Functional Pearl: Scrap Your Zippers

Michael D. Adams

Indiana University

adamsmd@cs.indiana.edu

Abstract

The “zipper” data type provides the ability for editing tree shaped data in a pure functional setting and has found many uses and applications. However the traditional zipper has two major limitations. First, requires a significant amount of boilerplate code to implement. Second, it can only operate on homogeneous data types. Data structures where there are multiple node types are beyond the range of what it can handle.

The generic zipper developed in this paper solves both these issues while maintaining type safety. It does this by encoding the path to the current position in the type of the zipper and by keeping an abstract representation of the object being traversed The techniques used develop the generic zipper also prove to have uses for other problems which will be briefly explored.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data*]: Data Structures

Keywords Zipper, boilerplate, generalized abstract data types

1. Introduction

The data structure known as the “zipper” [4] for provides the ability to edit tree shaped data in a purely functional setting. It has been used to implement filesystems [5] and even window managers [12]. In fact, any situation where there is a well defined focal point for edits the zipper may find applicability. In a text-editor this focal point is manifest as the cursor. In a filesystem it is manifest as the current working directory, and in a window manager as the window with focus.

Perhaps even more important than the broad applicability of the zipper is the fact that creating a zipper for a specific data type is a very straightforward, mechanical process [1, 9] which if done properly ensures correctness by construction for all editing operations.

However, the traditional zipper suffers from two major issues. The first is that it is limited to operating over homogeneous data types. That is, each node in the tree on which the zipper is operating must be of the same type. Thus, while the zipper may work wonderfully at making edits to the abstract syntax tree of a lambda-calculus interpreter, it provides little help when it comes time to edit the more complex abstract syntax trees that would be necessary to implement languages that support things like types and statements in addition to simple expressions.

The second issue is the same problem that any piece of code written in a mechanical way has. It shouldn’t have to be written in the first place! At least not by a human. Boilerplate code can be monotonous to write and hard to maintain as the data type it is for evolves. A tool could be written to generate the code implementing the zipper from the data type, but it would be better if we could avoid adding this sort of meta-level to our code.

The techniques described in the “Scrap your boilerplate” papers [6, 7, 8] are designed to solve precisely these sort of issues, but these techniques are not a perfect fit. They are designed with all-at-once traversals in mind, but the zipper by its very nature is an incremental data structure. Excessive use of the `cast` operator from those papers can arrive at a solution of sorts, but at a steep cost. Depending on whether the `cast` was to the correct type, it might fail. So every piece of code that contains a `cast` might also possibly fail.

The “generic zipper” presented in this paper overcomes all these issues. It operates over any data type regardless of whether it is homogeneous or not. The only proviso is that it must be an instance of the `Data` class. Further, all of the standard zipper operations (`get_value`, `set_value`, `move_left`, `move_right`, `move_up`) with the exception of `move_down` are total functions. They will never fail.

The generic zipper achieves this by putting information about its position into its type. Generalized Algebraic Data Types (GADTs) [11, 10] make this possible. Nevertheless, even without GADTs the techniques developed for the generic zipper prove to have independent value for implementing other, similar data structures.

The remainder of this paper is divided as follows. Section reviews 2 the traditional implementation of the zipper. Sections 3 and 4 respectively present the use and implementation of the generic zipper. Section 5 briefly covers how the techniques developed in this paper may be applied to problems beyond the zipper, and finally section 6 concludes.

2. Using the Traditional Zippers

The zipper is made up of two parts: a hole and a context. The hole is the portion of the object that is rooted at the current position of the zipper within the overall object. The context contains the overall object but with the hole missing. It also implicitly contains the path from the hole to the root of the overall object using pointer reversal.

To see how this works for the traditional zipper we will follow the development in [1] before moving on to the main topic of this paper, the generic zipper. The following is an abstract syntax tree for a hypothetical language.

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

```
  | If Term Term Term
```

To define a zipper for this type, a `TermContext` type needs to be defined. For each constructor in `Term` and each recursive child component of that constructor, the `TermContext` type needs to have a constructor which allows that child component to be missing. For example, in a one-hole context an `If` constructor could be missing either its first, second or third child, and `App` could be missing either of its two children. Since with the traditional zipper we are forced to operate over homogeneous types. The `String` argument to `Lambda` can't be considered a child, so `Lambda` can only be missing its second argument. Finally, `Var` has no `Term` children so it can't contain a hole.

```
data TermContext
  = TermRoot
  | Lambda_1 String TermContext
  | App_1 TermContext Term
  | App_2 Term TermContext
  | If_1 TermContext Term Term
  | If_2 Term TermContext Term
  | If_3 Term Term TermContext
```

In place of each of these holes the constructors of `TermContext` points to the context parent of the current context which in turn points its own parent and so on until the root of the object is reached with `TermRoot`.

