Zipper

Dariusz Bukowski

10.06.2015

Problem definition

In a purely functional setting we are unable to edit tree-shaped data efficiently. Everytime we want to modify a node we need to copy its path from the root of the tree.

Basic idea

We want to have a "pointer" to the node of interest. We will keep the value in this point and the context of this value, which will allow us to move around the tree structure.
There are many variations on the basic idea. First let us take a look at a version which pertains to trees with variadic arity anonymous tree nodes, and tree leaves injecting values from an unspecified item type.

Variadic arity

We assume a type parameter *item* of the elements we want to manipulate hierarchically. The tree structure is just hierarchical lists grouping trees in a section. For instance, in the UNIX file system, items would be files and sections would be directories:

```
type tree =
    Item of item
  | Section of tree list;;
```

Variadic arity

We assume a type parameter *item* of the elements we want to manipulate hierarchically. The tree structure is just hierarchical lists grouping trees in a section. For instance, in the UNIX file system, items would be files and sections would be directories:

```
type tree =
    Item of item
  | Section of tree list;;
```

The path in the tree would look as follows:

```
type path =
    Top
  | Node of tree list * path * tree list;;
```

Variadic arity

A *Node*(l, p, r) contains its the reversed list l of its left siblings, its father path p, and its list r of right siblings.

Variadic arity

A $Node(l, p, r)$ contains its the reversed list l of its left siblings, its father path p, and its list r of right siblings.
A location in the tree adresses a subtree, together with its path.

```
type location = Loc of tree * path;;
```

A location consists of a distinguished *tree*, the current focus of attention and its *path*, representing its surrounding context.

Example

Assume that we consider the parse tree of arithmetic expressions, with string items. The expression $a \times b + c \times d$ parses as the tree:

```
Section[Section[Item "a"; Item "*"; Item "b"];
        Item "+";
        Section[Item "c"; Item "*"; Item "d"]];;
```

Example

Assume that we consider the parse tree of arithmetic expressions, with string items. The expression $a \times b + c \times d$ parses as the tree:

```
Section[Section[Item "a"; Item "*"; Item "b"];
        Item "+";
        Section[Item "c"; Item "*"; Item "d"]];;
```

The location of the second multiplication sign in the tree is:

```
Loc(Item "*",
    Node([Item "c"],
         Node([Item "+"; Section [Item "a"; Item "*"; Item "b"]],
              Top,
              []),
         [Item "d"]))
```

Navigation

Navigation in such a structure is really simple:

```
let go_left (Loc(t,p)) = match p with
    Top -> failwith "left of top"
  | Node(l::left,up,right) -> Loc(l,Node(left,up,t::right))
  | Node([],up,right) -> failwith "left of first";;

let go_right (Loc(t,p)) = match p with
    Top -> failwith "right of top"
  | Node(left,up,r::right) -> Loc(r,Node(t::left,up,right))
  | _ -> failwith "right of last";;
```

Navigation

Going up and down the tree is slightly more complicated:

```
let go_up (Loc(t,p)) = match p with
    Top -> failwith "up of top"
  | Node(left,up,right) ->
                Loc(Section((rev left) @ (t::right)),up);;

let go_down (Loc(t,p)) = match t with
    Item(_) -> failwith "down of item"
  | Section(t1::trees) -> Loc(t1,Node([],p,trees))
  | _ -> failwith "down of empty";;
```

Note that all the navigation primitives take a constant time, except *go_up*, which is proportional to the length of the list of the left siblings of the current term.

