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.

© ACM, 2010. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in WGP ’09: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming, (September 2010). http://doi.acm.org/10.1145/1863496.1863499

WGP ’10, September 26, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0251-7/10/09...$10.00
downTerm :: TermZipper → Maybe TermZipper
downTerm (Var s c) = Nothing
downTerm (Lambda s t1 c) = Just (t1, Lambda s c)
downTerm (App t1 t2 c) = Just (t1, App1 c t2)
downTerm (if t1 t2 t3 c) = Just (t1, if1 c t2 t3)

upTerm :: TermZipper → Maybe TermZipper
upTerm (t1, RootTerm) = Nothing
upTerm (t1, Lambda s c) = Nothing
upTerm (t1, App1 c t2) = Just (t1, App c t2)
upTerm (t1, If1 c t2 t3) = Just (t1, If c t2 t3)
upTerm (t1, t2 t3 t4) = Just (t1, t2 t3 t4)

leftTerm :: TermZipper → Maybe TermZipper
leftTerm (t1, RootTerm) = Nothing
leftTerm (t1, Lambda s c) = Nothing
leftTerm (t1, App1 c t2) = Nothing
leftTerm (t1, If1 c t2 t3) = Just (t1, If c t2 t3)
leftTerm (t1, t2 t3 t4) = Just (t1, t2 t3 t4)

rightTerm :: TermZipper → Maybe TermZipper
rightTerm (t1, RootTerm) = Nothing
rightTerm (t1, Lambda s c) = Nothing
rightTerm (t1, App1 c t2) = Just (t1, App2 c t2)
rightTerm (t1, If1 c t2 t3) = Just (t1, If2 c t2 t3)
rightTerm (t1, t2 t3 t4) = Just (t1, t2 t3 t4)

fromZipperTerm :: TermZipper → Term
fromZipperTerm z = f z where
  f :: TermZipper → Term
  f (t1, RootTerm) = t1
  f (t1, Lambda s c) = f (Lambda s t1, c)
  f (t1, App1 c t2) = f (App t1 t2, c)
  f (t2, App2 t1 c) = f (App t2 t1, c)
  f (t1, If1 c t2 t3) = f (If t1 t2 t3, c)
  f (t1, If2 c t2 t3) = f (If t1 t2 t3, c)
  f (t2, t3 t4) = f (If t2 t3 t4, c)

toZipperTerm :: Term → TermZipper
toZipperTerm t = (t, RootTerm)

getHoleTerm :: TermZipper → Term
getHoleTerm (t, _) = t

set HoleTerm :: Term → TermZipper → TermZipper
set HoleTerm h (_, c) = (h, c)

Figure 2. Zipper Non-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
```

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 = RootTerm
                   | Lambda s String TermContext
                   | App1 Term TermContext
                   | App2 Term TermContext
                   | If1 Term TermContext
                   | If2 Term TermContext
                   | If3 Term TermContext
```

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 RootTerm 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 `downTerm, upTerm, leftTerm and rightTerm` 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 left-most 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 `fromZipperTerm`, these are trivial wrapper functions manipulating the pair that forms a zipper. The `fromZipperTerm` function moves all the way to the root context before returning the resulting value.

### 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 `toZipperTerm` 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`.

```latex
\begin{verbatim}
*Main> let t0 = toZipperTerm fac
*Main> getHoleTerm 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"))))))
\end{verbatim}
```

The first step to getting to the offending `*` is to use `downTerm`. Now the hole contains the body of the Lambda.

```latex
\begin{verbatim}
*Main> let Just t1 = downTerm t1
*Main> getHoleTerm t1
If (App (App (Var "=") (Var "n")) (Var "0"))
    (Var "1")
    (App (App (Var "+") (Var "n"))
        (App (Var "fac"
            (App (Var "pred") (Var "n"))))))
\end{verbatim}
```

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 `rightTerm` twice.

```latex
\begin{verbatim}
*Main> let Just t2 = downTerm t2
*Main> getHoleTerm t2
App (Var "=") (Var "n") (Var "0")
\end{verbatim}
```

```latex
\begin{verbatim}
*Main> let Just t3 = rightTerm t2
*Main> getHoleTerm t3
Var "1"
\end{verbatim}
```

```latex
\begin{verbatim}
*Main> let Just t4 = rightTerm t3
*Main> getHoleTerm t4
App (Var "+") (Var "n")
    (App (Var "fac"
        (App (Var "pred") (Var "n"))))
\end{verbatim}
```

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

```latex
\begin{verbatim}
*Main> let Just t5 = downTerm t4
*Main> getHoleTerm t5
App (Var "+") (Var "n")
\end{verbatim}
```

