Zippers

Neil Sculthorpe

Functional Programming Group
Information and Telecommunication Technology Center
University of Kansas
neil@ittc.ku.edu

EECS 776
Lawrence, Kansas
28th November 2012

What are Zippers?

What are Zippers?

- Zippers are nothing to do with:
  - Haskell's *zip* and *zipWith* functions, or the *ZipList* data type;

What are Zippers?

- Zippers are nothing to do with:
  - Haskell's *zip* and *zipWith* functions, or the *ZipList* data type;
  - the Zip archive file format.

What are Zippers?

- Zippers are nothing to do with:
    - Haskell's *zip* and *zipWith* functions, or the *ZipList* data type;
    - the Zip archive file format.
- A Zipper is a data structure with a focal point.

What are Zippers?

- Zippers are nothing to do with:
  - Haskell's *zip* and *zipWith* functions, or the *ZipList* data type;
  - the Zip archive file format.
- A Zipper is a data structure with a focal point.
  - Operations can be applied efficiently at the focal point.

What are Zippers?

- Zippers are nothing to do with:
  - Haskell's *zip* and *zipWith* functions, or the *ZipList* data type;
  - the Zip archive file format.
- A Zipper is a data structure with a focal point.
  - Operations can be applied efficiently at the focal point.
  - The focal point can be moved efficiently.

The Problem

- Sometimes we want to operate on:
  - a sub-structure of a data structure (e.g. the last 5 elements of a list)
  - the same sub-structure repeatedly
  - adjacent sub-structures (e.g. the last 5 elements, then the last 6)

The Problem

- Sometimes we want to operate on:
  - a sub-structure of a data structure (e.g. the last 5 elements of a list)
  - the same sub-structure repeatedly
  - adjacent sub-structures (e.g. the last 5 elements, then the last 6)
- In a purely functional setting with immutable data, we could define:

$$\begin{aligned}&\mathit{modifySuffix} :: \mathit{Int} \to ([a] \to [a]) \to [a] \to [a]\\&\mathit{modifySuffix}\ n\ f\ \mathit{as} = \mathbf{let}\ (\mathit{bs}, \mathit{cs}) = \mathit{splitAt}\ n\ \mathit{as}\\&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \mathit{bs} +\!\!+\ f\ \mathit{cs}\end{aligned}$$

The Problem

- Sometimes we want to operate on:
  - a sub-structure of a data structure (e.g. the last 5 elements of a list)
  - the same sub-structure repeatedly
  - adjacent sub-structures (e.g. the last 5 elements, then the last 6)

- In a purely functional setting with immutable data, we could define:

$$\begin{aligned}
&\mathit{modifySuffix} :: \mathit{Int} \to ([a] \to [a]) \to [a] \to [a] \\
&\mathit{modifySuffix}\ n\ f\ \mathit{as} = \textbf{let}\ (\mathit{bs}, \mathit{cs}) = \mathit{splitAt}\ n\ \mathit{as} \\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{bs} +\!\!+\ f\ \mathit{cs}
\end{aligned}$$

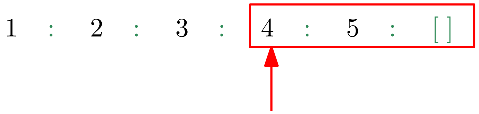
but this is inefficient as it traverses the list each time it is used.

A Solution

A Solution

- In an imperative setting with mutable data, we might:
  - maintain a pointer to the sub-structure of interest
  - use a data type with back-pointers (e.g. doubly linked lists) to move to adjacent sub-structures

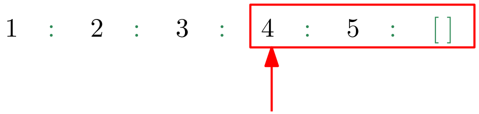
  thereby avoiding inefficient traversals.



A Solution

- In an imperative setting with mutable data, we might:
    - maintain a pointer to the sub-structure of interest
    - use a data type with back-pointers (e.g. doubly linked lists) to move to adjacent sub-structures

  thereby avoiding inefficient traversals.

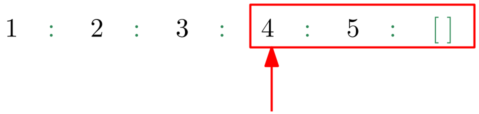


- Zippers are a way to do this in a purely functional setting.

A Solution

- In an imperative setting with mutable data, we might:
  - maintain a pointer to the sub-structure of interest
  - use a data type with back-pointers (e.g. doubly linked lists) to move to adjacent sub-structures

  thereby avoiding inefficient traversals.