Tree modification

As we can see on the previous slides, the navigation primitives are fast and easy. We may also want to mutate the structure at the current location:

```
let change (Loc(_,p)) t = Loc(t,p);;
```

Tree modification

Inserting new nodes:

```
let insert_right (Loc(t,p)) r = match p with
      Top -> failwith "insert of top"
    | Node(left,up,right) -> Loc(t,Node(left,up,r::right));;

let insert_left (Loc(t,p)) l = match p with
      Top -> failwith "insert of top"
    | Node(left,up,right) -> Loc(t,Node(l::left,up,right));;

let insert_down (Loc(t,p)) t1 = match t with
      Item(_) -> failwith "down of item"
    | Section(sons) -> Loc(t1,Node([],p,sons));;
```

Tree modification

We may also want to implement a deletion primitive. We may choose to move right, if possible, otherwise left, and up in case of an empty list.

```
let delete (Loc(_,p)) = match p with
     Top -> failwith "delete of top"
   | Node(left,up,r::right) -> Loc(r,Node(left,up,right))
   | Node(l::left,up,[]) -> Loc(l,Node(left,up,[]))
   | Node([],up,[]) -> Loc(Section[],up);;
```

First-order terms

So far, our structures are completely untyped – our tree nodes are not even labelled. If we want to implement a tree-manipulation editor for abstract-syntax trees, we have to label our tree nodes with operator names.

First-order terms

If we use items for this purpose, this suggests the usual LISP encoding of first-order terms: $F(T_1, \ldots, T_n)$ being coded as the tree $\mathit{Section}[\mathit{Item}(F); T_1; \ldots T_n]$.
This solution does not respect arity of operators.

First-order terms

We shall not pursue details of such generic variations any more, but rather consider how to adapt the idea to a specific given signature of operators given with their arities, in such a way that tree edition maintains well-formedness of the tree according to arities.
Basically, to each constructor F of the signature with arity n we associate n path operators $Node(F, i)$, with $1 \leq i \leq n$, each of arity n, used when going down the i-th subtree of an F-term. More precisely, $Node(F, i)$ has one path argument and $n - 1$ tree arguments holding the current siblings.

Example

Consider the abstract syntax tree for a hypothetical language:

```
data Term
= Var String
| Lambda String Term
| App Term Term
| If Term Term  Term
```

Example

A zipper for this type is a pairing of a *Term* that is the hole value with a *TermContext* that is the one-hole context. As stated before, for each recursive child of each constructor in *Term*, the *TermContext* type needs a constructor with that child missing and replaced by a reference to a parent context.

Example

```
type TermZipper = (Term, TermContext)
data TermContext
= RootTerm
| Lambda1 String TermContext
| App1 TermContext Term
| App2 Term TermContext
| If1 TermContext Term Term
| If2 Term TermContext Term
| If3 Term Term  TermContext
```

Movement operations for *Term*

We need to provide the movement operations for all constuctors of *TermContext*.

```
downTerm :: TermZipper -> Maybe TermZipper
downTerm (Var s, c) = Nothing
downTerm (Lambda s t1 , c) = Just (t1 , Lambda1 s c)
downTerm (App t1 t2 , c) = Just (t1 , App1 c t2 )
downTerm (If t1 t2 t3 , c) = Just (t1 , If1 c t2 t3 )

upTerm :: TermZipper -> Maybe TermZipper
upTerm (t1 , RootTerm ) = Nothing
upTerm (t1 , Lambda1 s c) = Just (Lambda s t1 , c)
upTerm (t1 , App1 c t2 ) = Just (App t1 t2 , c)
upTerm (t2 , App2 t1 c) = Just (App t1 t2 , c)
upTerm (t1 , If1 c t2 t3 ) = Just (If t1 t2 t3 , c)
upTerm (t2 , If2 t1 c t3 ) = Just (If t1 t2 t3 , c)
upTerm (t3 , If3 t1 t2 c) = Just (If t1 t2 t3 , c)
```