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

```latex
\begin{verbatim}
*Main> let Just t6 = downTerm t5
*Main> getHoleTerm t6
Var "*"
\end{verbatim}
```

Finally, `setHoleTerm` changes the value.

```latex
\begin{verbatim}
*Main> let t7 = setHoleTerm (Var "*") t6
\end{verbatim}
```

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

```latex
\begin{verbatim}
*Main> let Just t8 = upTerm t7
*Main> getHoleTerm t8
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"))))))
\end{verbatim}
```

Or we can use `fromZipperTerm` to go directly to the root and get the corrected definition.

```latex
\begin{verbatim}
*Main> fromZipperTerm t7
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"))))))
\end{verbatim}
```

### 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:

```latex
\begin{verbatim}
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
\end{verbatim}
```

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
fromZipper :: Zipper a → 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 follow-
ing department:

department :: Dept
department = D agamemnon [menelaus, achilles, odysseus]

agamemnon, menelaus,
achilles, odysseus :: Employee
agamemnon = E "Agamemnon" 5000.0
menelaus = E "Menelaus" 3000.0
achilles = E "Achilles" 2000.0
odysseus = E "Odysseus" 2000.0

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 department
*Main> :type g1  
g1 :: Zipper Dept

We would like to inspect the hole value, but first we must un-
derstand how the generic zipper deals with the type of the hole.
With a traditional zipper, the type of the hole is fixed. For example,
gGetHoleTerm 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 Boiler-
plate framework. It has the following type, where the Typeable
class is a superclass of the Data class. It returns Nothing to indi-
cate 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))

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

gGetHole :: (Typeable b) => Zipper a → Maybe b

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

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
gmap, 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 left-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 the current node 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 Name

Just "Agamemnon"
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 \texttt{trans} function, which applies a generic transformer to the hole. To construct a generic transformer for this example, we again borrow a function from \textit{Scrap your Boilerplate}.

\begin{verbatim}
setHole :: (Typeable a) => a \rightarrow Zipper b \rightarrow Zipper b
setHole h z = trans (mkT (const h)) z
\end{verbatim}

This function leaves the hole unchanged if it is not of type \texttt{a}. While we are giving the king his proper title, let's also move to the right and give the king a raise.

\begin{verbatim}
leftQ Nothing getHole g6 :: Maybe Salary
rightQ Nothing getHole g6 :: Maybe Salary
\end{verbatim}

\subsection{Sibling Movement}

These functions apply their \texttt{f} argument to the result of a movement, but only if the movement is legal. Otherwise, they return their \texttt{b} argument. For example, the following returns \texttt{False} as there is no sibling to the left of \texttt{g6}.

\begin{verbatim}
leftQ False (const True) g6
\end{verbatim}

On the other hand, the following returns \texttt{Just 8000.0} as the right sibling of \texttt{g6} is the king's salary.

\begin{verbatim}
rightQ Nothing getHole g6 :: Maybe Salary
\end{verbatim}

Although the generic zipper merely makes it possible for the traversal to be generic.

\subsection{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 \texttt{Maybe} returned by the zip movement functions. These are defined in terms of the core zip movement primitives, so they do not change the expressive power of the generic zipper.

\begin{verbatim}
queryM :: t -> Maybe t
queryM = \textbf{either} Nothing (fmap (\_ \rightarrow Just \_))
\end{verbatim}

\subsection{Query Movement}

These functions apply their \texttt{f} argument to the result of a movement, but only if the movement is legal. Otherwise, they return their \texttt{b} argument. For example, the following returns \texttt{False} as there is no sibling to the left of \texttt{g6}.

\begin{verbatim}
leftQ False (const True) g6
\end{verbatim}

On the other hand, the following returns \texttt{Just 8000.0} as the right sibling of \texttt{g6} is the king's salary.

\begin{verbatim}
rightQ Nothing getHole g6 :: Maybe Salary
\end{verbatim}

\subsection{Transformer Movement}

These functions extend query movement by replacing the hole value with the result of \texttt{f} and then moving the zipper back to its original position. If the movement is illegal, they leave the hole unchanged. Thus when changing Agamemmon's ... ahem ... King Agamemnon's salary, we could leave the \texttt{g6} zipper focused on his title and skip the intermediate \texttt{g7} step:

\begin{verbatim}
leftQ False (const True) g6
\end{verbatim}

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 \texttt{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 \texttt{Maybe} type.

