Scrap Your Zippers

A Generic Zipper for Heterogeneous Types

Michael D. Adams

School of Informatics and Computing, Indiana University

`http://www.cs.indiana.edu/~adamsmd/`

Abstract

The zipper type provides the ability to efficiently edit tree-shaped data in a purely functional setting by providing constant time edits at a focal point in an immutable structure. It is used in a number of applications and is widely applicable for manipulating tree-shaped data structures.

The traditional zipper suffers from two major limitations, however. First, it operates only on homogeneous types. That is to say, every node the zipper visits must have the same type. In practice, many tree-shaped types do not satisfy this condition, and thus cannot be handled by the traditional zipper. Second, the traditional zipper involves a significant amount of boilerplate code. A custom implementation must be written for each type the zipper traverses. This is error prone and must be updated whenever the type being traversed changes.

The generic zipper presented in this paper overcomes these limitations. It operates over any type and requires no boilerplate code to be written by the user. The only restriction is that the types traversed must be instances of the `Data` class from the *Scrap your Boilerplate* framework.

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

General Terms Design, Languages.

Keywords Generic programming, Zippers, Scrap your Boilerplate, Heterogeneous Types

1. Introduction

The zipper type provides the ability to efficiently edit tree-shaped data in a purely functional setting [14]. It has been used to implement text editors [6], file systems [15], and even window managers [31]. In general, the zipper is applicable whenever there is a well-defined focal point for edits. In a text editor, this focal point is manifest as the user's cursor. In a file system, it is manifest as the current working directory. In a window manager, it is the window with focus. Creating a zipper is a straightforward, formulaic process that derives from the shape of the type that the zipper traverses [9, 24].

The traditional zipper suffers from two major limitations, however. First, it is limited to operating over homogeneous types. Each node in the tree that the zipper traverses must be of the same type. Thus the traditional zipper works wonderfully for making edits to an abstract syntax tree for the lambda-calculus where everything is an expression, but it is unable to manipulate an abstract syntax tree for a language that, in addition to expressions, contains type annotations or statements.

The second limitation is the problem with any regularly structured, formulaic piece of code: It shouldn't be written in the first place! At least not by a human. Formulaic code is monotonous to write and maintain, and the zipper must be updated every time the underlying type changes. Tools exist to automatically generate zipper definitions from a type (e.g., using Template Haskell [30]), but this introduces a meta-layer to our programs that it would be better to avoid unless really necessary.

The generic zipper presented in this paper overcomes these limitations. It operates over any type irrespective of whether it is homogeneous or not. It requires no boilerplate code, and integrates within the existing Haskell language without a meta-level system. The only restriction is the types the zipper traverses must be instances of the `Data` class from the *Scrap your Boilerplate* framework [20–22].

This paper borrows terminology and operators from the *Scrap your Boilerplate* framework. They are explained when first used. Nonetheless, the reader might find it helpful to have some familiarity with that framework.

Also, the internals of the generic zipper use existential types and GADTs [26], but these do not show up in the user-level interface. Thus a basic understanding of them is necessary to understand the implementation but not to use the generic zipper.

The code for the generic zipper is available as a Haskell library from the HackageDB repository at `http://hackage.haskell.org/package/syz`.

The remainder of this paper is organized as follows. Section 2 reviews the traditional zipper and its implementation. Section 3 introduces the generic zipper and how to use it. Section 4 applies the generic zipper to generic traversals. Section 5 shows the implementation of the generic zipper. Section 6 reviews related work. Section 7 concludes.

2. Traditional Zippers

A zipper is made up of two parts: a one-hole context and a hole value to fill the hole in the context. The hole value is the portion of the object rooted at the current position of the zipper within the overall object. This position is called the focus. The context contains the overall object but with the hole value missing. To see how this works for the traditional zipper we follow the development in Hinze and Jeuring [9] before moving on to the generic zipper.

```
down_Term :: TermZipper → Maybe TermZipper
down_Term (Var s, c)           = Nothing
down_Term (Lambda s t_1, c)    = Just (t_1, Lambda_1 s c)
down_Term (App t_1 t_2, c)     = Just (t_1, App_1 c t_2)
down_Term (If t_1 t_2 t_3, c)  = Just (t_1, If_1 c t_2 t_3)
```

```
up_Term :: TermZipper → Maybe TermZipper
up_Term (t_1, Root_Term)        = Nothing
up_Term (t_1, Lambda_1 s c)     = Just (Lambda s t_1, c)
up_Term (t_1, App_1 c t_2)      = Just (App t_1 t_2, c)
up_Term (t_2, App_2 t_1 c)      = Just (App t_1 t_2, c)
up_Term (t_1, If_1 c t_2 t_3)   = Just (If t_1 t_2 t_3, c)
up_Term (t_2, If_2 t_1 c t_3)   = Just (If t_1 t_2 t_3, c)
up_Term (t_3, If_3 t_1 t_2 c)   = Just (If t_1 t_2 t_3, c)
```

```
left_Term :: TermZipper → Maybe TermZipper
left_Term (t_1, Root_Term)        = Nothing
left_Term (t_1, Lambda_1 s c)     = Nothing
left_Term (t_1, App_1 c t_2)      = Nothing
left_Term (t_2, App_2 t_1 c)      = Just (t_1, App_1 c t_2)
left_Term (t_1, If_1 c t_2 t_3)   = Nothing
left_Term (t_2, If_2 t_1 c t_3)   = Just (t_1, If_1 c t_2 t_3)
left_Term (t_3, If_3 t_1 t_2 c)   = Just (t_2, If_2 t_1 c t_3)
```

```
right_Term :: TermZipper → Maybe TermZipper
right_Term (t_1, Root_Term)        = Nothing
right_Term (t_1, Lambda_1 s c)     = Nothing
right_Term (t_1, App_1 c t_2)      = Just (t_2, App_2 t_1 c)
right_Term (t_2, App_2 t_1 c)      = Nothing
right_Term (t_1, If_1 c t_2 t_3)   = Just (t_2, If_2 t_1 c t_3)
right_Term (t_2, If_2 t_1 c t_3)   = Just (t_3, If_3 t_1 t_2 c)
right_Term (t_3, If_3 t_1 t_2 c)   = Nothing
```

Figure 1. Zipper Movement Operations for `Term`.

