The Zipper

Joel Wright

Introduction

This presentation introduces the Zipper in the context of a simple binary search tree, but...

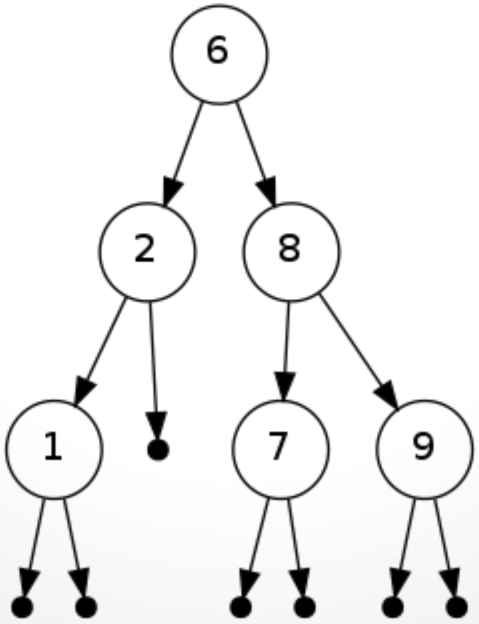
the Zipper can be applied to any inductively defined data type.

It's useful when you want to maintain the structure of a type, but have O(1) access to an arbitrary focal point, e.g.

- Editors
- XMonad Window Lists

A Simple Binary Search Tree

```haskell
data Tree a = Leaf
            | Node a (Tree a) (Tree a)
```



Adding a Focus of Interest

Now we need to add a focus of interest to our tree. For this, we'll need a way of describing a path to the node we're currently interested in:

```
data Direction   = L | R
type Directions  = [Direction]
```

and a new type of tree that combines our previous definition with a path to the node:

```
type FocusTree a = (Tree a, Directions)
```

Modifying the Tree

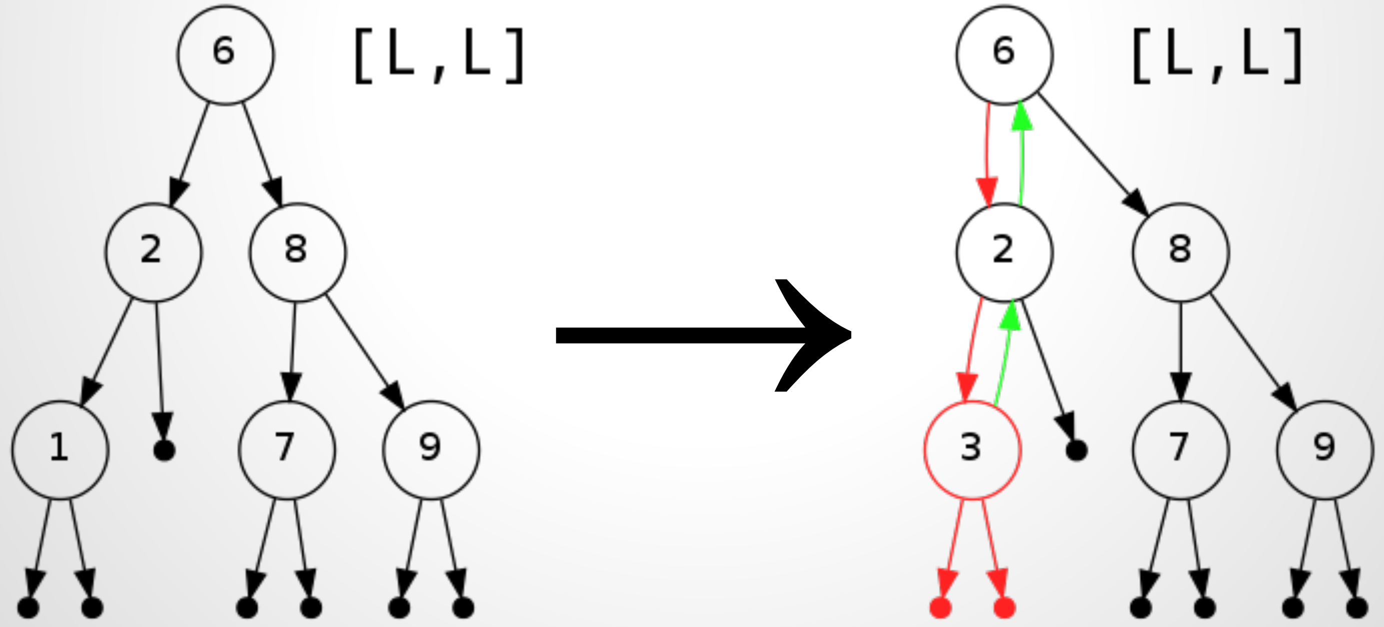
```haskell
modifyFocus :: FocusTree a → (Tree a → Tree a) → FocusTree a
modifyFocus (t, ds) f = let t' = modifyF' t ds f
                         in t' `seq` (t',ds)


modifyF' :: Tree a → Directions → (Tree a → Tree a) → Tree a
modifyF' t [] f = f t
modifyF' (Node x l r) (L:ds) f = let l' = modifyF' l ds f
                                  in l' `seq` (Node x l' r)
modifyF' (Node x l r) (R:ds) f = let r' = modifyF' r ds f
                                  in r' `seq` (Node x l r')
modifyF' _ _ _ = error "Can't move down from a Leaf"
```