- Zippers are a way to do this in a purely functional setting.
- A Zipper consists of:
  - the sub-structure of interest
  - a context containing everything else we need to reconstruct the original structure

A Zipper for Lists

A Zipper for Lists

The List Zipper Data Type

type $\mathit{ListZipper}\ a = (\mathit{ListContext}\ a, [a])$

type $\mathit{ListContext}\ a = [a]$

A Zipper for Lists

The List Zipper Data Type

$\textbf{type}\ \mathit{ListZipper}\ a = (\mathit{ListContext}\ a, [a])$

$\textbf{type}\ \mathit{ListContext}\ a = [a]$

Moving the Focal Point

$\mathit{forward} :: \mathit{ListZipper}\ a \to \mathit{ListZipper}\ a$
$\mathit{forward}\ (\mathit{ctx}, (a : \mathit{as})) = ((a : \mathit{ctx}), \mathit{as})$

$\mathit{backward} :: \mathit{ListZipper}\ a \to \mathit{ListZipper}\ a$
$\mathit{backward}\ ((a : \mathit{ctx}), \mathit{as}) = (\mathit{ctx}, (a : \mathit{as}))$

A Zipper for Lists

The List Zipper Data Type

$\textbf{type}\ \mathit{ListZipper}\ a = (\mathit{ListContext}\ a, [a])$

$\textbf{type}\ \mathit{ListContext}\ a = [a]$

Moving the Focal Point

$\mathit{forward} :: \mathit{ListZipper}\ a \to \mathit{ListZipper}\ a$
$\mathit{forward}\ (\mathit{ctx}, (a : \mathit{as})) = ((a : \mathit{ctx}), \mathit{as})$

$\mathit{backward} :: \mathit{ListZipper}\ a \to \mathit{ListZipper}\ a$
$\mathit{backward}\ ((a : \mathit{ctx}), \mathit{as}) = (\mathit{ctx}, (a : \mathit{as}))$

Operating at the Focal Point

$\mathit{modify} :: ([a] \to [a]) \to \mathit{ListZipper}\ a \to \mathit{ListZipper}\ a$
$\mathit{modify}\ f\ (\mathit{ctx}, \mathit{as}) = (\mathit{ctx}, f\ \mathit{as})$

Binary Trees

A Binary Tree Data Type

data *Tree a* = Branch (*Tree a*) (*Tree a*) | Leaf *a*

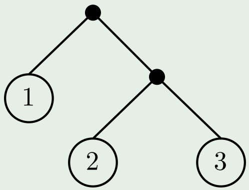
Binary Trees

A Binary Tree Data Type

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Example

```
tree1 :: Tree Int
tree1 = Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```



A Zipper for Binary Trees

A Zipper for Binary Trees

The Zipper Data Type

type $TreeZipper\ a = (TreeContext\ a, Tree\ a)$

type $TreeContext\ a = [(Direction, Tree\ a)]$

data $Direction = \mathsf{L} \mid \mathsf{R}$

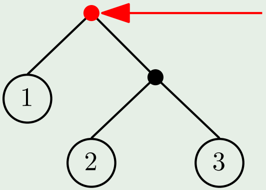
A Zipper for Binary Trees

The Zipper Data Type

```
type TreeZipper a = (TreeContext a, Tree a)
type TreeContext a = [(Direction, Tree a)]
data Direction = L | R
```

Examples