2.1 Implementing the Zipper

Consider this abstract syntax tree for a hypothetical language:

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

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. For each recursive child of each constructor in `Term`, the `TermContext` type has a constructor with that child missing and replaced by a reference to a parent context.

```
type TermZipper = (Term, TermContext)
data TermContext
  = Root_Term
  | 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
```

```
fromZipper_Term :: TermZipper → Term
fromZipper_Term z = f z where
  f :: TermZipper → Term
  f (t_1, Root_Term)       = t_1
  f (t_1, Lambda_1 s c)    = f (Lambda s t_1, c)
  f (t_1, App_1 c t_2)     = f (App t_1 t_2, c)
  f (t_2, App_2 t_1 c)     = f (App t_1 t_2, c)
  f (t_1, If_1 c t_2 t_3)  = f (If t_1 t_2 t_3, c)
  f (t_2, If_2 t_1 c t_3)  = f (If t_1 t_2 t_3, c)
  f (t_3, If_3 t_1 t_2 c)  = f (If t_1 t_2 t_3, c)
```

```
toZipper_Term :: Term → TermZipper
toZipper_Term t = (t, Root_Term)
```

```
getHole_Term :: TermZipper → Term
getHole_Term (t, _) = t
```

```
setHole_Term :: Term → TermZipper → TermZipper
setHole_Term h (_, c) = (h, c)
```

Figure 2. Zipper Non-Movement Operations for `Term`.

In the `Term` type:

– The `If` constructor has three children, so there are three corresponding constructors in `TermContext`.
– Likewise, `App` has two children and two corresponding constructors in `TermContext`.
– Since the traditional zipper operates over homogeneous types, the `String` argument to `Lambda` isn't considered a child, thus `Lambda` has only one constructor in `TermContext`, namely the constructor for when the `Term` child of `Lambda` is missing.
– Finally, `Var` has no `Term` children, so it has no corresponding constructors in `TermContext`.

Each `TermContext` contains a reference to a parent context, which in turn points to its own parent context. These parent contexts form a chain back to the root of the overall object. The chain is terminated by the `Root`$_{\text{Term}}$ constructor. Effectively, these contexts are the result of a pointer reversal. Instead of the parent pointing to the child, the child now points to the parent.

Navigating a `TermZipper` is implemented by `down`$_{\text{Term}}$, `up`$_{\text{Term}}$, `left`$_{\text{Term}}$ and `right`$_{\text{Term}}$ as shown in Figure 1. Moving down happens by deconstructing the hole value, extracting a child object, and extending the context with the other children. The extracted child object then becomes the new hole value.

Moving up is the reverse process. The siblings objects stored in the current context are combined with the hole value to form a new hole value, and the parent context becomes the current context.

Moving left and right are 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.

The movement operations may fail by returning `Nothing` if a movement is illegal (e.g., moving left when already at the leftmost position). Another common way the zipper interface can be defined is to ignore illegal movements and return the unchanged zipper instead of returning a `Maybe` type. Nevertheless, explicitly signaling illegal movements provides more information to the user and is used when defining generic zipper traversals in Section 4.

Finally, as shown in Figure 2, the zipper has functions for converting a value to and from a zipper, getting the hole value, and setting the hole value. With the exception of $\texttt{fromZipper}_{\texttt{Term}}$, these are trivial wrapper functions manipulating the pair that forms a zipper. The $\texttt{fromZipper}_{\texttt{Term}}$ function moves all the way to the root context before returning the resulting value.

2.2 Using the Zipper

To see how a zipper is used in practice, consider this abstract syntax tree that defines a factorial function:

```
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")))))
```

This definition contains a bug. The + operator should really be the * operator. A zipper can fix this.

First $\texttt{toZipper}_{\texttt{Term}}$ creates a new zipper. That zipper starts with a focus at the root of the object so the hole will contain the original definition of fac.

```
*Main> let t_0 = toZipper_Term fac
*Main> getHole_Term t_0

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")))))
```

The first step to getting to the offending * is to use $\texttt{down}_{\texttt{Term}}$. Now the hole contains the body of the Lambda.

```
*Main> let Just t_1 = down_Term t_1
*Main> getHole_Term t_1

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

Moving down again focuses the zipper on the test part of the If, but we want to go to the third child of the If, so we also move using $\texttt{right}_{\texttt{Term}}$ twice.

```
*Main> let Just t_2 = down_Term t_1
*Main> getHole_Term t_2

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

*Main> let Just t_3 = right_Term t_2
*Main> getHole_Term t_3

Var "1"

*Main> let Just t_4 = right_Term t_3
*Main> getHole_Term t_4

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

The zipper moves down again into the function position of the App.

```
*Main> let Just t_5 = down_Term t_4
*Main> getHole_Term t_5

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

The zipper moves down one last time to get to the Var "+" that we want to change to Var "*".

```
*Main> let Just t_6 = down_Term t_5
*Main> getHole_Term t_6

Var "+"
```

Finally, $\texttt{setHole}_{\texttt{Term}}$ changes the value.