Movement operations for *Term*

```
leftTerm :: TermZipper -> Maybe TermZipper
leftTerm (t1 , RootTerm ) = Nothing
leftTerm (t1 , Lambda1 s c) = Nothing
leftTerm (t1 , App1 c t2 ) = Nothing
leftTerm (t2 , App2 t1 c) = Just (t1 , App1 c t2 )
leftTerm (t1 , If1 c t2 t3 ) = Nothing
leftTerm (t2 , If2 t1 c t3 ) = Just (t1 , If1 c t2 t3 )
leftTerm (t3 , If3 t1 t2 c) = Just (t2 , If2 t1 c t3 )

rightTerm :: TermZipper -> Maybe TermZipper
rightTerm (t1 , RootTerm ) = Nothing
rightTerm (t1 , Lambda1 s c) = Nothing
rightTerm (t1 , App1 c t2 ) = Just (t2 , App2 t1 c)
rightTerm (t2 , App2 t1 c) = Nothing
rightTerm (t1 , If1 c t2 t3 ) = Just (t2 , If2 t1 c t3 )
rightTerm (t2 , If2 t1 c t3 ) = Just (t3 , If3 t1 t2 c)
rightTerm (t3 , If3 t1 t2 c) = Nothing
```

Non-movement operations for *Term*

```
fromZipperTerm :: TermZipper -> Term
fromZipperTerm z = f z where
f :: TermZipper -> Term
f (t1 , RootTerm ) = t1
f (t1 , Lambda1 s c) = f (Lambda s t1 , c)
f (t1 , App1 c t2 ) = f (App t1 t2 , c)
f (t2 , App2 t1 c) = f (App t1 t2 , c)
f (t1 , If1 c t2 t3 ) = f (If t1 t2 t3 , c)
f (t2 , If2 t1 c t3 ) = f (If t1 t2 t3 , c)
f (t3 , If3 t1 t2 c) = f (If t1 t2 t3 , c)

toZipperTerm :: Term -> TermZipper
toZipperTerm t = (t, RootTerm )

getHoleTerm :: TermZipper -> Term
getHoleTerm (t, _) = t

setHoleTerm :: Term -> TermZipper -> TermZipper
setHoleTerm h (_, c) = (h, c)
```

Non-movement operations for *Term*

With the exception of *fromZipperTerm*, these are trivial wrapper functions manipulating the pair that forms a zipper. The *fromZipperTerm* function moves all the way to the root context before returning the resulting value.

Limitations

All the operations are really simple. However, we have two major issues:

- traditional zipper is bound to a single type. For example we can't define a zipper for this type representing a department:

```
data Dept = D Manager [Employee]
deriving (Show, Typeable, Data)
data Employee = E Name Salary
deriving (Show, Typeable, Data)
type Salary = Float
type Manager = Employee
type Name =  String
```

- for each new type the traditional zipper requires a complete rewrite of the boilerplate code from previous slides

Generic Zipper

We want to address those limitations. We want a generic zipper that operates on nonhomogeneous types like *Dept* and *Employee* just as well as on homogeneous types like *Term*. Finally, we would like the generic zipper to require no boilerplate code on the user's part.

Generic Zipper

We want to address those limitations. We want a generic zipper that operates on nonhomogeneous types like *Dept* and *Employee* just as well as on homogeneous types like *Term*. Finally, we would like the generic zipper to require no boilerplate code on the user's part.
We will take a look at the Haskell library from the HackageDB repository at http://hackage.haskell.org/package/syz (*Scrap your zipper*)

Generic Zipper

The only restriction is that the type that the zipper traverses must be an instance of the *Data* class. The *Data* class is provided by the standard libraries packaged with GHC and GHC can automatically derive instances of *Data* for user defined types.

Generic Zipper Interface

Injection and Projection

```
toZipper :: (Data a) => a -> Zipper a
fromZipper :: Zipper a -> a
```

Movement

```
up :: Zipper a -> Maybe (Zipper a)
down :: Zipper a -> Maybe (Zipper a)
left :: Zipper a -> Maybe (Zipper a)
right :: Zipper a -> Maybe (Zipper a)
```

Hole manipulation

```
query :: GenericQ b -> Zipper a -> b
trans :: GenericT -> Zipper a -> Zipper a
transM :: (Monad m) => GenericM m -> Zipper a -> m (Zipper a)
```

Using generic zipper

```
dept :: Dept
dept = D agamemnon [menelaus, achilles, odysseus]

agamemnon, menelaus, achilles, odysseus :: Employee
agamemnon = E "Agamemnon" 5000
menelaus = E "Menelaus" 3000
achilles = E "Achilles" 2000
odysseus = E "Odysseus" 2000
```