```
treeZipper1 :: TreeZipper Int
treeZipper1 = ([], Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)))
```



A Zipper for Binary Trees

The Zipper Data Type

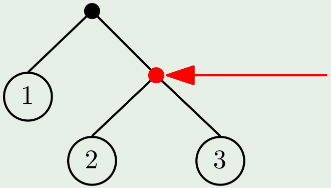
type *TreeZipper a* = (*TreeContext a*, *Tree a*)

type *TreeContext a* = [(*Direction*, *Tree a*)]

data *Direction* = L | R

Examples

treeZipper2 :: *TreeZipper Int*
treeZipper2 = ([(R, Leaf 1)], Branch (Leaf 2) (Leaf 3))



A Zipper for Binary Trees

The Zipper Data Type

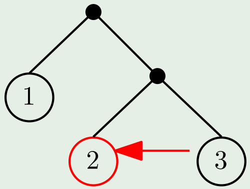
type *TreeZipper a* = (*TreeContext a*, *Tree a*)

type *TreeContext a* = [(*Direction*, *Tree a*)]

data *Direction* = L | R

Examples

treeZipper3 :: *TreeZipper Int*
treeZipper3 = ([(L, Leaf 3), (R, Leaf 1)], Leaf 2)



A Zipper for Binary Trees

The Zipper Data Type

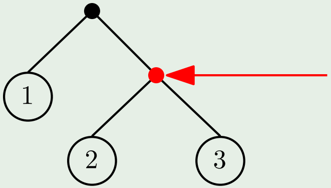
type *TreeZipper a* = (*TreeContext a*, *Tree a*)

type *TreeContext a* = [(*Direction*, *Tree a*)]

data *Direction* = L | R

Examples

treeZipper2 :: *TreeZipper Int*
treeZipper2 = ([(R, Leaf 1)], Branch (Leaf 2) (Leaf 3))



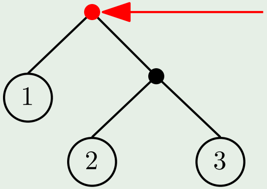
A Zipper for Binary Trees

The Zipper Data Type

type *TreeZipper a* = (*TreeContext a*, *Tree a*)
type *TreeContext a* = [(*Direction*, *Tree a*)]
data *Direction* = L | R

Examples

treeZipper1 :: *TreeZipper Int*
treeZipper1 = ([ ], Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)))



A Zipper for Binary Trees

The Zipper Data Type

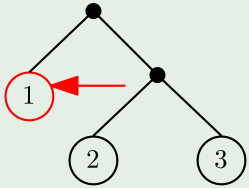
type $TreeZipper\ a = (TreeContext\ a, Tree\ a)$

type $TreeContext\ a = [(Direction, Tree\ a)]$

data $Direction =$ L | R

Examples

$treeZipper4 :: TreeZipper\ Int$
$treeZipper4 = ([($L, Branch (Leaf 2) (Leaf 3))], Leaf 1)$



A Zipper for Binary Trees

The Zipper Data Type

$$\textbf{type}\ \mathit{TreeZipper}\ a = (\mathit{TreeContext}\ a, \mathit{Tree}\ a)$$

$$\textbf{type}\ \mathit{TreeContext}\ a = [(\mathit{Direction}, \mathit{Tree}\ a)]$$

$$\textbf{data}\ \mathit{Direction} = \mathrm{L} \mid \mathrm{R}$$

Moving the Focal Point

$$\mathit{left} :: \mathit{TreeZipper}\ a \rightarrow \mathit{TreeZipper}\ a$$
$$\mathit{left}\ (\mathit{ctx}, \mathrm{Branch}\ l\ r) = (((\mathrm{L}, r) : \mathit{ctx}), l)$$

$$\mathit{right} :: \mathit{TreeZipper}\ a \rightarrow \mathit{TreeZipper}\ a$$
$$\mathit{right}\ (\mathit{ctx}, \mathrm{Branch}\ l\ r) = (((\mathrm{R}, l) : \mathit{ctx}), r)$$

$$\mathit{up} :: \mathit{TreeZipper}\ a \rightarrow \mathit{TreeZipper}\ a$$
$$\mathit{up}\ (((\mathrm{L}, r) : \mathit{ctx}), l) = (\mathit{ctx}, \mathrm{Branch}\ l\ r)$$
$$\mathit{up}\ (((\mathrm{R}, l) : \mathit{ctx}), r) = (\mathit{ctx}, \mathrm{Branch}\ l\ r)$$

A Zipper for Binary Trees

The Zipper Data Type

type *TreeZipper a* = (*TreeContext a*, *Tree a*)

type *TreeContext a* = [(*Direction*, *Tree a*)]

data *Direction* = L | R

Operating at the Focal Point

$modifyTree :: (Tree\ a \rightarrow Tree\ a) \rightarrow TreeZipper\ a \rightarrow TreeZipper\ a$
$modifyTree\ f\ (ctx, t) = (ctx, f\ t)$

Summary

- A Zipper is a data structure with a focal point.
- The purpose of a Zipper is to support efficient operations on immutable data types.
- Zippers can be defined for any algebraic data type.

Exercises (optional)

1. Define a Zipper for the following data type:

   $$\textbf{data}\ \mathit{BTree}\ a = \mathsf{Node}\ (\mathit{BTree}\ a)\ a\ (\mathit{BTree}\ a)\ |\ \mathsf{Empty}$$

2. Add modification and movement functions.

3. Define the following functions:
   - $\mathit{ancestors} :: \mathit{BTreeZipper}\ a \to [a]$
     that returns the values of all nodes above the focal point
   - $\mathit{top} :: \mathit{BTreeZipper}\ a \to \mathit{BTreeZipper}\ a$
     that navigates to the top of the tree
   - $\mathit{zipperToTree} :: \mathit{BTreeZipper}\ a \to \mathit{BTree}\ a$
     that converts a Zipper into a tree (by forgetting the focal point)