```
*Main> let t_7 = setHole_Term (Var "*") t_6
```

By moving up one level, we can see that our change is reflected in the larger term.

```
*Main> let Just t_8 = up_Term t_7
*Main> getHole_Term t_8

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

After moving up a few more times we can retrieve the corrected definition of fac.

```
*Main> let Just t_9 = up_Term t_8
*Main> let Just t_10 = up_Term t_9
*Main> let Just t_11 = up_Term t_10
*Main> getHole_Term t_11

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")))))
```

Or we can use $\texttt{fromZipper}_{\texttt{Term}}$ to go directly to the root and get the corrected definition.

```
*Main> fromZipper_Term t_7

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")))))
```

3. Using the Generic Zipper

While the traditional zipper works fine for homogeneous types like Term, it runs into problems with more complex types. Consider 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
```

Instead of just a single type in the case of Term, a zipper for this has to traverse five different types: Dept, Employee, Salary, Manager, and Name. Situations like this happen often. A compiler may have an abstract syntax tree that represents statements and type annotations in addition to expressions, or a graphical user interface toolkit may represent different classes of widgets with different types. The traditional zipper cannot handle this. Furthermore, for each new type, the traditional zipper requires a complete rewrite of the boilerplate code in Figures 1 and 2.

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)
```

Figure 3. Generic Zipper Interface.

The generic zipper surmounts these issues. It operates on non-homogeneous types like `Dept` and `Employee` just as well as on homogeneous types like `Term`. It also makes no difference if the type is directly or indirectly recursive. Finally, the generic zipper requires no boilerplate code on the user's part. 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 [7]. The interface to the generic zipper is shown in Figure 3.

As an example of using the generic zipper, consider the following department:

```
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
```

Suppose Agamemnon decides that his employee record should really refer to him as *King* Agamemnon. To start, `toZipper` creates a new generic zipper.

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

g1 :: Zipper Dept
```

We would like to inspect the hole value, but first we must understand how the generic zipper deals with the type of the hole. With a traditional zipper, the type of the hole is fixed. For example, $\texttt{getHole}_{\texttt{Term}}$ always returns a `Term`. But with a generic zipper, the type of the hole changes as the focus of the zipper moves around. While the type of the zipper contains the type of the root object, `Dept`, it hides the type of the current hole. Since the zipper is still at the root, the hole has the value and type of the original object, but the compiler doesn't know that. It knows only the type of the zipper, which doesn't expose the type of the hole. This is resolved by the hole-manipulation functions, `query`, `trans`, and `transM`. Each is defined in terms of a user-supplied generic function that operates on any argument type (e.g., the universally quantified `a` in Figure 4) provided the type is an instance of the `Data` class. To retrieve the contents of the hole, we supply to `query` a generic query function of type `∀ a. (Data a) => a → r`. For this example, we borrow the type-safe `cast` function from the *Scrap your Boilerplate* framework. It has the following type, where the `Typeable` class is a superclass of the `Data` class. It returns `Nothing` to indicate cast failure.

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

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 define a helper for it:

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

None of the core generic-zipper functions involve any casts. It is up to the user when a cast is used. Readers familiar with the *Scrap your Boilerplate* framework should note the generic zipper shares a similar design philosophy in this regard. For example, in *Scrap your Boilerplate* the generic mapping functions, `gmapT`, `gmapQ`, and `gmapM`, take a user-supplied generic function. As a consequence, the generic zipper also shares similar advantages (i.e., casts occur only where the user specifies) and disadvantages (i.e., sometimes the user must specify a cast).

To change the king's title, the zipper must navigate to the proper position. The first step is to move `down`. Just like the traditional zipper, if the current node has no children, `down` returns `Nothing`. Otherwise it returns `Just`.

```
*Main> let Just g2 = down g1
*Main> getHole g2  :: Maybe [Employee]

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

The zipper descends to the right-most child instead of the left-most 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)
```

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"
```

```
type GenericQ r = ∀ a. (Data a) => a → r
type GenericT   = ∀ a. (Data a) => a → a
type GenericM m = ∀ a. (Data a) => a → m a
```

Figure 4. Type Aliases for Generic Functions.

Once the zipper is focused at the right place, we are ready to give the king his proper title. This involves manipulating the contents of the hole, so we use the same trick as when retrieving the contents of the hole. Specifically, we use the `trans` function, which applies a generic transformer to the hole. To construct a generic transformer for this example, we again borrow a function from *Scrap your Boilerplate*. This time it is the `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. Like before with `getHole`, `mkT` implements the helper function `setHole`:

```
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`.

While we are giving the king his proper title, let's also move to the right and give the king a raise.

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

Finally, by moving up once more we can retrieve the now modified root object, or we can also get it using `fromZipper`, which automatically moves all the way to the root of the zipper and returns the resulting object.

```
*Main> fromZipper g9
D (E "King Agamemnon" 8000.0)
  [E "Menelaus" 3000.0,
   E "Achilles" 2000.0,
   E "Odysseus" 2000.0]
```

As mentioned before, every one of these operations is completely type-safe, and 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. This is also the case with a traditional zipper, and like the traditional zipper, it is also possible to define versions of the generic zipper movement functions that ignore illegal movements instead of returning a `Maybe` type.

4. Generic Traversals with Zippers

In the preceding section, the zipper traversed over a specific type, namely `Dept`. In that case, the generic zipper offers two advantages: it operates over heterogeneous types, and it does not entail writing any boilerplate code. Still, we can go a step further. The generic zipper can express generic traversals just as easily as non-generic traversals. Many generic programming systems provide generic traversals already, but the generic zipper is particularly suited to expressing traversals. After all, traditional zippers were invented for term traversal. The generic zipper merely makes it possible for the traversal to be generic.

4.1 Traversal Helpers

Before writing any generic zipper traversals, we define higher-level movement zipper operations in Figure 5. They abstract out common usage patterns and automatically handle the `Maybe` returned by the zipper movement functions. They are defined in terms of the core zipper movement primitives, so they do not change the expressive power of the generic zipper.

Query Movement These functions apply their `f` argument to the result of a movement, but only if the movement is legal. Otherwise, they return their `b` argument. For example, the following returns `False` as there is no sibling to the left of g_6.