Using generic zipper

```
dept :: Dept
dept = D agamemnon [menelaus, achilles, odysseus]

agamemnon, menelaus, achilles, odysseus :: Employee
agamemnon = E "Agamemnon" 5000
menelaus = E "Menelaus" 3000
achilles = E "Achilles" 2000
odysseus = E "Odysseus"  2000
```

```
*Main> let g1 = toZipper dept
*Main> :type g1

g1 :: Zipper Dept
```

Getting Hole Value

With a traditional zipper, the type of the hole is fixed. For example, *getHoleTerm* always returns a *Term*. But with a generic zipper, the type of the hole changes as the focus of the zipper moves around.
The compiler knows only the type of the zipper (for example *Dept*), which doesn't expose the type of the hole.

Getting Hole Value

This is resolved by the hole-manipulation functions, *query*, *trans*, and *transM*.

```
type GenericQ r = forall a. (Data a) => a -> r
type GenericT = forall a. (Data a) => a -> a
type GenericM m = forall a. (Data a) => a -> m a
```

Each is defined in terms of a user-supplied generic function that operates on any argument type (e.g., the universally quantified *a*) provided the type is an instance of the *Data* class.

Getting Hole Value

To retrieve the contents of the hole, we supply to *query* a generic query function of type $\mathit{forall}\ a.(\mathit{Data}\ a) => a \rightarrow r$.

Getting Hole Value

To retrieve the contents of the hole, we supply to *query* a generic query function of type $\mathit{forall}\ a.(\mathit{Data}\ a) => a \rightarrow r$. We will use type-safe *cast* function:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

Typeable class is a superclass of the *Data* class.

Getting Hole Value

As expected the hole contains the original object:

```
*Main> query cast g1 :: Maybe Dept
Just (D (E "Agamemnon" 5000.0)
[E "Menelaus" 3000.0,
E "Achilles" 2000.0,
E "Odysseus" 2000.0])
```

Since in this example we will be retrieving the contents of the hole several times, we might want to define a helper for it:

```
getHole :: (Typeable b) => Zipper a -> Maybe b
getHole = query cast
```

Getting Hole Value

None of the core generic zipper functions involve any casts. It is up to the user when a cast is used.

Setting Hole Value

Let’s say we want to change the name of the *Manager* from *Agamemnon* to *King Agamemnon*.

Setting Hole Value

Let's say we want to change the name of the *Manager* from *Agamemnon* to *King Agamemnon*.
To change the king's title, the zipper must navigate to the proper position. The first step is to move down:

```
*Main> let Just g2 = down g1
*Main> getHole g2 :: Maybe [Employee]
Just [E "Menelaus" 3000.0,
E "Achilles" 2000.0,
E "Odysseus"  2000.0]
```

Setting Hole Value

The zipper descends to the right-most child instead of the leftmost child. The generic zipper's *down* function always does this for reasons that are explained later in the implementation section. For now this means that Agamemnon's record is the left sibling of where the zipper is currently focused, so the next thing to do is move left:

```
*Main> let Just g3 = left g2
*Main> getHole g3 :: Maybe Employee
Just (E "Agamemnon" 5000.0)
```

Setting Hole Value

Now the current hole is Agamemnon's *Employee* record, and there is one [*Employee*] sibling to the right. Moving down once more and to the left will get us to the *Name* in his record:

```
*Main> let Just g4 = down g3
*Main> getHole g4 :: Maybe Salary
Just 5000.0
*Main> let Just g5 = left g4
*Main> getHole g5 :: Maybe Name
Just "Agamemnon"
```

Setting Hole Value

We came to the right position in the tree. All we have to do now is set the new value. To do it we will use the *trans* function, which applies a generic transformer to the hole.

