[CSE301 / Lecture 6]
Zippers and derivatives of data types

Noam Zeilberger

Ecole Polytechnique

9 October 2024

Motivation

Many efficient algorithms use pointers to navigate within a data structure and perform destructive updates, e.g.:

```
# ... removing node w/two children from a BST
parent, current = node, node.right
while current.left is not None:
    parent, current = current, current.left
if parent is node:
    parent.right = current.right
else:
    parent.left  = current.right
node.value = current.value
```

Can we perform such manipulations in a purely functional way?

What is a "zipper"?[1]

A **zipper** is a pair of a *value* and a *one-hole context*.

Idea: represent a pointer by the value being pointed to, together with a representation of its surrounding context.

There is some beautiful theory behind this idea, which is also very useful in practice (e.g., used in the XMonad window manager!)

[1] The terminology comes from a 1997 article by Gérard Huet in the *Journal of Functional Programming*. "Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name."

Example #1: a functional line editor using list zippers

How can we represent a “cursor” into a line of text, in a way that supports both efficient navigation and editing?

Example #1: a functional line editor using list zippers

To make this question concrete, let's introduce a class of cursor types with some expected operations.

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Cursor\ cur}\ \textbf{where} & \\
\quad \mathit{cursor} :: [a] \to \mathit{cur}\ a & \text{-- create a cursor to head of list} \\
\quad \mathit{value} :: \mathit{cur}\ a \to a & \text{-- return value at cursor} \\
\quad \mathit{list} :: \mathit{cur}\ a \to [a] & \text{-- return the underlying list} \\
\quad \mathit{movL} :: \mathit{cur}\ a \to \mathit{cur}\ a & \text{-- move the cursor to the left} \\
\quad \mathit{movR} :: \mathit{cur}\ a \to \mathit{cur}\ a & \text{-- move the cursor to the right} \\
\quad \mathit{ovw} :: a \to \mathit{cur}\ a \to \mathit{cur}\ a & \text{-- overwrite item with new value} \\
\quad \mathit{bks} :: \mathit{cur}\ a \to \mathit{cur}\ a & \text{-- remove the item to the left} \\
\quad \mathit{ins} :: a \to \mathit{cur}\ a \to \mathit{cur}\ a & \text{-- insert a new item to the left}
\end{array}
$$

Example #1: a functional line editor using list zippers

Naive solution: store the index of the element in focus.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{NaiveCursor}\ a = \mathit{NC}\ ([a], \mathit{Int}) \\
\textbf{instance}\ \mathit{Cursor}\ \mathit{NaiveCursor}\ \textbf{where} \\
\quad \mathit{cursor}\ \mathit{xs} = \mathit{NC}\ (\mathit{xs}, 0) \\
\quad \mathit{value}\ (\mathit{NC}\ (\mathit{xs}, i)) = \mathit{xs}\ !!\ i \\
\quad \mathit{list}\ (\mathit{NC}\ (\mathit{xs}, \_)) = \mathit{xs} \\
\quad \mathit{movL}\ (\mathit{NC}\ (\mathit{xs}, i)) = \mathit{NC}\ (\mathit{xs}, i - 1) \\
\quad \mathit{movR}\ (\mathit{NC}\ (\mathit{xs}, i)) = \mathit{NC}\ (\mathit{xs}, i + 1) \\
\quad \mathit{ovw}\ y\ (\mathit{NC}\ (\mathit{xs}, i)) = \mathit{NC}\ (\mathit{take}\ i\ \mathit{xs} +\!\!+ [y] +\!\!+ \mathit{drop}\ (i + 1)\ \mathit{xs}, i) \\
\quad \mathit{bks}\ (\mathit{NC}\ (\mathit{xs}, i)) = \\
\quad\quad \mathit{NC}\ (\mathit{take}\ (i - 1)\ \mathit{xs} +\!\!+ [\mathit{xs}\ !!\ i] +\!\!+ \mathit{drop}\ (i + 1)\ \mathit{xs}, i - 1) \\
\quad \mathit{ins}\ y\ (\mathit{NC}\ (\mathit{xs}, i)) = \\
\quad\quad \mathit{NC}\ (\mathit{take}\ i\ \mathit{xs} +\!\!+ [y, \mathit{xs}\ !!\ i] +\!\!+ \mathit{drop}\ (i + 1)\ \mathit{xs}, i + 1)
\end{array}
$$

Problem: *value*, *ovw*, *bks*, and *ins* all take time $O(n)$!