\section{Generic Traversals with Zippers}

In the preceding section, the zipper traversed over a specific type, namely \texttt{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.
4.3 Generic Outermost-Leftmost Reduction

The bottom-up traversal that \textit{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 \textit{zeverywhere}' function in Figure 6 implements the first step. It differs from \textit{zeverywhere} in that it is top-down and left-to-right, but more importantly, it explicitly controls movement up the zipper instead of letting \textit{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 \textit{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, \textit{leftmost} starts the traversal at the leftmost child. This is so \textit{f} can signal whether it succeeded at applying a reduction to the current node. Because of this change, \textit{zreduce} uses \textit{transM} instead of \textit{trans}. The \textit{transM} function lifts the \textit{Maybe} value so that when \textit{f} returns \textit{Nothing}, \textit{transM} also returns \textit{Nothing}, and when \textit{f} returns \textit{Just}, \textit{transM} also returns \textit{Just}.

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 \textit{zreduce} in Figure 7. In \textit{zeverywhere'}, \textit{f} has type:

\[
\forall a. \text{Data} \ a \Rightarrow a \rightarrow a
\]

But in \textit{zreduce}, \textit{f} has type:

\[
\forall a. \text{Data} \ a \Rightarrow a \rightarrow \text{Maybe} \ a
\]

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

When \textit{f} cannot apply a reduction, it returns \textit{Nothing}. In that case, \textit{zreduce} continues exactly the same as \textit{zeverywhere'}.

Figure 5. Traversal Helper Functions.

<table>
<thead>
<tr>
<th>Query Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftQ, rightQ, downQ, upQ ::</td>
</tr>
<tr>
<td>b \cdot (Zipper a \rightarrow b) \rightarrow Zipper a \rightarrow Zipper a</td>
</tr>
<tr>
<td>leftQ b f z = moveQ left b f z</td>
</tr>
<tr>
<td>rightQ b f z = moveQ right b f z</td>
</tr>
<tr>
<td>downQ b f z = moveQ down b f z</td>
</tr>
<tr>
<td>upQ b f z = moveQ up b f z</td>
</tr>
<tr>
<td>moveQ move b f z = case move z of</td>
</tr>
<tr>
<td>Nothing \rightarrow b</td>
</tr>
<tr>
<td>Just z \rightarrow f z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transformer Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftT, rightT, downT, upT ::</td>
</tr>
<tr>
<td>(Zipper a \rightarrow Zipper a) \rightarrow Zipper a \rightarrow Zipper a</td>
</tr>
<tr>
<td>leftT f z = moveT left right z f z</td>
</tr>
<tr>
<td>rightT f z = moveT right left z f z</td>
</tr>
<tr>
<td>downT f z = moveT down up f z f z</td>
</tr>
<tr>
<td>upT f z = g z</td>
</tr>
<tr>
<td>g z = moveT right left (h z) g z</td>
</tr>
<tr>
<td>h z = moveT up down z f z</td>
</tr>
<tr>
<td>moveT move1, move2 b f z =</td>
</tr>
<tr>
<td>moveQ move1 b (moveQ move2 b id \cdot f) z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sibling Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftmost, rightmost :: Zipper a \rightarrow Zipper a</td>
</tr>
<tr>
<td>leftmost z = leftQ z leftmost z</td>
</tr>
<tr>
<td>rightmost z = rightQ z rightmost z</td>
</tr>
</tbody>
</table>

Figure 6. Bottom-up and Top-down Zipper Traversals.
As with a traditional zipper, the

existentially quantified
data
hole and the zipper’s root type.

root

represents a one-hole context that contains a hole of type

Context

hole

. Except when it is the top-most

root

, 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.

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 =

\( \forall \) 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

NullCtx

, it contains a set of left siblings, a set of right siblings, and its parent context:

data Context hole root where

\( \text{NullCtx} \) :: Context a a

\( \text{ConsCtx} \) :: Left \( \ldots \) \( \rightarrow \) Right \( \ldots \) \( \rightarrow \) Context \( \ldots \)

\( \rightarrow \) 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

| \( \forall \) b. (Data b) => ConsLeft (Left (b \( \rightarrow \) expects)) b

The key to understanding this type is

b

, the existentially quantified type variable in

ConsLeft

. The first argument of

ConsLeft

is a

Left

that represents a partially applied constructor of type

b \( \rightarrow \) expects.

This is packaged up with a second argument of type

b

. This packaging represents the application of the former to the latter to construct 

an object of type

expects.

Because

ConsLeft

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

UnitLeft

.