The declaration for for `TermZipper` is then:

```
type TermZipper = (Term, TermContext)
```

Moving down a `TermZipper` is implemented by pulling apart the current hole, extracting the first child and extending the current context with the children other than the first child. This is implemented by `term_down`.

```
term_down (Var s, c) = error "can't go down"
term_down (Lambda s t1, c) = (t1, Lambda_1 s c)
term_down (App t1 t2, c) = (t1, App_1 c t2)
term_down (If t1 t2 t3, c) = (t1, If_1 c t2 t3)
```

Moving up the zipper is simply the reverse of that process. The siblings of the current hole get combined with the current hole to form a new hole and the parent context becomes the current context. The portion of `term_up` dealing with `If` contexts is shown here. A full implementation would have a case for each constructor in `TermContext`.

```
term_up (t1, If_1 c t2 t3) = (If t1 t2 t3, c)
term_up (t2, If_2 t1 c t3) = (If t1 t2 t3, c)
term_up (t3, If_3 t1 t2 c) = (If t1 t2 t3, c)
```

Moving left and right in a zipper are both very similar to each other. They each take the current hole and replace it with the sibling immediately to either the left or the right. Again, for the sake of brevity only parts of `move_left` and `move_right` dealing with `If` terms are shown here:

```
term_left (t1, If_1 c t2 t3) = error "bad left"
term_left (t2, If_2 t1 c t3) = (t1, If_1 c t2 t3)
term_left (t3, If_3 t1 t2 c) = (t2, If_2 t1 c t3)

term_right (t1, If_1 c t2 t3) = (t2, If_2 t1 c t3)
term_right (t2, If_2 t1 c t3) = (t3, If_3 t1 t2 c)
term_right (t3, If_3 t1 t2 c) = error "bad right"
```

All that remains are the three functions `term_begin` (which constructs and initial zipper), `term_get` (which gets the value of the current hole), and `term_set` (which sets the value of the current hole):

```
term_begin t = (t, TermRoot)
term_get (t, _) = t
term_set h (_, c) = h
```

To see the zipper used in practice, consider the following hypothetical definition for the body of a factorial implementation.

```
fac = Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
      (Var "1")
      (App (App (Var "+") (Var "n"))
           (App (Var "fac")
                (App (Var "pred") (Var "n")))))
```

Notice that this definition contains a bug. The `+` operator was incorrectly used instead of `*` in this definition. We can use the zipper as shown in the following interaction to fix this.

```
*Main> let t0 = term_begin fac
*Main> term_get t0

Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
      (Var "1")
      (App (App (Var "+") (Var "n"))
           (App (Var "fac")
                (App (Var "pred") (Var "n")))))

*Main> let t1 = term_down t1
*Main> term_get t1

(If (App (App (Var "=") (Var "n")) (Var "0"))
    (Var "1")
    (App (App (Var "+") (Var "n"))
         (App (Var "fac")
              (App (Var "pred") (Var "n")))))

*Main> let t2 = term_down t1
*Main> term_get t2

(App (App (Var "=") (Var "n")) (Var "0"))

*Main> let t3 = term_right t2
*Main> term_get t3

(Var "1")

*Main> let t4 = term_right t3
*Main> term_get t4

(App (App (Var "+") (Var "n"))
     (App (Var "fac")
          (App (Var "pred") (Var "n")))))

*Main> let t5 = term_down t4
*Main> term_get t5

(App (Var "+") (Var "n"))

*Main> let t6 = term_down t5
*Main> term_get t6

(Var "+")

*Main> let t7 = term_set (Var "*") t6
*Main> let t8 = term_up t7
*Main> term_get t8
```

```
(App (Var "*") (Var "n"))
```

3. Using the Generic Zipper

While the traditional zipper works fine for homogeneous types like `Term`, it runs into problems for more complex types Consider for example a data type to represent 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
```

Now we have to separate data types, `Dept` and `Employee`, that we would like to traverse over instead of just the one type that `Term` had. This particular case is fairly simple, but situations like this happen fairly often. For example, a compiler may have an abstract syntax tree that represents statements and types in addition to expressions. The traditional zipper has no way to handle this. The generic zipper on the other hand will work just fine for this as well as any other generic data type. On top of this, it requires no boilerplate code on the user's part. Instead of writing `down`, `left`, `right` and `up` functions, the data type only needs to derive from the `Data` type class which is provided as part of GHC's libraries.

Here is a small example department:

```
company :: Dept
company =
  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
```

Now suppose Agamemnon decides that his employee record should really refer to him as ”King” Agamemnon. We want to edit `company`, so we initialize a generic zipper with `begin_zipper`.