Example #1: a functional line editor using list zippers

Better solution: use a zipper!

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{ListZip}\ a = \mathit{LZ}\ ([a], a, [a]) \\
\textbf{instance}\ \mathit{Cursor}\ \mathit{ListZip}\ \textbf{where} \\
\quad \mathit{cursor}\ \mathit{xs} = \mathit{LZ}\ ([\,], \mathit{head}\ \mathit{xs}, \mathit{tail}\ \mathit{xs}) \\
\quad \mathit{value}\ (\mathit{LZ}\ (\mathit{ls}, x, \mathit{rs})) = x \\
\quad \mathit{list}\ (\mathit{LZ}\ (\mathit{ls}, x, \mathit{rs})) = \mathit{reverse}\ \mathit{ls} + [x] + \mathit{rs} \\
\quad \mathit{movL}\ (\mathit{LZ}\ (l : \mathit{ls}, x, \mathit{rs})) = \mathit{LZ}\ (\mathit{ls}, l, x : \mathit{rs}) \\
\quad \mathit{movR}\ (\mathit{LZ}\ (\mathit{ls}, x, r : \mathit{rs})) = \mathit{LZ}\ (x : \mathit{ls}, r, \mathit{rs}) \\
\quad \mathit{ovw}\ y\ (\mathit{LZ}\ (\mathit{ls}, x, \mathit{rs})) = \mathit{LZ}\ (\mathit{ls}, y, \mathit{rs}) \\
\quad \mathit{bks}\ (\mathit{LZ}\ (\_ : \mathit{ls}, x, \mathit{rs})) = \mathit{LZ}\ (\mathit{ls}, x, \mathit{rs}) \\
\quad \mathit{ins}\ y\ (\mathit{LZ}\ (\mathit{ls}, x, \mathit{rs})) = \mathit{LZ}\ (y : \mathit{ls}, x, \mathit{rs})
\end{array}
$$

Now almost all operations are $O(1)$.

Example #1: a functional line editor using list zippers

In this example, a one-hole context is just a pair of lists: the list of values to the left and the list of values to the right.

(But observe that the values to the left are stored in reverse order, i.e., in order of distance from the hole.)

[Let's load the file `ListZipper.hs` to see this in action...]

Example #2: binary tree zippers

Recall the data type of binary trees with labelled leaves:

$$\textbf{data}\ \mathit{Bin}\ a = L\ a \mid B\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a)$$

Now a data type of one-hole contexts for binary trees:

$$\begin{aligned}&\textbf{data}\ \mathit{BinCxt}\ a = \mathit{Hole}\\ &\quad \mid \mathit{B0}\ (\mathit{BinCxt}\ a)\ (\mathit{Bin}\ a)\\ &\quad \mid \mathit{B1}\ (\mathit{Bin}\ a)\ (\mathit{BinCxt}\ a)\end{aligned}$$

Idea: a value of type *BinCxt a* provides a description of what we see on a path from the hole to the root of the tree. (Observe that *BinCxt a* is isomorphic to a list of pairs $[(\mathit{Bool}, \mathit{Bin}\ a)]$.)