Setting Hole Value

We came to the right position in the tree. All we have to do now is set the new value. To do it we will use the *trans* function, which applies a generic transformer to the hole.
This time we will use *mkT* function:

```
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
```

It takes as an argument a function that transforms one type of object and lifts that function to be a generic transformer for any type of object.

Setting Hole Value

Like before with *getHole*, we can implement the helper function *setHole* with *mkT*:

```
setHole :: (Typeable a) => a -> Zipper b -> Zipper b
setHole h z = trans (mkT (const h))  z
```

This function leaves the hole unchanged if it is not of type a.

Setting Hole Value

Now that we provided a function to set the hole value we can use it to change the *Name* of the *Manager*.

```
*Main> let g6 = setHole "King Agamemnon" g5
```

Since we are already here lets also give him a raise:

```
*Main> let Just g7 = right g6
*Main> let g8 = setHole (8000.0 :: Float) g7
```

Setting Hole Value

Now that we provided a function to set the hole value we can use it to change the *Name* of the *Manager*.

```
*Main> let g6 = setHole "King Agamemnon" g5
```

Since we are already here lets also give him a raise:

```
*Main> let Just g7 = right g6
*Main> let g8 = setHole (8000.0 :: Float) g7
```

If we traverse up the zipper, we can verify that the changes we made had the proper effect:

```
*Main> let Just g9 = up g8
*Main> getHole g9 :: Maybe Employee
Just (E "King Agamemnon" 8000.0)
```

Generic Zipper

- there are no type casts or dynamic type checks except those that are part of the user-supplied generic functions
- at worst, the movement operations may fail by returning *Nothing* when the user tries an illegal movement (like in traditional zipper). Of course we can instead ignore such movements.

Implementation

Just as with the traditional zipper, the generic zipper is made up of a hole and a context. However, while the type of the hole is fixed in a traditional zipper, in a generic zipper it may change as the focus moves.
Thus we must construct a type that expresses this variability in a type-safe way. This is done by the *Zipper* type. It contains an existentially quantified hole and a context that matches both the hole and the zipper's root type:

```
data Zipper root =
forall hole. (Data hole) => Zipper hole (Context hole root)
```

Implementation

As with a traditional zipper, the *Context* type keeps track of the siblings and parents of the current hole and ensures that they are of appropriate types. From a high-level perspective, a *Context* represents a one-hole context that contains a hole of type *hole* and a top-most node of type *root*. Except when it is the top-most context represented by *NullCtxt*, it contains a set of left siblings, a set of right siblings, and its parent context:

```
data Context hole root where
NullCtxt :: Context a a
ConsCtxt :: Left (???) -> Right (???) -> Context (???)
-> Context hole root
```

Implementation

As with a traditional zipper, the *Context* type keeps track of the siblings and parents of the current hole and ensures that they are of appropriate types. From a high-level perspective, a *Context* represents a one-hole context that contains a hole of type *hole* and a top-most node of type *root*. Except when it is the top-most context represented by *NullCtxt*, it contains a set of left siblings, a set of right siblings, and its parent context:

```
data Context hole root where
NullCtxt :: Context a a
ConsCtxt :: Left (???) -> Right (???) -> Context (???)
-> Context hole root
```

How should we fill the blanks?

Implementation

Lets first take a look at the *Left* and *Right* siblings types.

```
data Left expects
= LeftUnit expects
| forall b. (Data b) => LeftCons (Left (b -> expects)) b
```

The first argument of *ConsLeft* is a *Left* that represents a partially applied constructor of type $b \rightarrow expects$. This is packaged up with a second argument of type *b*. This packaging represents the application of the former to the latter to construct an object of type *expects*.
Multiple virtual applications are chained together to supply each of the arguments for a multi-argument constructor. The base case for this is a raw constructor that is not applied to anything and is represented with *UnitLeft*.