```
*Main> let g1 = begin_zipper company
*Main> :type g1

  g1 :: Zipper (Up (Top, Dept, Top) Top)
```

The type of the zipper encodes the types of the objects in the current path and the types of the current hole's siblings. The `Top` type serves as a terminator for this encoding. In this case the type indicates that there are that no siblings to the left (the first `Top`), and that the current hole is a `Dept`. Also there are no siblings to the right (the second `Top`), and the zipper is at the root of the object thus having no parents (the third `Top`).

The contents of this zipper can be retrieved with `get_hole`. Since the zipper is still in its initialized state, the hole has the value of the original object.

```
*Main> get_hole g1

D (E "Agamemnon" 5000.0)
  [E "Menelaus" 3000.0,
   E "Achilles" 2000.0,
   E "Odysseus" 2000.0]
```

The generic zipper contains information about the current path, but it doesn't have any information about the types of the current hole's children. After all, that depends on which constructor is in the current hole.

This means that in order to move down the structure, we have to supply that extra type information. The zipper will verify whether that information is correct (using the generic type-safe `cast` operator) and return `Just` if it was correct. Otherwise it will return `Nothing`. If the user passes the constructor of the current hole to `move_down'`, it can infer the needed parts of the type from the type of the constructor. (The full version of `move_down` allows the user to specify the expected type explicitly, but since the type of the path can get complicated rather quickly, the `move_down'` wrapper function is often easier to use.)

```
*Main> isJust (move_down' g1 D)

True

*Main> let Just g2 = move_down' g1 D
*Main> :type g2

g2 :: Zipper
  (Up (Top -> Employee, [Employee], Dept)
  (Up (Top, Dept, Top)
  Top))

*Main> get_hole g2

[E "Menelaus" 3000.0,
 E "Achilles" 2000.0,
 E "Odysseus" 2000.0]
```

The type of `g2` in this example indicates that the current hole is a `[Employee]`, there is one `Employee` sibling to the left and none to the right, and that the parent is a `Dept`. Note that the generic zipper has descended to the right-most child as opposed to the traditional left-most child. This makes the internal implementation easier, but the upshot of this is that Agamemnon's record is the current zipper's left sibling. The next thing we have to do is `move_left`.

```
*Main> let g3 = move_left g2
*Main> :type g3

g3 :: Zipper
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top))

*Main> get_hole g3

  E "Agamemnon" 5000.0
```

Now the current hole is Agamemnon's `Employee` record and there is one `[Employee]` sibling to the right. Moving down once more and moving to the left will get us to the `Name` part of his record.

```
*Main> let Just g4 = move_down' g3 E
*Main> :type g4

g4 :: Zipper
  (Up (Top -> [Char], Float, Employee)
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top)))
```

```
*Main> get_hole g4

  5000.0

*Main> let g5 = move_left g4
*Main> :type g5

g5 :: Zipper
  (Up (Top, [Char], Float -> Employee)
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top)))

*Main> get_hole g5

  "Agamemnon"
```

We can change the value of the current hole with `set_hole`. While we're at it, let's also move to the right and give the king a raise.

```
*Main> let g6 = set_hole "King Agamemnon" g5
*Main> let g7 = move_right g6
*Main> let g8 = set_hole 8000 g7
```

Every one of these operations is completely type-safe, and with the exception of `move_down` none of them have any failure modes. Compare this with the traditional zipper where moving to far to the left or right could throw an error. The generic zipper can prevent this from happening because it has in its type signature the needed information about how many and what type of siblings there are. Attempting to move too far, results in a type error at compile time.

```
*Main> :type move_right g8

<interactive>:1:11:
Couldn't match expected type ‘h_new -> r’
       against inferred type ‘Employee’
  Expected type: Zipper
    (Up (Top -> [Char], Float, h_new -> r) up)
  Inferred type: Zipper
    (Up (Top -> [Char], Float, Employee)
    (Up (Top, Employee, [Employee] -> Dept)
    (Up (Top, Dept, Top)
    Top)))
In the first argument of ‘move_right’, namely ‘g8’
```

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

```
*Main> let g9 = move_up g8
*Main> :t g9

g9 :: Zipper
  (Up (Top, Employee, [Employee] -> Dept)
  (Up (Top, Dept, Top)
  Top))

*Main> get_hole g9

  E "King Agamemnon" 8000.0
```

Finally, by moving up once more we can retrieve the now modified root object.

```
*Main> let g10 = move_up g9
*Main> :type g10

  g10 :: Zipper (Up (Top, Dept, Top) Top)

*Main> get_hole g10

D (E "King Agamemnon" 8000.0)
  [E "Menelaus" 3000.0,
   E "Achilles" 2000.0,
   E "Odysseus" 2000.0]
```

4. Generic Zippers

Just as with the traditional zipper, the generic zipper is made up of a hole and a context. However, with the generic zipper as it moves about within an object, the type of the hole may change. Thus we must construct a type that is able to contain this variability in a type safe manner.

This is done by the `Zipper` GADT. it is almost trivial leaving the hard work to the `Context` type. The only part it has to ensure is that the value it contains for the hole matches the hole in the context.