Example #2: binary tree zippers

This path description is interpreted by the "plugging" function:

$$
\begin{aligned}
&\mathit{plug} :: \mathit{BinCxt}\ a \to \mathit{Bin}\ a \to \mathit{Bin}\ a \\
&\mathit{plug}\ \mathit{Hole}\ t = t \\
&\mathit{plug}\ (\mathit{B0}\ c\ \mathit{t2})\ t = \mathit{plug}\ c\ (B\ t\ \mathit{t2}) \\
&\mathit{plug}\ (\mathit{B1}\ \mathit{t1}\ c)\ t = \mathit{plug}\ c\ (B\ \mathit{t1}\ t)
\end{aligned}
$$

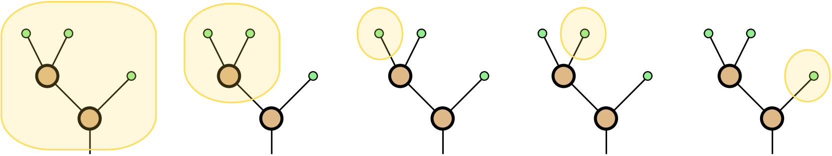
A (binary tree) zipper is a pair of a one-hole context and a tree:

$$\textbf{type}\ \mathit{BinZip}\ a = (\mathit{BinCxt}\ a, \mathit{Bin}\ a)$$

Intuitively, we can think of a zipper (c, t) as defining a (purely functional) pointer to a subtree of $u = \mathit{plug}\ c\ t$.

Example #2: binary tree zippers

For example, the tree $u = B\ (B\ L\ L)\ L$ has five subtrees (c, t):



1. $c = \mathit{Hole}$, $t = B\ (B\ L\ L)\ L$
2. $c = \mathit{B0}\ \mathit{Hole}\ L$, $t = B\ L\ L$
3. $c = \mathit{B0}\ (\mathit{B0}\ \mathit{Hole}\ L)\ L$, $t = L$
4. $c = \mathit{B1}\ L\ (\mathit{B0}\ \mathit{Hole}\ L)$, $t = L$
5. $c = \mathit{B1}\ (B\ L\ L)\ \mathit{Hole}$, $t = L$

Example #2: binary tree zippers

Easy to implement operations for navigating through a tree:[2]

$$
\begin{aligned}
&\mathit{go\_left} :: \mathit{BinZip}\ a \to \mathit{Maybe}\ (\mathit{BinZip}\ a)\\
&\mathit{go\_left}\ (c, B\ t1\ t2) = \mathit{Just}\ (B0\ c\ t2, t1)\\
&\mathit{go\_left}\ (c, L\ \_) \quad\ = \mathit{Nothing}\\
&\mathit{go\_right} :: \mathit{BinZip}\ a \to \mathit{Maybe}\ (\mathit{BinZip}\ a)\\
&\mathit{go\_right}\ (c, B\ t1\ t2) = \mathit{Just}\ (B1\ t1\ c, t2)\\
&\mathit{go\_right}\ (c, L\ \_) = \mathit{Nothing}\\
&\mathit{go\_down} :: \mathit{BinZip}\ a \to \mathit{Maybe}\ (\mathit{BinZip}\ a)\\
&\mathit{go\_down}\ (B0\ c\ t2, t) = \mathit{Just}\ (c, B\ t\ t2)\\
&\mathit{go\_down}\ (B1\ t1\ c, t) = \mathit{Just}\ (c, B\ t1\ t)\\
&\mathit{go\_down}\ (\mathit{Hole}, t) = \mathit{Nothing}
\end{aligned}
$$

[2] These operations can fail if one tries to navigate off the end of the tree. We chose to represent failure explicitly by using *Maybe* for the return type, but an alternative is to raise an exception.

Example #2: binary tree zippers

Similarly easy to implement operations for performing local edits, such as say grafting another tree off to the left or right of the subtree in focus.

$$
\begin{aligned}
&\mathit{graft\_left}, \mathit{graft\_right} :: \mathit{Bin}\ a \rightarrow \mathit{BinZip}\ a \rightarrow \mathit{BinZip}\ a \\
&\mathit{graft\_left}\ g\ (c, t) = (c, B\ g\ t) \\
&\mathit{graft\_right}\ g\ (c, t) = (c, B\ t\ g)
\end{aligned}
$$

[Let's have a look at the solution set for Lab 4 to see how this can be used to solve the optional problem on random binary trees!]