Implementation

Lets take a look at the example. Suppose for the moment that we want to use *Left* to represent constructor applications for the type *Foo*:

```
data Foo = Bar Int Char | Baz Float
```

To build a *Foo* object we start with the *Bar* constructor. This is represented by the value *UnitLeft Bar*. The type of this value is:

```
UnitLeft Bar :: Left (Int -> Char -> Foo)
```

Implementation

The arguments that *Bar* is expecting are manifest in the type of the resulting *Left* object. We can add those arguments with *ConsLeft*, and the way *ConsLeft* is defined ensures that those arguments are of the proper type.

```
*Main> :type UnitLeft Bar
'ConsLeft ' 1
it :: Left (Char -> Foo)
*Main> :type UnitLeft Bar
'ConsLeft ' 1
'ConsLeft ' 'a'
it :: Left Foo

*Main> :type UnitLeft Baz
it :: Left (Float -> Foo)
*Main> :type UnitLeft Baz
'ConsLeft ' 1.0
it :: Left  Foo
```

Left contains a value of existentially quantified type *b* provided *b* matches the argument type expected by the constructor.

Implementation

The representation of right siblings is very similar to that of left siblings. The major difference is that instead of encoding what children the partial constructor application expects, the type needs to encode what children it provides.

Implementation

```
data Right provides parent where
NullRight :: Right parent parent
ConsRight :: (Data b) => b -> Right a t -> Right (b -> a) t
```

The *NullRight* constructor represents when there are no siblings to the right of the current hole. When there are siblings to the right, they are represented with *ConsRight*. The parent parameter to this type is used later when we combine *Left* and *Right* into a *Context*, where it ensures that context types properly match.

Implementation

Let's again take a look at the example.
Consider a *Right* that represents right siblings to be fed to the *Bar* constructor. Every *Right* starts off with a *NullRight*:

```
*Main> :type NullRight
it :: Right parent parent
```

A *Right* stores its values starting with the rightmost, so the first value stored must have the type of the last argument to *Bar*, namely *Char*:

```
*Main> :type ConsRight 'a' NullRight
it :: Right (Char -> a) a
```

Next the preceding argument to *Bar* is added:

```
*Main> :type ConsRight 1 (ConsRight 'a' NullRight )
it :: Right (Int -> Char -> a) a
```

Implementation

Except for the universally quantified *a*, the provides type parameter of the resulting *Right* now matches the type of the *Bar* constructor (i.e., $Int \to Char \to Foo$). This encodes the fact that the *Right* object provides values that match what *Bar* expects as arguments. The universal quantification of type *a* is a bit worrisome, but we will handle it during combining the siblings.

Implementation

With a *Left* and *Right* object of the appropriate types we should be able to reconstruct the object that they represent by first performing the applications represented in the *Left* and then applying the result to the arguments stored in the *Right*. A *Left* and *Right* are of appropriate types when the *expects* type parameter of the *Left* equals the *provides* parameter of the *Right*.

Implementation

The *fromLeft* helper function does the applications that are represented by a *Left*, and the *fromRight* helper function applies a function to the values stored in a *Right*.

```
combine :: Left (hole -> rights) -> hole
-> Right rights parent -> parent
combine lefts hole rights =
fromRight ((fromLeft lefts) hole) rights

fromLeft :: Left r -> r
fromLeft (UnitLeft a) = a
fromLeft (ConsLeft f b) = fromLeft f b

fromRight :: r -> Right r parent -> parent
fromRight f (NullRight ) = f
fromRight f (ConsRight b r) = fromRight (f b)  r
```

The *combine* function first uses *fromLeft* to apply all the values stored in the lefts. Then it applies the result to *hole*. Lastly that result is applied to the values stored in the rights using *fromRight*.

Implementation

Given the way *Right* is defined, the *parent* parameter always is a suffix of the *provides* parameter. Furthermore, in a call to *combine*, the *provides* parameter of the *Right* is *rights*, which is part of the expects parameter of the *Left*.
This eliminates the problem with *parent* being universally quantified seen in the *Right* example, and along with the matching of the *expects* and *provides* type parameters, serves a key role in the implementation of *Context*.