```
data Zipper path where
  Zipper :: hole
         -> Context (Up (left, hole, right) up)
         -> Zipper (Up (left, hole, right) up)
```

The `Context` type does most of the work of keeping track of keeping track of the types of siblings and parents. Since this type is so complicated we will tackle it in parts. This is its general shape:

```
data Context path where
  ContTop :: Context (Up (Top, a, Top) Top)
  Cont :: Left l (...)
       -> Right r (...)
       -> Context (Up (l_parent, h_parent, r_parent)
                   path)
       -> Context (Up (l, h, r)
                   (Up (l_parent, h_parent, r_parent)
                   path))
```

Except for the top-most context, `ContTop`, every `Context` contains a set of left siblings, right siblings and its parent context.

The parts marked by ellipses have been omitted for the moment. They are what ensures that the parent's hole, `h_parent`, is compatible with the current hole, `h`, and sibling types, `l` and `r`. But before those parts can be understood, we must consider how these siblings will be represented.

4.1 Left Siblings

Consider the following preliminary draft of a type which could be used to hold the left siblings of the current hole.

```
data BasicLeft a
  = BasicLeftUnit a
  | forall b. BasicLeftCons (BasicLeft (b -> a)) b
```

The key to this type is the `BasicLeftCons` constructor. Its first argument is a `BasicLeft` that *represents* a partially applied constructor of type `b -> a`. This is packaged up with the second argument of type `b`. This packaging represents the application of the former to the latter to construct an object of type `a`. Because it does not actually perform the application (it merely represents it) the `b` object can be re-extracted at a later time. These virtual applications can be stacked together with `BasicLeftCons`. The base case for this is the raw constructor before it has been applied to anything, and `BasicLeftUnit` allows that to be handled. [1]

[1] Readers familiar with [3] and [2] may recognize this type as `Spine`. However, variants of this type that appear later in this paper have differences that go beyond the work in [3] and [2]

Understanding this type should be much clearer with an example. Suppose for the moment that we want to use `BasicLeft` to represent constructor applications of the type `Foo`.

```
data Foo = Foo1 Int Char | Foo2 Float
```

To begin building a `Foo` object let's start with the `Foo1` constructor. This is represented by the value `BasicLeftUnit Foo1`. The type for such a value is `BasicLeft (Int -> Char -> Foo)` .

Notice that the arguments that `Foo1` is expecting are made manifest in the type of `BasicLeftCons`. In the following examples that part of the type signature guarantees that the arguments will be of the proper type as each `BasicLeftCons` is added.

```
*Main> :type BasicLeftUnit Foo1
              `BasicLeftCons` 1

it :: BasicLeft (Char -> Foo)

*Main> :type BasicLeftUnit Foo1
              `BasicLeftCons` 1
              `BasicLeftCons` 'a'

it :: BasicLeft Foo

*Main> :type BasicLeftUnit Foo2

it :: BasicLeft (Float -> Foo)

*Main> :type BasicLeftUnit Foo2
              `BasicLeftCons` 1.0

it :: BasicLeft Foo
```

The existentially quantified type `b` in `BasicLeftCons` allows a `BasicLeft` representing an `a` to contain whatever type of children are necessary. [2] The only requirement on the children is that they match the type of the arguments to the constructor.

However, using an existential has a drawback. Since the value of the type is hidden by the existential, the children must be treated as opaque objects. This is a problem if we want to implement a zipper, because when moving left, one of those hidden children will become the new hole. What we need is a type that reflects not only the types of the *remaining* arguments (as `BasicLeft` does), but also the types of the already applied arguments.

Fortunately, this can be achieved with the following type.

```
data Left contains expects where
  LeftUnit :: a -> Left Top a
  LeftCons :: Left c (b -> a)
           -> b
           -> Left (c -> b) a

data Top
  {- no constructors -}
```

The `expects` type parameter plays the same role as the parameter to `BasicLeft` did before. The `contains` parameter provides an additional record of the types of the children already added by `LeftCons`. Finally, the `Top` type provides a base case for the `contains` argument.

With `Left` instead of `BasicLeft` the previous example becomes:

```
*Main> :type LeftUnit Foo1

it :: Left Top (Int -> Char -> Foo)

*Main> :type LeftUnit Foo1
              `LeftCons` 1

it :: Left (Top -> Int) (Char -> Foo)

*Main> :type LeftUnit Foo1
              `LeftCons` 1
              `LeftCons` 'a'

it :: Left ((Top -> Int) -> Char) Foo

*Main> :type LeftUnit Foo2

it :: Left Top (Float -> Foo)

*Main> :type LeftUnit Foo2
              `LeftCons` 1.0

it :: Left (Top -> Float) Foo
```

The `expects` parameter is the same as it was before, but now the types of the children don't disappear when the `LeftCons` is applied.

4.2 Right Siblings

Representing the right siblings is very similar to how it was with the left siblings. The major difference is that instead of the type needing to encode what children the partial constructor application `expects`, the type needs to encode what children it `provides`.