```
  leftQ False (const True) g6
```

On the other hand, the following returns `Just 8000.0` as the right sibling of g_6 is the king's salary.

```
  rightQ Nothing getHole g6 :: Maybe Salary
```

Transformer Movement These functions extend query movement by replacing the hole value with the result of `f` and then moving the zipper back to its original position. If the movement is illegal, they leave the hole unchanged. Thus when changing Agamemnon's ... ahem ... *King* Agamemnon's salary, we could leave the g_6 zipper focused on his title and skip the intermediate g_7 step:

```
*Main> let g8 =
         rightT (setHole (8000.0 :: Float)) g6
*Main> getHole g8 :: Maybe Name

 Just "King Agamemnon"

*Main> let Just g9 = up g8
*Main> getHole g9 :: Maybe Employee

 Just (E "King Agamemnon" 8000.0)
```

The `upT` function has an additional complication as naively moving down after moving up leaves the zipper at the rightmost sibling instead of the original position. Thus to preserve the original position, it wraps extra left and right movements around the core up and back down movement.

Sibling Movement Finally, the `leftmost` and `rightmost` functions move a zipper to the leftmost or rightmost sibling by repeatedly applying `leftQ` and `rightQ`. These are used to dictate which child to start at after a downward movement.

4.2 Generic Bottom-Up Traversal

As an example of a generic zipper traversal, consider the classic bottom-up traversal that applies a given transformer in post-order to every node in the tree structure of an object. Algorithmically, the traversal consists of moving down whenever a node has children and recursively applying the traversal to each child. After the traversal is applied to the children, the transformer is applied to the current node.

In Figure 6, `zeverywhere` expresses this algorithm in terms of a generic zipper. The `downT` function applies `g` to the rightmost child whenever the current node has children. When there are no children, it returns the zipper unchanged. The `g` function then applies `zeverywhere` to the child and uses `leftT` to iteratively apply `g` to the remaining left siblings. This stops when `leftT`

Query Movement

```
leftQ, rightQ, downQ, upQ ::
  b → (Zipper a → b) → Zipper a → b
leftQ  b f z = moveQ left  b f z
rightQ b f z = moveQ right b f z
downQ  b f z = moveQ down  b f z
upQ    b f z = moveQ up    b f z

moveQ move b f z = case move z of
                     Nothing → b
                     Just z → f z
```

Transformer Movement

```
leftT, rightT, downT, upT ::
  (Zipper a → Zipper a) → Zipper a → Zipper a
leftT  f z = moveT left  right z f z
rightT f z = moveT right left  z f z
downT  f z = moveT down  up    z f z
upT    f z = g z where
  g z = moveT right left (h z) g z
  h z = moveT up    down z     f z

moveT move₁ move₂ b f z =
  moveQ move₁ b (moveQ move₂ b id . f) z
```

Sibling Movement

```
leftmost, rightmost :: Zipper a → Zipper a
leftmost  z = leftQ  z leftmost z
rightmost z = rightQ z rightmost z
```

Figure 5. Traversal Helper Functions.

detects that there are no more left siblings. At that point the call stack unwinds and `leftT` and `downT` move the zipper right and up to its original position. Finally, the `trans` function applies `f` to the value in the hole.

4.3 Generic Outermost-Leftmost Reduction

The bottom-up traversal that `zeverywhere` implements is rather simple. Most generic programming systems easily express it. The generic zipper, however, can also express much more sophisticated traversals. Consider the problem of repeatedly applying the outermost, leftmost reduction. The first reduction is easily found by a standard top-down, left-to-right traversal. The reductions after that require more care because applying a reduction may make new reductions possible in the ancestors of the current node. The traversal must always apply the outermost reduction first. A naive solution is to restart the traversal at the root after each reduction, but this is inefficient. Only direct ancestors of the current node can contain a new reduction. Already traversed siblings need not be searched. A better solution searches only these ancestors.

This more efficient algorithm is presented in two steps using the generic zipper. The first step exposes explicit control of the zipper movements. The second step adds searching for reducible ancestors.

4.3.1 Exposing Explicit Control

The `zeverywhere'` function in Figure 6 implements the first step. It differs from `zeverywhere` in that it is top-down and left-to-right, but more importantly, it explicitly controls movement up the zipper

```
zeverywhere :: GenericT → Zipper a → Zipper a
zeverywhere f z = trans f (downT g z) where
  g z = leftT g (zeverywhere f z)
```