Implementation

Now we are ready to fill the blanks. Given a matching *Left* and *Right*, the only part missing is the parent context. That parent context must have a hole that matches the type that is constructed from the *Left* and *Right* siblings, i.e., the *parent* parameter of *Right*. The full definitionof *Context* is as follows, where both *rights* and *parent* are existentially quantified:

```
data Context hole root where
CtxtNull :: Context a a
CtxtCons ::
forall rights parent. (Data parent) =>
Left (hole -> rights)
-> Right rights parent
-> Context parent root
-> Context hole  root
```

Implementation

Now that types are defined, implementing movement for generic zipper is easy:

```
left (Zipper _ NullCtxt ) = Nothing
left (Zipper _ (ConsCtxt (UnitLeft _) _ _)) = Nothing
left (Zipper h (ConsCtxt (ConsLeft l h') r c)) =
Just (Zipper h' (ConsCtxt l (ConsRight h r) c))

right (Zipper _ NullCtxt ) = Nothing
right (Zipper _ (ConsCtxt _ NullRight _)) = Nothing
right (Zipper h (ConsCtxt l (ConsRight h' r) c)) =
Just (Zipper h' (ConsCtxt (ConsLeft l h) r c))

up (Zipper _ NullCtxt ) = Nothing
up (Zipper hole (ConsCtxt l r ctxt)) =
Just (Zipper (combine l hole r) ctxt)
```

Non-movement operations

```
fromZipper (Zipper hole NullCtxt ) = hole
fromZipper (Zipper hole (ConsCtxt l r ctxt)) =
fromZipper (Zipper (combine l hole r) ctxt)

toZipper x = Zipper x NullCtxt

query f (Zipper hole ctxt) = f hole

trans f (Zipper hole ctxt) = Zipper (f hole) ctxt

transM f (Zipper hole ctxt) = do
hole' <- f hole
return (Zipper hole' ctxt)
```

Implementing down

With a single exception, *down*, all of the core zipper operations simply shuffle the constructors of a zipper around.
When moving down, the values of *Left*, *Right* and *Context* do not yet exist - they must be built by deconstructing the hole. We will use the *gfoldl* for that.

Implementing down

The *gfoldl* function is defined so that the call *gfoldl f k a* deconstructs the object *a*, applies *k* to the extracted constructor of *a*, and then reapplies each of the constructor's arguments using *f*.
For example the call:

```
*Main> gfoldl f k (Bar 5 'd')
```

deconstructs *Bar* 5 *′d′* into three parts: *Bar*, 5, and *′d′*. The *k* function wraps around *Bar*, and then *f* reapplies 5 and *′d′*. The end result is that our gfoldl call is equivalent to:

```
*Main> ((k Bar) 'f' 5) 'f' 'd '
```

Implementing down

The generic zipper uses *gfoldl* to implement the *toLeft* helper, which deconstructs a value into a set of left siblings:

```
toLeft :: (Data a) => a -> Left a
toLeft a = gfoldl ConsLeft UnitLeft  a
```

For example *toLeft* (*Bar* 5 $'d'$) results in the value:

```
(UnitLeft Bar) 'ConsLeft ' 5 'ConsLeft' 'd '
```

Implementing down

The *down* function is implemented by injecting the hole into a *Left* with *toLeft* and extracting its rightmost element. This rightmost element (if there is one) becomes the new hole, and the remaining elements become the left siblings:

```
down (Zipper hole ctxt) =
case toLeft hole of
UnitLeft _ -> Nothing
ConsLeft l hole' ->
Just (Zipper hole' (ConsCtxt l NullRight ctxt))
```

Implementing down

There are two drawbacks of using *gfoldl*:

- *Data* class constraints in *Context*, *Left* and *Right* types.
- *gfoldl* is a left fold, so the outermost *ConsLeft* constructor that comes out of *toLeft* contains the rightmost child. This means that the simplest implementation of *down* always moves to the rightmost child.

Bibliography

- Gérard Huet: *Functional Pearl. The Zipper*
- Michael D. Adams: *Scrap Your Zippers. A Generic Zipper for Heterogeneous Types*

Thank you for your attention!

Questions?