```
data Right provides final where
  RightNull :: Right final final
  RightCons :: b
            -> Right a final
            -> Right (b -> a) final
```

(The `final` parameter to this type does no useful work for now, but will be used when we return to the `Context` type where it will help ensure that the nodes in the `Zipper` match that of their surrounding context.)

If the children provided by a `Right` match the children expected by a `Left`, we have enough to completely apply the constructor. If the corresponding `Left` already has all its arguments, `Right` doesn't need to provide any children. This case is represented by `RightNull`. Children that `Right` does provide are added with `RightCons`.

```
*Main> :type RightNull

it :: Right final final

*Main> :type 'a' `RightCons`
              RightNull

it :: Right (Char -> a) a

*Main> :type (1::Int) `RightCons`
              ('a' `RightCons`
              RightNull)

it :: Right (Int -> Char -> a) a
```

The fact that in this example, `final` is left universally quantified is slightly worrisome, but this will be soon rectified.

4.3 Combining Left and Right

Before returning to the problem of specifying a complete zipper context, consider how a matching `Left` and `Right` may be combined. With a left portion `Left l r` and a right portion

[2] GHC uses the `forall` keyword for existentials as well as universals. The only distinction between the two is where it is positioned.

`Right r final`, in principal there is enough information to construct a complete `final`. Since it will be useful later and in order to make sure that matching up `Left` and a `Right` in this way is feasible we should try to write a function that constructs such a `final` from a `Left` and a `Right`. But since we eventually want to leave a hole, `h`, in the context, we will instead write a function that takes a `Left l (h -> r)`, an `h` and a `Right r final`. The implementation of this function, `collapse`, proves to be quite simple:

```
collapse :: Left l (h -> r) -> h -> Right r final
         -> final
collapse l h r = total where
  left = collapse_left l
  mid = left h
  total = collapse_right mid r

collapse_left :: Left l r -> r
collapse_left (LeftUnit a)   = a
collapse_left (LeftCons f b) =
  collapse_left f b

collapse_right :: r -> Right r final -> final
collapse_right f (RightNull)     = f
collapse_right f (RightCons b r) =
  collapse_right (f b) r
```

Notice how the the `final` parameter to `Right` behaves here. From the way that `Right` is defined, the `final` parameter will always be always a suffix of the `provides` parameter. Further, in any call to `collapse` the `provides` parameter of the `Right` will be `r` which is also the `expects` parameter of the `Left`. So `final` must match what is at the end of the `expects` parameter, and this is precisely the result type of the constructor. This eliminates the problem with `final` being universally quantified seen in the earlier example, and is crucial to the implementation of `Context`.

4.4 Context

With both `Left` and `Right` defined, we can now return to the `Context` type. Given a matching `Left` and `Right`, all that remains to build a complete zipper context is the ability to point to a parent context. That parent context must have a hole that matches the type which could be constructed from the `Left` and `Right` siblings. This means that in order to maintain type correctness, the type of the parent context must encode the type of the hole that the parent context contains, as well as the types of the left and right siblings that the parent context contains. And since the the parent context itself has a parent context, by induction the current context's type will need to encode a complete path back to the root node.

At each point in this path there will be a left, a hole and a right type. We will package these together with tuple type. These tuples in turn will be linked together by the type constructor `Up`.

```
data Up a b
  {- no constructors -}
```

As before, the type `Top` will terminate such a chain. Like `Top`, the type `Up` has no constructors because it operates as a phantom type.

Putting all of this together the `Context` type may finally be defined thusly:

```
data Context path where
  ContTop :: Context (Up (Top, a, Top) Top)
  Cont :: Left l (h -> r)
       -> Right r h_parent
       -> Context (Up (l_parent, h_parent, r_parent)
                   path)
       -> Context (Up (l, h, r)
                   (Up (l_parent, h_parent, r_parent)
                   path))
```

While at first that data type may appear complicated, notice that each argument to the `Cont` constructor shares at least one type parameter with one of the other arguments. The `Left` shares `r` with the `Right`, and the `Right` shares `h_parent` with the parent `Context`. This means that we can produce a new `Context` if we have matching `Left` and `Right` siblings that combine to fill the hole, `h_parent`, in the parent `Context`. The `path` of the resulting `Context` is the `path` of the parent `Context` extended with the current sibling and hole types.

A `path` such as

```
(Up (self_left, self_hole, self_right)
(Up (parent_left, parent_hole, parent_right)
(Up (grandparent_left,
     grandparent_hole,
     grandparent_right)
Top)))
```

would mean that the current zipper position has a left, hole and right of `self_left`, `self_hole` and `self_right` respectively, the parent of the current zipper position has a left, hole and right of `parent_left`, `parent_hole` and `parent_right` respectively, and the grandparent of the current zipper position has a left, hole and right of `grandparent_left`, `grandparent_hole` and `grandparent_right` respectively. Lastly, the grandparent of the current zipper position has no further parents (signaled by the use of `Top`) and is thus the root of the object which the zipper traversing.

4.5 Zipper Operations

Implementing movement with the `Zipper` is quite easy. The implementation of `move_left` simply requires pulling off a `LeftCons` from the left part of the context and adding on a `RightCons` to the right part of the context.

