Instant Zipper

Ruud Koot Bram Schuur

November 14, 2010

1 Introduction

In this paper we present an implementation of a generic zipper type implemented in Instant Generics.

2 Using the Instant Zipper

Before moving on the implementation details of our zipper we present an example of how it can be used to edit a heterogeneous datatype. We present a similar example as given by Adams for his Scrap Your Zippers, to make it clear what the differences of our approach are.

We start with a number of datatypes represeting a university departement and its employees:

```
type Salary   = Float
type Manager  = Employee
type Name     = String

data Dept = D Manager [Employee]
  deriving (Eq, Show, Typeable)

data Employee = E Name Salary
  deriving (Eq, Show, Typeable)
```

To define a zipper over these datatype we will need to:

1. Make the datatypes an instance of Typeable.

2. Derive the Instant Generics representation using Template Haskell:

   ```
   $ (deriveAll '' Dept)
   $ (deriveAll '' Employee)
   ```

3. Declare a Family for all the types we want to navigate into:

   ```
   data Fam a where
     Dept     :: Fam Dept
     Employee :: Fam Employee
     Salary   :: Fam Salary
     Name     :: Fam Name
     List     :: (Show a) ⇒ Fam a → Fam [a]
   deriving instance Show (Fam a)
   instance Family Fam
   ```

4. Finally, we instantiate the the datatypes as Zippers:

   ```
   instance Zipper Dept
   instance Zipper Employee
   ```

While we could easily have designed our zipper in such a way as to not require the Family instance, this approach will make our navigations more elegant in appearance. Furthermore, we believe most of this can be automatically derived using Template Haskell.

We can now give a concrete value for the Dept type:

```
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
```

and edit it using the zipper:

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

$\gg\!=$ *return* $\circ$ *up*
$\gg\!=$ *return* $\circ$ *up*
$\gg\!=$ *downR* (*List Employee*)
$\gg\!=$ *down* (*List Employee*)
$\gg\!=$ *down* (*List Employee*)
$\gg\!=$ *down Employee*
$\gg\!=$ *downR Salary*
$\gg\!=$ *return* $\circ$ *setHole* 100.0
$\gg\!=$ *return* $\circ$ *leave*

The diffence with SYZ is clear: we need to annotate our navigation function with types. Giving a wrong type can cause a failure (Nothing) at runtime, but can also be important when moving down along the spine of list, or, down into a value in a list. The getHole and setHole operations can now be statically typed however.

3 Implementing the Instant Zipper

A zipper consists of the part of the data structure which is currently *in focus*, together with the *context* in which it appears. Alternatively we can view the part of the data structure which is in focus as filling a *hole* in the context.

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

Because our zipper can navigate over *heterogeneous datatypes*, we have to use a few tricks to make the zipper work. This is why some parts of the datatypes are not filled in, they will be in the upcoming chapters.

The context is a stack of *one-hole contexts*:

```
data Context ... where
  Empty :: Context ...
  Push  :: (Zipper parent) ⇒
           Derivative (Rep parent)
        → Context ...
        → Context ...
```

As we move down into the datastructure we peel of the constructors (minus one hole) for the structure in focus and push it onto the context stack. The type of a datatype with one hole per constructor is given by the *derivative* of that datatype [2].

3.1 Derivatives

We store the derivative of a datatype in an associated datatype:

```
class Derivable f where
  data Derivative f :: ∗
```

Calculating the derivative in the sum-of-products view used by Instant Generics is straigforward, following the algebraic rules:

```
instance Derivable Int where
    data Derivative Int
instance Derivable U where
    data Derivative U
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
instance (Derivable a) ⇒
  Derivable (Var a) where
    data Derivative (Var a) = Variable
```

3.2 Navigation

In this section the most important functions of our zipper are presented along with the impact they have on the design and usability. First we will introduce 2 auxiliary functions fill' and first', with which are then used to build the navigation functions up, down and leave for our zipper.