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

\( \text{UnitLeft} \) Bar

. The type of this value is:

UnitLeft Bar :: Left (Int \( \rightarrow \) Char \( \rightarrow \) Foo)

The arguments that

Bar

is expecting are manifest in the type of the resulting

Left

object. We can add those arguments with

ConsLeft

, and the way

ConsLeft

is defined ensures that those arguments are of the proper type.

*Main> :type UnitLeft Bar

\( \text{ConsLeft} \) \( \_ \_ \) 1

\( \_ \_ \) :: Left (Char \( \rightarrow \) Foo)

*Main> :type UnitLeft Bar

\( \text{ConsLeft} \) \( \_ \_ \) 1

\( \_ \_ \) :: Left (Float \( \rightarrow \) Foo)

*Main> :type UnitLeft Baz

\( \text{ConsLeft} \) \( \_ \_ \) \'a'

\( \_ \_ \) :: Left Foo

*Main> :type UnitLeft Baz

\( \text{ConsLeft} \) \( \_ \_ \) 1.0

\( \_ \_ \) :: Left Foo

In short, Left contains a value of existentially quantified type

b

provided

b

matches the argument type expected by the constructor.

\(^1\) GHC uses the \( \forall \) keyword for both existential and universal types. The distinction between the two is where the keyword is positioned.

\(^2\) Readers familiar with the

Frame your Boilerplate: Reloaded

framework [10, 12] may recognize this type as

Spine

.
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
   NullR :: Right parent parent
   ConsR ::
      (Data b) => b + Right a t + Right (b + a) t

The NullR 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 ConsR. 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 NullR:

   *Main> :type NullR
   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 ConsR 'a' NullR
   it :: Right (Char + a) a

Next the preceding argument to Bar is added:

   *Main> :type ConsR 1 (ConsR 'a' NullR)
   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 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
   NullC :: Context a a
   ConsC ::
      ∀ 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 ConsC 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 ConsL off of the left part of the context and adds a ConsR onto the right part of the context. The right function does the reverse. Similarly, the up function pulls a ConsC 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 _ (ConslCtxt (UniCtxt _ _))) = Nothing
left (Zipper h (ConslCtxt (Consleft l h') r c)) =
    Just (Zipper h' (ConslCtxt l (Consrights h r) c))

right (Zipper _ NullCtxt) = Nothing
right (Zipper _ (ConsrCtxt _ Nullrights _)) = Nothing
right (Zipper h (ConsrCtxt l (Consrights h' r) c)) =
    Just (Zipper h' (ConsrCtxt (Consright l h) r c))

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

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)

fromZipper (Zipper hole NullCtxt) = hole
fromZipper (Zipper hole (ConslCtxt l r ctxt)) =
    fromZipper (Zipper (combine l hole r) ctxt)

toZipper x = Zipper x NullCtxt

Figure 9. Generic Zipper Movement.

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 reapplyes 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 reappplies 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 ConslLeft UniLeft a

For example toLeft (Bar 5 'd') results in the value:
(Unitleft Bar) 'ConslLeft' 5 'ConslLeft' 'd'

fromZipper (Zipper hole NullCtxt) = hole
fromZipper (Zipper hole (ConslCtxt l r ctxt)) =
    fromZipper (Zipper (combine l hole r) ctxt)

toZipper x = Zipper x NullCtxt

Figure 10. Generic Zipper Non-Movement Operations.

The down function is implemented by injecting the hole into a
Left with toLeft and extracting its rightmost element. This right-
most 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 down-
ward 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' (ConslCtxt l Nullrights ctxt))

The use of gfoldl is the source of two peculiarities in the imple-
mentation. 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 ConslLeft
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 pro-
gramming and generic types [13, 27]. Here we focus on three sys-
tems 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 simultane-
ous 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.
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 applied 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.

6.2.2 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 to 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` 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 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 `map` 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 sys-
tems 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 it 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

POPL ’07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT 
symposium on Principles of programming languages, pages 137–142, New 

[20] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical de-
sign 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. 

[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, 
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, 
1086365.1086391.

[23] C. McBride. Clowns to the left of me, jokers to the right (pearl): dis-
secting 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-
strictlypositive.org/diff.pdf.

types. In Types for Proofs and Programs, volume 3839 of Lecture 
Notes in Computer Science, pages 252–267. Springer Berlin / Heidel-

unification-based type inference for GADTs. In ICFP ’06: Proceed-
ings of the eleventh ACM SIGPLAN international conference on Func-
tional programming, pages 50–61, New York, NY, USA, 2006. ACM. 

[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 

programming with fixed points for mutually recursive datatypes. In ICFP ’09: Proceedings of the 14th ACM SIGPLAN international 

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.

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 
unsw.edu.au/~dons/blog/2007/05/17.