```
zeverywhere' :: GenericT → Zipper a → Zipper a
zeverywhere' f z =
  downQ (g x) (zeverywhere' f . leftmost) x where
    x = trans f z
    g z = rightQ (upQ z g z) (zeverywhere' f) z
```

Figure 6. Bottom-up and Top-down Zipper Traversals.

instead of letting `downT` automatically move the up zipper. This explicit control is used in the second step when checking ancestors for new reductions.

To make the traversal top-down, the `trans` function is applied before recurring instead of after. To make it left-to-right, not only are left movements replaced with right movements, but when moving down, `leftmost` starts the traversal at the leftmost child.

The original `zeverywhere` function used `downT` and `leftT` to return the zipper automatically to its starting position. This is a problem when a new reduction is found in one the ancestors and the zipper starts traversing at a brand new position. Thus instead of `downT` and `rightT`, `zeverywhere'` uses `downQ` and `rightQ`. The former two automatically move the zipper back up or left after they are done, but the latter two do not.

Using these, `zeverywhere'` expresses its traversal in terms of the well-known algorithm for traversing a tree in constant stack space [18]. First, it keeps recurring down the leftmost child using `downQ` and `leftmost`. Once it reaches a leaf where there are no children, `downQ` evaluates to `g x`. The `g` function searches for a place to move right by starting with the current node and checking each ancestor. If there is no right sibling, then `rightQ` calls `upQ`, which moves the zipper up and calls back to `g` to continue searching. Once `g` finds a place to move right, it starts moving down again by calling `zeverywhere'`. Finally, if it reaches the root in the process of finding a place to move right, then the traversal stops and returns the first argument of `upQ`, namely `z`.

Stated simply, the traversal moves down until it cannot do so anymore at which point it tries to move right. If it cannot move right at the current node, then it moves up until it finds a place to move right. Once it finds a place to move right, it continues the downward traversal.

4.3.2 Checking Ancestors

The second step uses the explicit movement control to check for new reductions in the ancestors. This is implemented by `zreduce` in Figure 7. In `zeverywhere'`, `f` has type:

```
∀ a. (Data a) => a → a
```

But in `zreduce`, `f` has type:

```
∀ a. (Data a) => a → Maybe a
```

This is so `f` can signal whether it succeeded at applying a reduction to the current node. Because of this change, `zreduce` uses `transM` instead of `trans`. The `transM` function lifts the `Maybe` value so that when `f` returns `Nothing`, `transM` also returns `Nothing`, and when `f` returns `Just`, `transM` also returns `Just`.

When `f` cannot apply a reduction, it returns `Nothing`. In that case, `zreduce` continues exactly the same as `zeverywhere'`. When `f` succeeds at applying a reduction, it returns `Just` and `reduceAncestors` runs to see if any ancestors are reducible and repositions the zipper accordingly. Once `reduceAncestors` finishes, zreduce continues the traversal at the position that reduceAncestors left the zipper.

The reduceAncestors function takes an extra argument, def, in addition to the transformer, f, and the current zipper, z. The def argument is the default value that reduceAncestors should return if it finds no reducible ancestors of z. It is returned by upQ when z has no parent. Otherwise, g is called with the parent zipper and continues the search with a default value of def'. When computing def', if the parent is not reducible, then transM f z returns Nothing and the original default is returned. But if the parent is reducible, then the reduced zipper, x, is returned after it is checked to see if the reduction caused other ancestors to be reducible. Due to Haskell's laziness, the ancestor check on x is not computed unless there are no further reducible ancestors of z that take precedence over x.

Implementing zreduce with the generic zipper requires only straightforward, direct-style zipper manipulation. The generic zipper inherits the advantages that the traditional zipper has in expressing traversals and extends them to generic traversals.

```
zreduce :: GenericM Maybe → Zipper a → Zipper a
zreduce f z =
  case transM f z of
    Nothing →
      downQ (g z) (zreduce f . leftmost) z where
        g z = rightQ (upQ z g z) (zreduce f) z
    Just x  → zreduce f (reduceAncestors f x x)
```

```
reduceAncestors ::
  GenericM Maybe → Zipper a → Zipper a → Zipper a
reduceAncestors f z def = upQ def g z where
  g z = reduceAncestors f z def' where
    def' = case transM f z of
             Nothing → def
             Just x  → reduceAncestors f x x
```

Figure 7. Optimized Zipper Reduction Traversal.

5. Implementing the Generic Zipper

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[1] hole and a context that matches both the hole and the zipper's root type.

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

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 $\texttt{Null}_{\texttt{Ctxt}}$, 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
```

The parts marked by ellipses are omitted for now. They ensure that the parent's hole is compatible with the current hole and siblings. We will fill them in after we see how the siblings are represented by Left and Right.

5.1 Left Siblings

The Left type[2] holds the left siblings of the current hole:

```
data Left expects
  = UnitLeft expects
  | ∀ b. (Data b) => ConsLeft (Left (b → expects)) b
```

The key to understanding this type is b, the existentially quantified type variable in $\texttt{Cons}_{\texttt{Left}}$. The first argument of $\texttt{Cons}_{\texttt{Left}}$ is a Left that represents a partially applied constructor of type b → expects.

[1] GHC uses the ∀ keyword for both existential and universal types. The distinction between the two is where the keyword is positioned.

[2] Readers familiar with the *Scrap your Boilerplate: Reloaded* framework [10, 12] may recognize this type as Spine.

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. Because $\texttt{Cons}_{\texttt{Left}}$ does not actually perform the application—it merely represents it—the b object can be extracted at a later time. 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 $\texttt{Unit}_{\texttt{Left}}$.

Understanding this type should be clearer with an 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 $\texttt{Unit}_{\texttt{Left}}$ Bar. The type of this value is:

```
 UnitLeft Bar :: Left (Int → Char → Foo)
```

The arguments that Bar is expecting are manifest in the type of the resulting Left object. We can add those arguments with $\texttt{Cons}_{\texttt{Left}}$, and the way $\texttt{Cons}_{\texttt{Left}}$ 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
```

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

5.2 Right Siblings

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.