3.2.1 Fill

In Instant Generics we create functions on representation types by creating a type class wrapping the function and creating an instance of this typeclass for each member of the representation type. An important function for zippers is the fill function, which receives a context and a value and puts

the value into the hole in the context, resulting in a new value.

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

Important here is the Typeable class-constraint for the value we want to plug in. The Derivative can have a hole of any type (see explaination of Derivative), and thus we can not guarantee, using the type system, that the type of the element we want to plug in is indeed the type of the hole. To overcome this problem we require both the hole and the item we want to plug in to be Typeable so we can use cast. This is also why the fill function returns a Maybe type, because casting may fail. Now the Fillable instance for Rec, which represents the position of the hole in the context, looks as follows:

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

Note that we give a similar instance for Var. Rec and Var are treated equally in our zipper library.

3.2.2 First

Another important function for our zipper is the first function. The first function takes a value and splits this value into its leftmost value and the corresponding context. It effectively punches a hole in a value. A similar problem as with the fill function arises here. We want to produce a hole with type a. Here again we use casting to achieve this.

```
class Firstable f where
  first :: (Zipper a) ⇒
    f → Maybe (a, Derivative f)
```

Notice that when this first' function is invoked with a type for a that is not present in one of the holes of Derivative f, the function will produce Nothing. The function will create a hole and context for the types that are given, where the hole is the leftmost hole with the specified type.

```
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
```

The product instance tries to create a result to the left first, if it fails to the right. If both fail no context can be given.

3.3 Ambiguous Type problem

Suppose we now would like to execute the following function:

```
id = uncurry fill <$> first
```

We first create a context and a hole and afterwards put the value back again. Intuitively this should not give any problem, but GHC will give us an "ambiguous type" error. What does this mean? This is the error that occurs when executing (show . read). GHC cannot infer the type between the read and show, because both of their behavious depends on this type. This same problem occurs in our example, but now with casting. GHC cannot infer the types between the casts. To solve this the user has to add more type information, so the type becomes known. The rest of the design of our zipper is built around limiting the amount of typing information the user has to type to solve this problem.

3.4 Updated Context

To limit the amount of typing information the user has to write, we maintain typing information in our context. We use 2 techniques. The first is a type-leve list, similar to [1], which we use to keep type information of the items in the context stack. We also keep type information on the root type of our tree and the type of the current hole. The result looks as

```
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)
```

$\rightarrow$ *Context parent root cs*
$\rightarrow$ *Context hole root* (*parent* :<: *cs*)

3.4.1 Up

The up function goes 1 element up in the context stack. Because we added type information to our context, this function now becomes simple, all the type information has been kept through are Context GADT. The function looks as follows:

```
up :: (Zipper h, Zipper h') ⇒
  Loc h r (h' :<: c) → Loc h' r c
up (Loc h (Push c cs)) =
  fromJust $ (λx → Loc (to x) cs) <$> fill' c h
```

Note that the additional type information also prevents us form going up in an empty context. We can also add fromJust because we can be sure the result will be correct.

3.4.2 Leave

The leave function applies up repeatedly until we get our original datatypes back, here we can use the fact that we stored the root type of our tree to give the function a result type.

```
leave :: (Zipper h) ⇒ Loc h r c → r
leave (Loc h Empty) = h
leave loc@(Loc _ (Push _ _)) = leave ∘ up $ loc
```

3.4.3 Down

The last important function that we need to implement is the down function, which goes down into the current hole, adding a context to the context stack and creates a new hole. We use the first' function to do this. In the previous up/leave functions we could avoid the need for type-annotations by chosing our datastructures cleverly and maintaining type information. With the down function there is no way for us to infer what type we want our new hole to have, so we need type-annotations from the user. The normal down function looks as follows:

```
down :: (Zipper h, Zipper h') ⇒
  Loc h r c → Maybe (Loc h' r (h :<: c))
down (Loc h cs) =
  (λ(h', c) → Loc h' (Push c cs)) <$> first (from h)
```