The `seq` calls are a necessary optimisation for the demo code in the repository, and we've just had a presentation on performance and optimisation, so I thought I'd leave them in. Without them we get a stack overflow on large numbers (>500,000) of tree modifications.

Problem with this Approach

Every time we want to inspect or modify the focus of interest we must traverse the Tree.



How can we do better?

We can't avoid the cost of modifying the bit of the Tree that we're interested in, but can we avoid the cost of getting to it?

That's where the idea of a Zipper comes in...

Keeping Things in Context

First, let's throw away the simple idea of "directions" to our focal point. Instead, let's build a data type that keeps track of our movements around the tree, and crucially, the bits of the tree we chose **NOT** to look at:

```
data Context a = CRoot
               | CLeft (Context a) (a,(Tree a))
               | CRight (Context a) (a,(Tree a))

type ZipTree a = (Context a, Tree a)

mkZipTree :: Tree a -> ZipTree a
mkZipTree t = (CRoot, t)
```

Moving around the Zipper Tree

```
left :: ZipTree a → ZipTree a
left (c, Node x l r) = (CLeft c (x,r), l)
left _ = error "Can't move down from a Leaf"

right :: ZipTree a → ZipTree a
right (c, Node x l r) = (CRight c (x,l), r)
right _ = error "Can't move down from a Leaf"

up :: ZipTree a → ZipTree a
up (CLeft c (x,r), t) = (c, Node x t r)
up (CRight c (x,l), t) = (c, Node x l t)
up _ = error "Can't go up from the root of the Tree"

top :: ZipTree a → ZipTree a
top z@(CRoot, t) = z
top z = top.up $ z
```

Modifying the Zipper Tree

```
modify :: ZipTree a → (Tree a → Tree a) → ZipTree a
modify (c, t) f = let t' = f t
                   in t' `seq` (c, t')
```

What have we achieved?

Performance improvements :)

We've had some fun with data types

Further Reading

All the code and slides are available here:

`https://github.com/joelwright/ZipperHS`

Lots of further reading and the results of research following a similar theme:

- "The Zipper", Gerard Huet
- "Weaving a Web", Ralf Hinze & Johan Jeuring
- "Scrap Your Zippers", Michael D. Adams
- Haskell Wiki Page, http://www.haskell.org/haskellwiki/Zipper
- Haskell/Zippers (Wikibooks) http://en.wikibooks.org/wiki/Haskell/Zippers
- Chapter 14 in "Learn You a Haskell for Great Good!"

And to really mess with your mind, try reading:

"The Derivative of a Regular Type is its Type of One-Hole Contexts", and related works by Conor McBride.