop :: Path here here
  Step :: tc start mid \rightarrow Path tc mid end \rightarrow Path tc start end
data Route from to where
Route :: Path (MovementI Up) from mid \rightarrow Path (MovementI Down) mid to \rightarrow Route from to

data Path tc start end where
Stop :: Path here here
Step :: tc start mid \rightarrow Path tc mid end \rightarrow Path tc start end

Assign "x"
If
Assign "y"

Then

Const 3

IVar "x"
Const 4
Const 4

LT

Root
Bookmarks

**data Route from to where**

Route :: Path (MovementI Up) from mid →
       Path (MovementI Down) mid to →
       Route from to

**data Path tc start end where**

Stop :: Path here here
Step :: tc start mid →
       Path tc mid end →
       Path tc start end

MAAssignToIExp :: MovementI Down Exp IExp
MUp :: MovementI Down b a → MovementI Up a b

Route (Step (MUp MAAssignToIExp) Stop)
       (Stop) :: Route IExp Exp
data Route from to where
Route :: Path (MovementI Up) from mid →
Path (MovementI Down) mid to →
Route from to

data Path tc start end where
Stop :: Path here here
Step :: tc start mid →
Path tc mid end →
Path tc start end

LT

MUp :: MovementI Down b a → MovementI Up a b
MAssignToIExp :: MovementI Down Exp IExp
MThenToIExp¹ :: MovementI Down Exp Exp

Route (Step (MUp MAssignToIExp)) (Step (MUp MThenToIExp¹) Stop))
(Stop) :: Route IExp Exp
Assign “x"

Const 3

IVar “x"

Const 4

Assign “y"

Con 4

LT

Then

If

Root

data Route from to where
Route :: Path (MovementI Up) from mid → Path (MovementI Down) mid to → Route from to

data Path tc start end where
Stop :: Path here here
Step :: tc start mid → Path tc mid end → Path tc start end

MThenToIExp² :: MovementI Down Exp Exp
MThenToIExp¹ :: MovementI Down Exp Exp
MAssignToIExp :: MovementI Down Exp IExp
MUp :: MovementI Down b a → MovementI Up a b

Route (Step (MUp MAssignToIExp)) (Step (MUp MThenToExp¹) Stop))
(Step MThenToIExp² Stop) :: Route IExp Exp

Strengthening the Zipper

Allwood & Eisenbach
data Route from to where
  Route :: Path (MovementI Up) from mid → Path (MovementI Down) mid to → Route from to

data Path tc start end where
  Stop :: Path here here
  Step :: tc start mid → Path tc mid end → Path tc start end

MIfToBExp :: MovementI Down Exp BExp
MThenToIExp^2 :: MovementI Down Exp Exp
MThenToIExp^1 :: MovementI Down Exp Exp
MAssignToIExp :: MovementI Down Exp IExp
MUp :: MovementI Down b a → MovementI Up a b

Route (Step (MUp MAssignToIExp) (Step (MUp MThenToIExp^1) Stop))
  (Step MThenToIExp^2 (Step MIfToBExp Stop)) :: Route IExp BExp
Cursors with Bookmarks

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

```
Cursor {  
  it = If (LT (IVar "x") (Const 4)) (Assig "y" (Const 4)),  
  ctx = Step (Then'1 (Assig "x" (Const 3))) (Step Root' Stop)  
} :: Cursor Stat
```
Cursors with Bookmarks

\[
\text{data Cursor } x \ a = \text{Cursor } \{
\begin{align*}
\text{it} &:: a, \\
\text{ctx} &:: \text{Path ContextI} \ a \ \text{Root}, \\
\text{log} &:: \text{Route} \ a \ x
\end{align*}
\}
\]

Cursor \{
\begin{align*}
\text{it} &= \ \text{If (LT (IVar “x”) (Const 4)) (Assig “y” (Const 4))}, \\
\text{ctx} &= \ \text{Step (Then’1 (Assig “x” (Const 3))) (Step Root’ Stop)}, \\
\text{log} &= \ \text{Route Stop (Step MIfToExp (Step MAssigToIExp Stop))}
\end{align*}
\} :: \text{Cursor IExp Stat}

Moving

data Cursor x a = Cursor {
  it :: a,
  ctx :: Path ContextI a Root,
  log :: Route a x
}

genericMoveUp, genericMoveDown, genericMoveLeft, genericMoveRight :: Cursor x a → Maybe (∃a'. . Cursor x a')
Writing a GUI

data CursorHolder where
  CH :: Cursor a a →
  Map Int (∃a'. Route a a') →
  CursorHolder

mainLoop cursorHolder = do
  key ← getKey
  let cursorHolder' = onKeyPress key cursorHolder
  render cursorHolder'
  mainLoop cursorHolder'
onKeyPress – new bookmark

```haskell
data CursorHolder where
  CH :: Cursor a a →
      Map Int (a' . Route a a') →
      CursorHolder

onKeyPress (KeySaveBookmark i) (CH cursor bookmarks) =
  let bookmarks' = Map.insert i (Exists emptyRoute) bookmarks
  in
      CH cursor bookmarks'

emptyRoute :: Route a a
emptyRoute = Route Stop Stop
```
```
data CursorHolder where
  CH :: Cursor a a →
  Map Int (∃a' . Route a a') →
  CursorHolder
```

```haskell
onKeyPress KeyDown ch@(CH cursor bookmarks) = fromMaybe ch $ do
  Exists cursor' ← genericMoveDown cursor
  ...
```
onKeyPress - movement

```haskell
data CursorHolder where
  CH :: Cursor a a →
  Map Int (∃a' . Route a a') →
  CursorHolder

onKeyPress KeyDown ch@(CH cursor bookmarks)
  = fromMaybe ch $ do
    Exists cursor' ← genericMoveDown cursor
    ...
```