Specifically, the type of h' is the type which we cannot infer. This system is unworkable if we have to annotate each call to down with its full type signature. Thus we introduce a more convenient way for specifying the type of h', a phantom variable! This is a variable which is not used but only there for the extra type information, the function now looks like this:

```
down' :: (Zipper h, Zipper h') ⇒
  h' → Loc h r c → Maybe (Loc h' r (h :<: c))
down' _ (Loc h cs) = down
```

Filling in the argument with a static type is now sufficient, this function could be used in such a way:

```
downInt = down (⊥ :: Int)
```

Thus there are 2 ways to call the down function, with or without phantom argument type.

3.4.4 Other functions

The functions right and left, which move the hole right or left into a new hole of a new (specified) type, are implemented in the same manner as the down function. For these two functions we also need extra type annotations.

3.5 Tying the recursive knot

Having defined all operations on our we need to "tie the recursive knot" to create our zipper:

```
class (Representable f
      , Typeable     f
      , Fillable   (Rep f)
      , Firstable  (Rep f)
      , Nextable   (Rep f)
      , Lastable   (Rep f)
      , Prevable   (Rep f)) ⇒ Zipper f
instance Zipper Int
instance Zipper Char
instance Zipper Float
instance (Zipper a) ⇒ Zipper [a]
```

3.6 Error messages

Because we have no compile-time safety for the correctness of traversals within a value, traversals written in our zipper become hard to debug. When, for example, the user tries to go down into a type which isn't there, the zipper will just return a Nothing. To help the user in debugging traversals, we replaced the Maybe constructs in our navigation functions with the error monad. We also extended the functions so they report the operation at which something goes wrong. An example extension of the downL function looks as follows:

```
downL_ :: (Zipper h, Zipper h') ⇒
  Loc h r c → ZipperR (Loc h' r (h :<: c))
downL_ (Loc h cs) =
  maybe (Left "Error going down left")
    (λ(h', c) → Right (Loc h' (Push c cs)))
      $ first (from h)
```

The implementation is similar for the other navigation functions. Using the GADTs described below, we can also give information on the types with which the syste went wrong.

3.7 Tidying up with GADTs

Although the introduction of phantom variables greately reduces the burden of writing type information, the code still gets cluttered with a lot undefineds and ad-hoc type annotations. What we would like is a general solution to the problem of having to specify a type without a value.

We would prefer to simply pass a type as a parameter, but unfortunaly this is not allowed in Haskell. We can however use GADT's to emulate dependent types to extent necessary here. We declare a type class

```
class Family (f :: * → *)
```

containing no methods. We can now give (constructor) names to types using a GADT:

```
data Fam a where
  Char ::                        Fam Char
  Int  ::                        Fam Int
  Float ::                       Fam Float
  List  :: (Family f) ⇒ f a → Fam [a]
```

In the navigation functions we now expact a phantom variable of type

```
Family f ⇒ f h'
```

instead of h':

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

4 Future research

An important improvement would be to write TH functions for the Family GADTs, so these don't have to be written by hand. Another extension would be to use a technique similar to Alloy to catch more errors at compile time when going down into types that are not there.

5 Conclusion

The end result of our efforts to create a zipper in Instant Generics, is a zipper which lies somewhere between the syb and the multirec zipper. We use casting at runtime to determine what position to traverse into, but use the type-driven derivative functionality to create our zipper types.

The fact that we need type annotations may look like a burden at first, but with the Family GADT and other type-level tricks this burden is mostly lifted. The type annotations that remain actually add to the expressiveness of the functions. It is now possible to specify more precisely where to navigate to, and the typing info always gives debug information. Thus we end up with a very useful zipper which is inspired by the techniques of multirec and syb, and even a bit beyond.

References

[1] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[2] N. Ghani M. Abbot, T. Altenkirch and C. McBride. *Derivatives of containers*, volume 2701 of *Lecture Notes in Computer Science*. Springer Berlin, Heidelberg, 2003.