One-hole contexts and derivatives

There is a remarkable link between computing the types of zippers and *differential calculus*, summarized by the slogan that "the derivative of a type is its type of one-hole contexts."[3]

More precisely, the type of one-hole contexts for a parameterized type $T(a)$ may be computed as a partial derivative $\frac{d}{da}T(a)$.

The type of zippers is then obtained by multiplying $a \cdot \frac{d}{da}T(a)$.

[3] Which is also (almost) the title of an unpublished but very influential paper by Conor McBride, `http://strictlypositive.org/diff.pdf`

Types as algebraic expressions

Recall the link we discussed in Lecture 1:

$$
\begin{aligned}
\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b &\iff a + b \\
\textbf{type}\ \mathit{Pair}\ a\ b = (a, b) &\iff a \cdot b \\
\textbf{data}\ \mathit{Void} &\iff 0 \\
\textbf{data}\ () = () &\iff 1 \\
\textbf{data}\ \mathit{Bool} = \mathit{True} \mid \mathit{False} &\iff 2 \\
\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a &\iff 1 + a
\end{aligned}
$$

Many algebraic laws are realized as type isomorphisms (e.g., $a + b = b + a \iff \mathit{Either}\ a\ b \cong \mathit{Either}\ b\ a$).

What about the laws of differentiation?

Some laws of differentiation

$$
\begin{aligned}
\frac{d}{da}(S(a) + T(a)) &= \frac{d}{da}S(a) + \frac{d}{da}T(a) \\
\frac{d}{da}(S(a) \cdot T(a)) &= \frac{d}{da}S(a) \cdot T(a) + S(a) \cdot \frac{d}{da}T(a) \\
\frac{d}{da}(S(T(a))) &= \frac{d}{db}S(b)|_{b=T(a)} \cdot \frac{d}{da}T(a)
\end{aligned}
$$

These all have interpretations as poking holes in data types!

Derivatives of data types: example

Consider the type of pairs of values of the same type:

$$\textbf{type}\ \mathit{Square}\ a = (a, a)$$

A one-hole context for *Square a* is of the form $(-, a)$ or $(a, -)$.

We can represent this with the type

$$\textbf{data}\ \mathit{SquareCxt}\ a = L\ a \mid R\ a$$

which is isomorphic to the product type (Bool, a).

Compare this with $\frac{d}{da} a^2 = 2a$.

Derivatives of data types: example

Consider the type of lists:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

Now expand it algebraically, and differentiate:

$$\begin{aligned}
L(a) &= 1 + a \cdot L(a) \\
&= 1 + a \cdot (1 + a \cdot L(a)) \\
&= 1 + a + a^2 + a^3 + \dots \\
&= (1 - a)^{-1} \\
\frac{d}{da} L(a) &= (1 - a)^{-2} = L(a)^2
\end{aligned}$$

We recover that $\mathit{ListCxt}\ a \cong ([a], [a])$ and $\mathit{ListZip}\ a \cong ([a], a, [a])$!

Derivatives of data types: example

Consider the type of binary trees:

$$\textbf{data}\ Bin\ a = L\ a \mid B\ (Bin\ a)\ (Bin\ a)$$

Now expand it algebraically, and differentiate:

$$\begin{aligned} T(a) &= a + T(a)^2 \\ \frac{d}{da}T(a) &= 1 + 2T(a)\frac{d}{da}T(a) \\ &= \frac{1}{1 - 2T(a)} \end{aligned}$$

We recover that $BinCxt\ a \cong List\ (Bool, Bin\ a)$!

Some references

On zippers and derivatives of data types:

- The Haskell Wikibook on Zippers
- Gérard Huet, "The Zipper", *JFP* 7:5, 1997
- Conor McBride, "The derivative of a regular type is its type of one-hole contexts", unpublished manuscript, 2000

On the closely related theory of "combinatorial species":

- André Joyal, "Une théorie combinatoire des séries formelles", *Advances in Mathematics* 42(1), 1981
- Brent Yorgey, "Species and functors and types, oh my!", in proceedings of *Haskell '10*, 2010 (see also his PhD thesis)