```
move_left :: Zipper (Up (l -> h_new, h_old, r) up)
          -> Zipper (Up (l, h_new, h_old -> r) up)
move_left
  (Zipper h_old (Cont (LeftCons l h_new) r up)) =
  (Zipper h_new (Cont l (RightCons h_old r) up))
```

Doing the reverse gives us `move_right`.

```
move_right :: Zipper (Up (l, h_old, h_new -> r) up)
           -> Zipper (Up (l -> h_old, h_new, r) up)
move_right
  (Zipper h_old (Cont l (RightCons h_new r) up)) =
  (Zipper h_new (Cont (LeftCons l h_old) r up))
```

And reusing the `collapse` function from before, `move_up` is even easier.

```
move_up :: Zipper (Up child (Up self parent))
        -> Zipper (Up self parent)
move_up (Zipper h (Cont l r up)) =
        (Zipper (collapse l h r) up)
```

Finally, constructing a zipper from scratch with `begin_zipper`, getting the value of the current hole with `get_hole`, and setting the value of the current hole with `set_hole` are all trivial wrappers around the `Zipper` constructors.

```
begin_zipper :: h -> Zipper (Up (Top, h, Top) Top)
begin_zipper a = Zipper a ContTop
```

```
get_hole :: Zipper (Up (l, h, r) up) -> h
get_hole (Zipper h _) = h

set_hole :: h
         -> Zipper (Up (l, h, r) up)
         -> Zipper (Up (l, h, r) up)
set_hole h (Zipper _ context) = Zipper h context
```

4.6 Implementing Down

Up until now, none of the operations over the `Zipper` had any failure modes. The one remaining function, `move_down`, isn't quite so lucky. The previous functions could prevent failure by encoding the types of the parents and siblings in the type of the `Zipper`, but nothing in the existing `Context` or `Zipper` types indicate types of the children of the current node. There are a number of options that could remedy this situation.

The first is to go ahead and add that information to the type of the `Zipper`. Unfortunately, this would end up requiring the types of not just the immediate children, but of all descendants to be encoded in the type of the `Zipper`. While in some applications this might be acceptable, in many this would unreasonably constrain the type and shape of the data contained in the `Zipper`.

For some data types there is a second option. If the type of the children are always the same and are known in advance then an implementation of `move_down` could be written which takes that into account. With most data types this is not a viable option so we seek a more general solution.

The last option is to use `gfoldl` from [6] which provides just such generality but it does have a cost. The type signature of `gfoldl` is

```
gfoldl :: (Data a)
  => (forall a1 b. (Data a1)
      => c (a1 -> b) -> a1 -> c b)
       -- Lifted application
  -> (forall g. g -> c g)
       -- Constructor injection
  -> a -- The object to be folded
  -> c a
```

The semantics of `gfoldl` are such that the call

```
gfoldl f k (Foo1 5 'd')
```

is equivalent to

```
((k Foo1) `f` 5) `f` 'd'
```

The `gfoldl` function removes the need for the caller to perform case analysis or even know anything about the type being manipulated.

However, the result type, `c a`, does not manifest any of the types of the children that where folded over, and `Zipper` needs those types to construct the types of the current hole's siblings. In order to do this we must hide the `contains` parameter of `Left` from the `gfoldl` by wrapping it in the existential type `Erase`.

```
data Erase c a =
  forall b. (Typeable b) => Erase (c b a)
```

Once the `gfoldl` is complete the `contains` parameter is re-exposed by `cast` (also from [6]). This function casts one type to another, but returns its result wrapped a `Maybe` so that it can produce `Nothing` if the types are not compatible. Since such a cast may fail with `Nothing`, this introduces the possibility that `move_down` could fail. This is a design trade-off compared to the other options.

