Instant Zipper

Bram Schuur Ruud Koot

12 November 2010

Research question

- Zipper for heterogenous types
- i.e. The Zipper can traverse into structures of any type
- (Much like SYB zipper)
- But with some extras!

Usage

Given our familiar example..

```
type Salary   = Float
type Manager  = Employee
type Name     = String
data Dept     = D Manager [Employee] deriving Typeable
data Employee = E Name Salary        deriving Typeable
```

```
dept :: Dept
dept =  D doaitse [johan,sean,pedro]
  where doaitse,johan,sean,pedro :: Employee
    doaitse =  E "Doaitse" 8000
    johan   =  E "Johan"   8000
    sean    =  E "Sean"    2600
    pedro   =  E "Pedro"   2400
```

Usage

We can fix it using our zipper as follows:

```
fixDept :: Either String Dept
fixDept =  return (enter dept)
       >>= down Employee
       >>= down Name
       >>= return ∘ setHole "Prof. dr. Swierstra"
       >>= right  Salary
       >>= return ∘ setHole 9000.0
       >>= return ∘ leave
```

Zipper

- A zipper consists of the part of the data structure which is currently *in focus*, together with the *context* in which it appears.

```
data Loc ... = Loc {focus :: hole,context :: Context ... }
```

- The Context is a stack of *one-hole contexts*
- A type for all one-hole contexts of a datatype can be found by taking its derivative

```
data Context ... where
  Empty :: Context ...
  Push  :: Derivative (Rep parent)   -- One-hole context
        → Context parent ...
        → Context ...
```

Derivative

We calculate the derivative data structure of a structure representation using associated datatypes.

```
class Derivable f where
  data Derivative f :: *
instance (Derivable f,Derivable g) ⇒ Derivable (f :+: g) where
  data Derivative (f :+: g) =
    CL (Derivative f) | CR (Derivative g)
instance (Derivable f,Derivable g) ⇒ Derivable (f :*: g) where
  data Derivative (f :*: g) =
    C1 (Derivative f) g | C2 f (Derivative g)
instance (Derivable a) ⇒ Derivable (Rec a) where
  data Derivative (Rec a) =  Recursive
```

Note that we throw away the type information at the recursive position!

Operations on Contexts

- Now we can define some functions for one-hole contexts which help us create navigation functions for our zipper
- The function fill takes a one-hole context and a value and inserts the value into the hole, yielding the original value.

```
class Fillable f where
  fill :: (Typeable a) ⇒ Derivative f → a → Maybe f

instance (Fillable f,Fillable g) ⇒ Fillable (f :*: g) where
  fill (C1 c r) v =  flip (:*:) r <$> fill c v
  fill (C2 l c) v =  (l :*:)      <$> fill c v

instance (Typeable a) ⇒ Fillable (Rec a) where
  fill CRec v =  Rec <$> cast v
```

- Note that we need casting and Typeable to make this function work!
- The use of cast here has a great impact on the design of the rest of the Zipper
- This is why the function return a Maybe

First

- Another important function is the first function
- This function takes a value and splits it into the leftmost value within this value and the corresponding context

```
class Firstable f where
  first :: (Zipper a) ⇒
    f → Maybe (a,Derivative f)
instance (Firstable f,Firstable g) ⇒ Firstable (f :∗: g) where
  first (l :∗: r) = mapSnd (flip C1 r) <$> first l
                  <|> mapSnd (C2 l)    <$> first r
instance (Typeable f) ⇒ Firstable (Rec f) where
  first (Rec v) = (λx → (x,Recursive)) <$> cast v
```

Casting & Ambiguous types

- The usage of casting poses us with a problem
- Suppose we write the following

$$id\ x = \ uncurry\ fill < \$ > first\ x$$

- This will produce an "ambiguous type" error
- This is because we use cast after another cast and the types in between cannot be inferred
- To solve this, we need explicit typing information
- This means the user will have provide typing annotations when some functions are invoked
- We employ several techniques to limit the amount of type annotations required