```
data Right provides parent where
  Null_Right  :: Right parent parent
  Cons_Right  ::
    (Data b) => b → Right a t → Right (b → a) t
```

The Null$_{\text{Right}}$ 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 Cons$_{\text{Right}}$. 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.

Consider a Right that represents right siblings to be fed to the Bar constructor. Every Right starts off with a Null$_{\text{Right}}$:

```
*Main> :type Null_Right
 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 Cons_Right  'a' Null_Right
 it :: Right (Char → a) a
```

Next the preceding argument to Bar is added:

```
*Main> :type Cons_Right  1 (Cons_Right  'a' Null_Right )
 it :: Right (Int → Char → a) a
```

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 → Char → 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 that is rectified in the next section.

5.3 Combining Left and Right

Before returning to the complete zipper context, consider how a matching Left and Right are combined. 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.

The combine function in Figure 8 does this combination, but it also leaves room for an extra argument, hole, that goes between the Left and the Right. 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. 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. Conceptually, combine is an evaluator for the language of applications embodied in Left and Right.

Consider how the parent parameter to Right behaves. 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. So parent must match what is at

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

```
fromLeft :: Left r → r
fromLeft (Unit_Left a)   = a
fromLeft (Cons_Left f b) = fromLeft f b
```

```
fromRight :: r → Right r parent → parent
fromRight f (Null_Right )     = f
fromRight f (Cons_Right  b r) = fromRight (f b) r
```

Figure 8. The combine implementation.

the end of the expects parameter, and this is precisely the result type of the constructor. This eliminates the problem with parent being universally quantified seen in the earlier example, and along with the matching of the expects and provides type parameters, serves a key role in the implementation of Context.

5.4 Context

With both Left and Right defined, we can now return to the Context type. 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 definition of Context is as follows, where both rights and parent are existentially quantified:

```
data Context hole root where
    Null_Ctxt :: Context a a
    Cons_Ctxt ::
      ∀ rights parent. (Data parent) =>
        Left (hole → rights)
        → Right rights parent
        → Context parent root
        → Context hole root
```

While this type may appear complicated, notice that each argument to the Cons$_{\text{Ctxt}}$ constructor shares at least one type parameter with one of the other arguments. The Left shares rights with the Right, and the Right shares parent with the parent Context. This means that we can produce a new Context only if we have matching Left and Right siblings that combine to fill the parent hole in the parent Context. The root of the resulting Context is the same as the root of the parent Context.

5.5 Zipper Operations

Once the types are defined, implementing movement for the generic zipper is easy. The implementations of left, right and up are shown in Figure 9. The left function simply pulls a Cons$_{\text{Left}}$ off of the left part of the context and adds a Cons$_{\text{Right}}$ onto the right part of the context. The right function does the reverse. Similarly, the up function pulls a Cons$_{\text{Ctxt}}$ off of the context and, using combine, constructs a new value for the hole of the parent context.

Finally, in Figure 10 we define functions for constructing a zipper from scratch with toZipper, extracting the root object of a zipper with fromZipper, querying the value of the current hole

```
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)
```

Figure 9. Generic Zipper Movement.

```
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)
```

Figure 10. Generic Zipper Non-Movement Operations.

with query, and transforming the value of the current hole with trans and transM. With the exception of fromZipper these are trivial wrappers around the Zipper constructor. The fromZipper function does the same as up except it continues moving up until it reaches the root.

5.6 Implementing Down

With a single exception, all of the core zipper operations simply shuffle the constructors of a zipper around. The one exception, down, is slightly more sophisticated. The other operations manipulate contexts and siblings that already exist as Context, Left and Right values. When moving down, however, those values do not yet exist; they must be built by deconstructing the hole.

To construct those values we use the gfoldl function defined in the Data class from the *Scrap your Boilerplate* framework:

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

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'
```

The gfoldl function is significant because it provides a way to access the pieces of a value without performing case analysis or knowing anything about the type being manipulated. GHC automatically generates a definition of it when deriving a Data class instance for a type.

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'
```

The down function is implemented by injecting the hole into a Left with toLeft and extracting its rightmost element. This rightmost element becomes the new hole, and the remaining elements become the left siblings. If there is no rightmost element (i.e., the UnitLeft case), then the original hole had no children and downward movement is illegal. In that case Nothing is returned.

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

The use of gfoldl is the source of two peculiarities in the implementation. First, it the reason for the Data class constraints that appear in the Context, Left, and Right types. These constraints ensure that we can apply gfoldl to any object that could, through some combination of left, right, and up movements, arrive at the hole of the zipper.

Second, because gfoldl is a left fold, 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. This differs from most traditional zippers, which start at the leftmost child. If the user desires a version of down that starts at the leftmost child, this is easily implemented by:

```
down' z = liftM leftmost (down z)
```

6. Related Work

6.1 Generic Programming

There has been a tremendous amount of research on generic programming and generic types [13, 27]. Here we focus on three systems that have a particularly close connection to the generic zipper.

The first is the *Scrap your Boilerplate* framework [20–22]. Our generic zipper builds upon and integrates smoothly with its design and philosophy. For example, the Data class originates there. As mentioned in Section 4, most traversals in the framework can be reimplemented in terms of a generic zipper and are often more straightforward with a zipper. For example, the simultaneous traversal of two values presented in Lämmel and Peyton Jones [21] takes a bit of thought to construct, but with the generic zipper the solution is trivial: use a separate zipper for each value. *Scrap your Boilerplate* implements more than just traversals. It also defines type-safe casts, constructor introspection, and other generic programming facilities. The generic zipper provides only traversal, and thus supplements the framework, but does not replace it.

The second is the *Scrap your Boilerplate: Reloaded* framework [10, 12]. The `Left` type and `toLeft` function for the generic zipper are equivalent to the `Spine` type and `toSpine` function from that framework. Both share the same observation that a value can be dissected and represented as a constructor and a list of arguments. The generic zipper takes the extra step of splitting this list into left siblings, right siblings, and hole.

The third is the work on heterogeneous collections by Kiselyov et al. [17]. In our generic zipper, `Right` is essentially a heterogeneous list that uses function-arrow notation (i.e., `a → b`) to encode the types it contains. Likewise `Context` is a heterogeneous list that restricts the `hole` of one element to match the `parent` of another. Thus on the surface it seems that heterogeneous collections could be used instead of GADTs. Nevertheless, encoding the type constraints on heterogeneous collections is a subtle and delicate task, and it is not clear that these particular constraints are expressible.

6.2 Zippers

The concept of a zipper is first documented by Huet [14]. That construction limits zippers to homogeneous types where the type of the hole is fixed and cannot change during traversal. In addition, each type needs its own custom-written zipper type and zipper movement functions.

This is simplified by Hinze and Jeuring [9] so that only one function has to be rewritten for each new type, but it still operates only on homogeneous types and requires boilerplate code.

6.2.1 One-hole Contexts

The theory behind the one-hole contexts that are part of a zipper is extensively studied by McBride [24] and later Abbott et al. [1, 2]. They show how to express a one-hole context in terms of a formal derivative and formalize the mechanistic generation of contexts.

This is used by Morris et al. [25] to implement a zipper for the dependently typed language Epigram that needs no boilerplate code. This zipper does not operate on standard user-defined types. Instead, the types are defined in terms of an explicit representation type `Reg` that expresses all types in terms of primitive type-functor operations such as products and sums. A more detailed account is given by Altenkirch et al. [5].

In these constructions, the type of the hole must remain fixed for the lifetime of a zipper, but it may vary between different traversals. So in our example from Section 3, if the types are encoded in terms of `Reg`, we can either have a zipper with holes of type `Name` or a zipper with holes of type `Salary`. A zipper cannot change mid-traversal from `Name` to `Salary`.

Other work by McBride [23] lifts the restrictions slightly. It has one type for values to the left of the focus and another for values to the right. Still, the left and right types remain fixed for the lifetime of the zipper. It does not address mutually recursive types except to remark that they rapidly reach the limits of the techniques shown.

6.2.2 Functor-based Zippers

Work by Hinze et al. [11] and further explained by Hinze and Jeuring [8] uses Generic Haskell to define a zipper that involves no boilerplate code. It requires the type that the zipper traverses be defined as the fixed-point of a type functor. Furthermore, the holes of the resulting zipper are all the same type as the root object. For example, the `Term` type from Section 2 would need to be defined as in Figure 11. The resulting zipper has holes only of type `Term`. Thus the example from Section 3, which involves both `Salary` and `Name` holes cannot be expressed.

The *MultiRec* framework by Rodriguez Yakushev et al. [28] generalizes the concept of functors to pattern functors that are indexed by types. The type of the hole can be any type listed in the pattern functor. The hole manipulation functions deal with the

```
data Fix f = Fix (f (Fix f))
```

```
data TermF term
  = VarF String
  | LambdaF String term
  | AppF term term
  | IfF term term term
```