∃ o .
  cursor :: Cursor o o
  bookmarks :: Map Int (∃a' . Route o a')

∃ n .
  cursor' :: Cursor o n
onKeyPress - movement

```haskell
data CursorHolder where
  CH :: Cursor a a →
      Map Int (∃a'. Route a a') →
      CursorHolder
```

```haskell
onKeyPress KeyDown ch@(CH cursor bookmarks)
  = fromMaybe ch $ do
    Exists cursor' ← genericMoveDown cursor
    return (CH cursor' bookmarks)
```

```
∃ o .
  cursor :: Cursor o o    bookmarks :: Map Int (∃a' . Route o a')

∃ n .
  cursor' :: Cursor o n
```
onKeyPress - movement

```
data CursorHolder where
  CH :: Cursor a a →
    Map Int (∃a'. Route a a') →
    CursorHolder

appendRoute :: Route a b →
    Route b c →
    Route a c
```

```
onKeyPress KeyDown ch@(CH cursor bookmarks)
  = fromMaybe ch $ do
    Exists cursor' ← genericMoveDown cursor
    let bookmarks' =
        Map.map (λ bm → ∃ (log cursor' `appendRoute` bm))
          bookmarks
```

```
exists o .
  cursor :: Cursor o o
  bookmarks :: Map Int (∃a'. Route o a')
exists n .
  cursor' :: Cursor o n
  bookmarks' :: Map Int (∃a'. Route n a')
```
onKeyPress - movement

```haskell
data CursorHolder where
  CH :: Cursor a a →
  Map Int (∃ a' . Route a a') →
  CursorHolder

appendRoute :: Route a b →
  Route b c →
  Route a c

onKeyPress KeyDown ch@(CH cursor bookmarks)
  = fromMaybe ch $ do
    Exists cursor' ← genericMoveDown cursor
    let bookmarks' =
        Map.map (∃ bm → ∃ (log cursor' `appendRoute` bm))
            bookmarks
    return (CH cursor' bookmarks')

∃ o .
  cursor :: Cursor o o
  bookmarks :: Map Int (∃ a'.Route o a')

∃ n .
  cursor' :: Cursor o n
  bookmarks' :: Map Int (∃ a'.Route n a')
```

Strengthening the Zipper

Allwood & Eisenbach
onKeyPress - movement

```haskell
data CursorHolder where
  CH :: Cursor a a →
      Map Int (∃a' . Route a a') →
      CursorHolder

resetLog :: Cursor x a →
          Cursor a a

onKeyPress KeyDown ch@(CH cursor bookmarks) = fromMaybe ch $ do
  Exists cursor' ← genericMoveDown cursor
  let bookmarks' =
      Map.map (∃ bm → ∃ (log cursor' `appendRoute` bm))
        bookmarks
  return (CH (resetLog cursor') bookmarks')
```

```
∃ o .
  cursor :: Cursor o o
  bookmarks :: Map Int (∃a'.Route o a')

∃ n .
  cursor' :: Cursor o n
  bookmarks' :: Map Int (∃a'.Route n a')
```
onKeyPress - movement

So the Cursor design means the invariant:

**Bookmarks stay in sync with focus**

Can be partially encoded in the type system

and checked statically by the compiler

\[ \exists n \cdot \text{cursor': Cursor } o \ n \ \text{bookmarks': Map } \text{Int} (\exists a'. \text{Route } n \ a') \]
onKeyPress - movement

```haskell
data CursorHolder where
  CH :: Cursor a a →
  Map Int (∃a'. Route a a') →
  ∃ →
  CursorHolder

resetLog :: Cursor a a →
  Map Int (∃a'. Route a a') →
  ∃ σ o 
  ∃ n .
  cursor' :: Cursor o n
  bookmarks' :: Map Int (∃a'. Route n a')
```

So the Cursor design means the invariant:

**Bookmarks stay in sync with focus**

Can be partially encoded in the type system

and checked statically by the compiler
and checked statically by the compiler

\[ \exists n \cdot \text{cursor'} :: \text{Cursor } o n \quad \text{bookmarks'} :: \text{Map } \text{Int} \quad (\exists a'.\text{Route } n \ a') \]
onKeyPress - select bookmark

onKeyPress (KeyLoadBookmark i) ch@(CH cursor bookmarks) = fromMaybe ch $ do
  Exists route ← Map.lookup i bookmarks
  Exists cursor' ← cursor `followRoute` route
  let bookmarks' =
      Map.map (\∃ bm → ∃ (log cursor' `appendRoute` bm)) bookmarks
  return (CH (resetLog cursor') bookmarks')

followRoute :: Cursor x a → Route a c → Maybe (Cursor x c)
CLASE also supports...

- Automatic generation of Context and primitive Movement data types from simple data type declarations
- Automatic generation of a closed reflection scheme for user data types
- Adapters to help with traversals of a Cursor, suitable for (e.g.) rendering a Cursor
- Interface to add bound information to traversals
- Simple Persistence (Read/Show) for Cursors
Thank you for listening!

www.zonetora.co.uk/clase/