Context

- One of them is extending the context datatype with a type-level list, so the types of context values are maintained
- We also maintain type information on what the hole and root types are, just like the SYB zipper

```
data Loc hole root c =
  Loc {focus :: hole,context :: Context hole root c}
data Context hole root l where
  Empty :: Context hole hole Epsilon
  Push :: (Zipper parent) ⇒ Derivative (Rep parent)
                          → Context parent root cs
                          → Context hole root (parent :<: cs)
```

Up

- With or extended context, we can easily write the up function, which goes one item up in the context stack

$$
\begin{aligned}
&\mathbf{up} :: (\mathit{Zipper}\ h, \mathit{Zipper}\ h') \Rightarrow \mathit{Loc}\ h\ r\ (h' :<: c) \rightarrow \mathit{Loc}\ h'\ r\ c \\
&\mathbf{up}\ (\mathit{Loc}\ h\ (\mathit{Push}\ c\ cs)) = \\
&\quad \mathit{fromJust}\ \$\ (\lambda x \rightarrow \mathit{Loc}\ (\mathit{to}\ x)\ cs) < \$ > \mathit{fill}\ c\ h
\end{aligned}
$$

- Note that the type-level list ensures that we cannot go up in the empty context
- We can also be sure that the fill succeeds, because else our program wouldn't typecheck, thus we can use fromJust
- The user does not have to type this function explicitly, the type information is maintained in the context

Down

- The down function navigates down into the current value

$$
\begin{aligned}
&\textbf{down} :: (\mathit{Zipper}\ h, \mathit{Zipper}\ h') \Rightarrow \\
&\quad \mathit{Loc}\ h\ r\ c \rightarrow \mathit{Maybe}\ (\mathit{Loc}\ h'\ r\ (h :<: c)) \\
&\textbf{down}\ (\mathit{Loc}\ h\ cs) = \\
&\quad (\lambda(h', c) \rightarrow \mathit{Loc}\ h'\ (\mathit{Push}\ c\ cs)) <\$> \mathit{first}\ (\mathit{from}\ h)
\end{aligned}
$$

- The type of h' (the type of the hole we navigate into) cannot be known up front
- We need user-annotations for this to type-check
- We can avoid having to write the whole type signature

$$
\begin{aligned}
&\mathit{down}' :: (\mathit{Zipper}\ h, \mathit{Zipper}\ h') \Rightarrow \\
&\quad h' \rightarrow \mathit{Loc}\ h\ r\ c \rightarrow \mathit{Maybe}\ (\mathit{Loc}\ h'\ r\ (h :<: c)) \\
&\mathit{down}'\ \_\ (\mathit{Loc}\ h\ cs) = \textbf{down}
\end{aligned}
$$

- Here we have a phantom variable h' which is not used but is there to guide the type-checking

$$
\mathit{downInt} = \mathit{down}'\ (\bot :: \mathit{Int})
$$

Families

Instead of passing the phantom type in the usual way

$$\bot :: \mathit{Employee}$$

we use a nicer approach:

```
data Fam a where
  Dept     :: Fam Dept
  Employee :: Fam Employee
  Salary   :: Fam Salary
  Name     :: Fam Name

instance Family Fam
```

Families

```
class Family (f :: * → *)

down :: (Zipper h,Zipper h',Family f,Show (f h')) ⇒
   f h' → Loc h r c → ZipperR (Loc h' r (h :<: c))
down =  downL
```

Error messages

- Some of the operations on our zipper may fail
- For example, when tying to go down into a type which isn't there
- Instead of returning Nothing, we return an error message with information on what operation went wrong
- If the user uses the GADT typing system, additional typing information is given

Conclusion

- We have written a working Generic Zipper using Instant Generics
- We don't need type annotation on the get/set operations
- Type annotations on the navigation functions contribute to their expressiveness
  - I.e. you can express what type you want to navigate into

Future work

- Write Template Haskell code to generate Family instances
- Employ type classes/families to catch errors at compile time
  - (Alloy?)