```
move_down ::
  (Typeable l_down, Typeable h_down,
   Typeable l, Data h, Typeable r, Typeable up)
  =>         Zipper (Up (l, h, r) up)
  -> Maybe (Zipper (Up (l_down, h_down, h)
                   (Up (l, h, r) up)))
move_down (Zipper h c) =
  case gfoldl erased_left_cons erased_left_unit h of
    Erase l ->
      case cast l of
        Just (LeftCons l' h_down) ->
          Just (Zipper h_down (Cont l' RightNull c))
        Nothing -> Nothing

instance Typeable Top where
  typeOf _ = mkTyConApp (mkTyCon "Top") []

instance Typeable2 Left where
  typeOf2 _ = mkTyConApp (mkTyCon "Left") []

instance Typeable2 Up where
  typeOf2 _ = mkTyConApp (mkTyCon "Up") []
```

The `erased_left_cons` and `erased_left_unit` functions are simply `LeftCons` and `LeftUnit` but with the first type constructor argument hidden by `Erase`.

```
erased_left_cons :: (Typeable b)
  => Erase Left (b -> a)
  -> b -> Erase Left a
erased_left_cons (Erase c) b =
  Erase (LeftCons c b)

erased_left_unit :: a -> Erase Left a
erased_left_unit a = Erase (LeftUnit a)
```

This design concentrates everything about a zipper that could fail into one function, `move_down`. The other functions will never fail thanks to the constraints enforced by their type signatures.

Because `move_down` contains a `cast` within it, the result type is ambiguous and will be left with universally quantified type variables. Anything using `move_down` would have to specify these variables in one way or another so the success or failure of the `cast` can be determined. This means the user would have to put an explicit type signature on each call to `move_down`. Since the signature of a call to `move_down` includes a full encoding of the path, requiring the user to write it out would be a bit of a burden. The following wrapper function provides a slightly easier alternative by inferring the type variables from the type of the constructor that the user claims is in the current hole.

```
move_down' ::
  (Typeable l_down, Typeable h_down,
   Typeable l, Data h, Typeable r, Typeable up,
   Foldl Top constr_type h (l_down -> h_down))
  =>         Zipper (Up (l, h, r) up)
  -> constr_type
  -> Maybe (Zipper (Up (l_down, h_down, h)
                   (Up (l, h, r) up)))
move_down' z _ = move_down z
```

A call like `move_down' z Foo1` instructs the `move_down'` function to assume that the the constructor of the current hole has the same type signature as `Foo1` and to infer the result type based on that. If the constructor has a different type, then `move_down'` will return `Nothing` just like `move_down` would have done if it was called with the wrong signature.

The purpose of Foldl is to compute the proper values for l_down and h_down from the provided constructor signature by flipping from the usual right associative function arrows to the left associative form needed by Left.

```
class Foldl acc right stop left
          | acc right stop -> left where
  {- no methods -}

instance Foldl acc stop stop acc where
  {- no methods -}

instance Foldl (acc -> a) b stop left
      => Foldl acc (a -> b) stop left where
  {- no methods -}
```

Despite its brevity, the definition of Foldl may be a bit challenging to understand. The motivated reader is encouraged to work through what value for left would be calculated by the functional dependencies in Foldl in order to satisfy the constraint

```
  Foldl Top (a -> b -> c -> d) d left
```

In any case Foldl is used only by move_down' and so is not essential to the other parts of this paper.

5. Beyond Zippers

Though the implementation of the generic Zipper relies heavily on GADTs, the technique of defining a data type that acts as a stand-in for applying a constructor to its arguments has broader applications even without GADTs. The original BasicLeft provided just such a stand-in but avoiding the use of GADTs. It was not sufficient to implement the generic zipper, but with a few modifications it has other uses.

What would happen if the second argument of BasicLeft where more than just a b? What if that argument were wrapped inside something such as a Maybe, an Either or a tuple? Or in another BasicLeft?

The Annotate type is the general form of these scenarios. With an Annotate m, each b is wrapped inside an m, and with FixAnnotate m, each b gets further wrapped inside yet another FixAnnotate. This effectively means every point in an algebraic data structure gets wrapped by an m from the top all the way down to the leaves.

```
newtype FixAnnotate  m a
  = FixA (m (FixAnnotate' m a))

type    FixAnnotate' m a
  = Annotate (FixAnnotate m) a

data Annotate m a
  = AnnotateUnit a
  | forall b. (Data b) =>
    AnnotateCons (Annotate m (b -> a))
                 (m  b)

instance (Typeable1 m) =>
          Typeable1 (Annotate m) where
  typeOf1 _ = mkTyConApp
               (mkTyCon "Annotate")
               [typeOf1 (undefined :: m ())]

instance (Typeable1 m) =>
          Typeable1 (FixAnnotate m) where
  typeOf1 _ = mkTyConApp
               (mkTyCon "FixAnnotate")
               [typeOf1 (undefined :: m ())]
```

If m is a Maybe, this allows any node within a data type to be either present as a Just or missing as a Nothing. If Either were used instead then any node could use an alternate set of constructors beyond those in the original data type being represented. Another possibility would be to use a tuple type so all the nodes would be annotated with some extra information but without replacing the existing value. The Annotate and FixAnnotate types encompass all of these possibilities. For example, with Maybe we can define a type that models one aspect of how Haskell style patterns behave, namely that a value may be left unspecified. We will not implement variable binding by a pattern here, but it is possible by using an Either String instead of a Maybe.