```
type Term = Fix TermF
```

Figure 11. A Fixed-point Version of `Term`.

changing type of the hole using a trick similar to that used by `trans` and `query`. Namely, they take as argument a transformer or query function that is parameterized by the index of the type of the hole.

Besides handling heterogeneous types, another advantage of the framework is it does not require the type to be rewritten in terms of a functor. Instead the pattern functor is an auxiliary structure that values are converted into only as needed. The framework makes extensive use of type families [29].

The framework requires boilerplate code as each system of mutually recursive types requires the declaration of a GADT, a type-family instance, and two class instances. The authors ameliorate this by generating most of the boilerplate with Template Haskell, but the user must still list every type that occurs in the pattern functor as a constructor of the GADT that indexes the pattern functor. For example, in Section 3 adding an ID field to the `Employee` type requires adding the ID type to the indexing GADT. Furthermore, because the *MultiRec* framework focuses on *systems* of types, functions written for one system cannot be used for another system even when manipulating only the types that are common to both systems. Accordingly, there are no generic lifting functions like `mkQ` or `mkT`.

Allwood and Eisenbach [3, 4] also implement a zipper for heterogeneous types. It is targeted specifically at the problem of implementing a zipper, unlike the other functor-based zippers in this section, which are presented only as part of general-purpose generic-programming frameworks. The zipper requires a significant amount of boilerplate code for each system of types to be traversed. The code can be generated with Template Haskell, but the user must still list all types in the system of types to be traversed. It thus has many of the same limitations as the *MultiRec* framework.

6.2.3 Unusual Zippers

Kiselyov and Shan [16] take a unique approach to zippers and treat them as delimited continuations of a traversal. These zippers can only move one direction, forward, but other work by Kiselyov [15] extends them to move in multiple directions by directing traversal with a call-back. Both works consider only zippers over homogeneous types, but they should generalize to zippers over heterogeneous types.

Lämmel [19] defines a zipper-like `Context` type that does not include sibling information. It provides context during traversal but is not editable. It operates on heterogeneous types by use of an `Any` type that hides the type of the contained object. Manipulation thus involves a type-safe cast and a run-time type-check.

6.2.4 Summary

There are a number of existing zipper implementations, but those that implement generic zippers all have limitations. Some require a special encoding for types being traversed [5, 23, 25], or that types be written in a particular form [8, 11]. Others are generic over systems of types rather than individual types [3, 4, 28]. Thus the user must maintain a list of every type in the system and manually update it whenever the types change. Furthermore, functions written for one system are not usable in another system. Finally one zipper-like implementation packs everything into an opaque, existential `Any` type and thus uses run-time type casts everywhere [19]. Our generic zipper has none of these limitations.

7. Conclusion

The traditional zipper is a powerful design pattern for representing editable cursors over immutable data, but it has two major limitations. First, it operates only on homogeneous types. Thus while the traditional zipper can represent an abstract syntax tree for the lambda-calculus where everything is an expression, it cannot represent abstract syntax trees of more complicated languages that also include type annotations or statements. Second, it requires a significant amount of boilerplate code. A custom `Context` type and custom movement functions must be written for each type that the zipper traverses, and these definitions are often quite long. For example, the definition of the `Term` type is only five lines of code, but defining its zipper takes thirty-eight lines. This adds significant programming overhead and is a deterrent to using zippers.

The generic zipper presented in this paper has none of these limitations. It operates over any type irrespective of whether it is homogeneous or not. It requires no boilerplate code and integrates within the existing Haskell language without a meta-level system. The only restriction is that the types the zipper traverses must be instances of the `Data` class from the *Scrap your Boilerplate* framework. These instances can be generated automatically by GHC using the deriving mechanism [7]. No other setup is required of the programmer to use the generic zipper.

Finally, the generic zipper is applicable to more than just traditional zipper traversals. Because it is generic, it is also applicable to *generic* traversals of the sort usually provided by generic programming frameworks. Because the generic zipper represents the current position of the traversal as a first-class value, these traversals are often easier to write using the generic zipper. For example, in the *Scrap your Boilerplate* framework, simultaneous traversal of two values takes about one and a half pages to explain [21], but with the generic zipper, it is as simple as using two zippers. Of course, the traditional zipper has long been recognized as a powerful tool for expressing non-generic traversals, so the power that the generic zipper has at expressing generic traversals should come as no surprise. The generic zipper merely makes this power available in the generic case.

Despite the power that a zipper provides, in the past is has perhaps not been used as widely as it could because it requires a significant amount of investment by the programmer to write a custom zipper for each type. Even then it is applicable only when the type is homogeneous. The generic zipper, on the other hand, requires no custom code and operates on any instance of `Data`. With a lower barrier to entry, the generic zipper should allow programmers to use a zipper in cases where previously the programming overhead was too high.

Acknowledgments

Discussions with Perry Alexander and Gerrin Kimmell developed ideas leading to this paper. Comments by Amr Sabry, Amal Ahmed, Daniel P. Friedman, David S. Wise, Ramana Kumar, Lindsey Kuper, and the anonymous reviewers improved the presentation.

References

[1] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. doi: 10.1007/3-540-44904-3_2.

[2] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. ∂ for data: Differentiating data structures. *Fundamenta Informaticae*, 65(1–2): 1–28, February–March 2005.

[3] T. O. R. Allwood and S. Eisenbach. Clase: cursor library for a structured editor. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 123–124, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411302.

[4] T. O. R. Allwood and S. Eisenbach. Strengthening the zipper. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science, pages 2–17, March 2009.

[5] T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 209–257. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-76786-2_4.

[6] J.-P. Bernardy. Lazy functional incremental parsing. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 49–60, New York, NY, USA, 2009. ACM. doi: 10.1145/1596638.1596645.

[7] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2*. The GHC Team. URL `http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/`.

[8] R. Hinze and J. Jeuring. Chapter 2. Generic Haskell: Applications. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–96. Springer Berlin / Heidelberg, 2003. doi: 10.1007/978-3-540-45191-4_2.

[9] R. Hinze and J. Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, November 2001. doi: 10.1017/S0956796801004129.

[10] R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11783596_13.

[11] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45442-X_10.

[12] R. Hinze, A. Löh, and B. C. d. S. Oliveira. "Scrap your boilerplate" reloaded. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11737414_3.

[13] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-76786-2_2.

[14] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. doi: 10.1017/S0956796897002864.

[15] O. Kiselyov. Tool demonstration: A zipper based file/operating system. Presentation at the *2005 ACM SIGPLAN workshop on Haskell*, September 2005. URL `http://okmij.org/ftp/Computation/Continuations.html#zipper-fs`.

[16] O. Kiselyov and C.-c. Shan. Delimited continuations in operating systems. In *Modeling and Using Context*, volume 4635 of *Lecture Notes in Computer Science*, pages 291–302. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-74255-5_22.

[17] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM. doi: 10.1145/1017472.1017488.

[18] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*, page 562. Addison-Wesley, 1st edition, 1968. ISBN 0-201-03801-3.

[19] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 137–142, New York, NY, USA, 2007. ACM. doi: 10.1145/1190216.1190240.

[20] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM. doi: 10.1145/604174.604179.

[21] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255, New York, NY, USA, 2004. ACM. doi: 10.1145/1016850.1016883.

[22] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215, New York, NY, USA, 2005. ACM. doi: 10.1145/1086365.1086391.

[23] C. McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, New York, NY, USA, 2008. ACM. doi: 10.1145/1328438.1328474.

[24] C. McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001. URL `http://strictlypositive.org/diff.pdf`.

[25] P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11617990_16.

[26] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM. doi: 10.1145/1159803.1159811.

[27] A. Rodriguez Yakushev, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411301.

[28] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 233–244, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596585.

[29] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM. doi: 10.1145/1411204.1411215.

[30] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. doi: 10.1145/581690.581691.

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