```
type Match  a = FixAnnotate  Maybe a
type Match' a = FixAnnotate' Maybe a
```

With this definition of Match, writing a function to check whether two such values "pattern match" against each other [3] is almost trivial. One simply needs to check if the constructors are the same and if the children match, but a Nothing matches against anything.

```
match :: (Data a) => Match a -> Match a -> Bool
match (FixA Nothing) _ = True
match _ (FixA Nothing) = True
match (FixA (Just x)) (FixA (Just y)) =
  same_constr x y &&
  match_children x y
```

The constructors can be extracted so they may be compared by using the toConstr function available in the Data.Generics module [6, 7, 8]. This requires the constructor to actually be applied to its arguments in order to have an object on which toConstr can operate, but those arguments might not all be available since any child could be a Nothing instead of a Just. Fortunately, the the particular value of those arguments will never be touched toConstr so we can safely use an error value in lieu the actual argument.

```
same_constr :: (Data a)
  => Match' a -> Match' a -> Bool
same_constr x y =
  toConstr (fold x) == toConstr (fold y) where
    fold :: Match' a -> a
    fold (AnnotateUnit f) = f
    fold (AnnotateCons f _) =
      (fold f) (error "Never used")
```

Because the children are quantified by an existential type, matching them against each other might at first seem to pose a problem, but that is easily remedied by using cast. Unlike the previous uses of cast in move_down, this wont cause extraneous failures; if the children being compared are of different types, then the constructors had to have been different and the pattern match we are implementing should return False anyway.

```
match_children :: (Typeable a)
  => Match' a -> Match' a -> Bool
match_children (AnnotateUnit _)
               (AnnotateUnit _) = True
match_children (AnnotateCons f_x b_x)
```

[3] Haskell patterns actually match a value against a pattern instead of two patterns against each other, but comparing two Match values against each other is easier to implement. A value can always be wrapped inside a Match, so this is also more general.

```
              (AnnotateCons f_y b_y) =
   case (cast f_y, cast b_y) of
     (Just f_y', Just b_y') ->
       match_children f_x f_y' &&
       match b_x b_y'
     _ -> False
 match_children _ _ = False
```

These tricks could of course be avoided for data types that have already been written in a fixed-point style, but for data types that are already written or for which writing a fixed-point would be difficult (e.g. non-homogeneous data types), the ability to add annotations to the data type in this way could be an easier option.

Also just as `Match` and `Annotate` both expand on the ideas at the root of `Left`, the core idea in `Right` may have other applications, but it will not be explored here.

6. Conclusion

The generic zipper goes beyond the capabilities of the traditional zipper in two ways. First, it doesn't require the user to write any boilerplate code to implement it. All it requires is an instance of `Data`. It doesn't automate any sort of all-at-once traversal because the zipper is instead designed for incremental traversals, but it will automate the incremental zipper movement operations.

Second and more importantly, the generic zipper is not limited to homogeneous data types. The traditional zipper can only deal with types such as `Term` where every node is of the same type. The generic zipper on the other hand, can handle not only `Term` but also types like `Dept` that have nodes with many different types. The generic zipper does this while ensuring type safety, and with the exception of `move_down`, it does this while avoiding the need to flag any sorts of errors.

Finally, the implementation techniques used by the generic zipper can be applied to other problems. One possible application, pattern matching, is sketched here but any problem where it would be useful to wrap the subparts of a data value in some other type could benefit from these techniques.

References

[1] Ralf Hinze and Johan Jeuring. Functional Pearl: Weaving a web. *Journal of Functional Programming*, 11(6):681–689, November 2001.

[2] Ralf Hinze and Andres Löh. "Scrap your boilerplate" revolutions. In *MPC*, pages 180–208, 2006.

[3] Ralf Hinze, Andres Löh, and Bruno C. D. S. Oliveira. "Scrap your boilerplate" reloaded. In *FLOPS*, pages 13–29, 2006.

[4] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[5] Oleg Kiselyov. Tool demonstration: A zipper based file/operating system. In *Haskell Workshop*. ACM Press, September 2005.

[6] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[7] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[8] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

[9] Conor McBride. The derivative of a regular type is its type of one-hole contexts. `http://www.cs.nott.ac.uk/~ctm/diff.pdf`, 2001. Unpublished manuscript, 2001.

[10] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, September 2006. ACM SIGPLAN.

[11] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[12] Don Stewart. Roll your own window manager: Tracking focus with a zipper. `http://cgi.cse.unsw.edu.au/~dons/blog/2007/05/17`, May 2007.