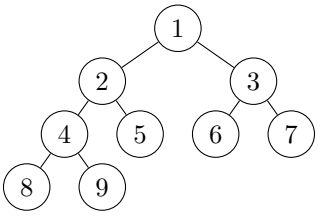
Notes on "The Derivative of a Regular Type is its Type of One-Hole Contexts" by Conor McBride

Felix Filozov

Monday 20th October, 2014

Timestamp:
20141020161700

Suppose we have the following binary tree



and we would like to make *local modifications*, efficiently. Meaning, we're only interested in modifying nodes within a subtree. For this particular example, we're going to focus on modifying (5), we'll call the *target node*.

Using a language like C, which allows mutation, modifying (5) can be done very efficiently. We simply create a pointer to (5), and modify the node through the pointer in $O(1)$.

This seems to be a problem for languages that do not allow mutation, pure languages, or so it seems. Trying to modify (5) the naive way, would mean recursing down to (5), changing it, and then recursing back up. This would have the effect of copying every node on the path from (1) to (5). Here's an example of the naive approach in Haskell:

```haskell
data Tree = Empty | Node Int Tree Tree deriving (Show)

-- Search for 5 in a Tree and replace it with a new value.
naiveModify :: Tree -> Int -> Tree
naiveModify (Node oldVal t1 t2) newVal =
    if oldVal == 5
    then (Node newVal t1 t2)
    else (Node oldVal (naiveModify t1 newVal)
                      (naiveModify t2 newVal))
naiveModify Empty newVal = Empty
```

Once *naiveModify* finds the target node, and replaces the value, it will create new nodes as it recurses back up to the root of the tree.

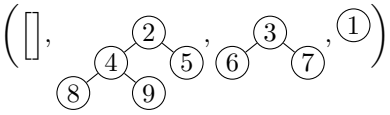
The goal is to come up with a way so that local modifications to the above binary tree can be done efficiently in a pure language.

An elegant solution to this problem is to use a data structure called the *zipper* [1] [3]. Using the zipper, we can traverse the data structure, and make modifications to it. To make local modifications to any "regular" data structure efficiently, we simply have to create a zipper for it. This means other data structures can have their own zippers. For example, lists can also have zippers. However, all zippers work on the same principles. For the purpose of this talk, I will describe how a zipper works for binary trees, so from now on when I mention "zipper", I mean a zipper for binary trees, unless stated otherwise.

Before I tell you how a zipper actually looks like, let me first tell you what operations a zipper supports. A zipper supports the following operations: *left()*, *right()*, *up()*, and *modify(v)*. We could add more, but these are sufficient.

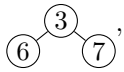
For example, if our location is (2), and we call up(), then the zipper moves to (1). If we call left(), then the zipper moves to (4). If we call right(), then the zipper moves to (5). If we call modify(v), then the value of (2) will be replaced with *v*. It should be clear that using these operations we can walk around the tree, and have the ability to modify any node.

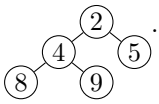
Here's how a zipper looks like:



What we see above is the initial state of the zipper. The first element is the empty list. We'll see later that this list is used by left(), right(), and up(). The fourth element is the *focus node*, that is, the node where we're currently located. The second element is the left branch of the focus node, and the third element is the right branch of the focus node.

At this point you might be thinking that I've sold you snake oil! Is this really what a zipper is? "But it's just the tree broken apart and put into a tuple!", you say. Yes, but the magic actually happens in the operations. So let's look at how they work.

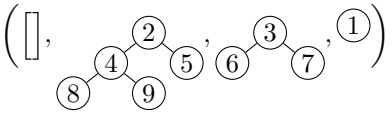
As the zipper moves down the tree using left() and right(), it will be recording all the branches that it did not move to. So if it moves left, it will record the right branch into the previously mentioned list. If it moves right, it will record the left branch. For example, if the starting position is (1), then calling left() would move the zipper to (2), and in addition it would also record the right branch, 

, into the list. Starting from the same position, if we call right(), the zipper would move to (3), and record the left branch 

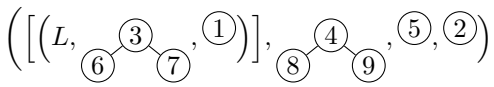
.

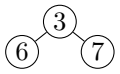
So as the zipper moves down the tree, it doesn't just discard the branches it passes by, instead it saves them. The metaphor for this is to imagine the zipper unzipping the tree. As it moves down, it "unzips" the tree, that is, it splits the tree along a path.

Let's look concretely at how left() works. For convenience, the initial state of the zipper is repeated:



After a single call to left(), the zipper looks like this:

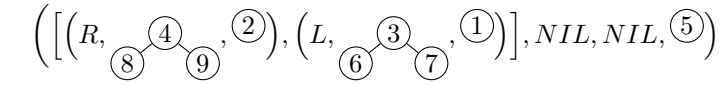


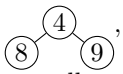
Let's focus on the entry that was added to the list. The L is a marker which means that the zipper moved along the left branch. The second element is the branch the zipper did not move to as it moved left. It did not move to the right branch of ①, that is  , therefore the right branch was saved. The third element is the element from which the zipper moved away, which was ①.

Every time the zipper moves left or right, this list will be augmented with additional entries.

Also, notice that the rest of the tuple also changed. The focus node has become ②, and the left and right branch of the focus node has been recorded in the tuple as well.

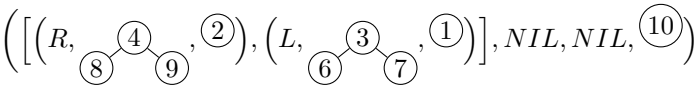
If we then call right(), the zipper changes to:



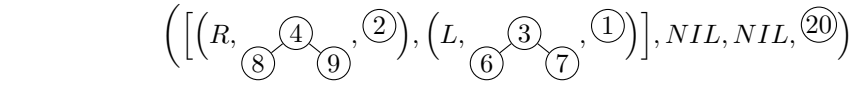
A new entry was added to the list. The R is a marker to indicate that the zipper moved right. Since the zipper moved right, the left branch of ②, that is  , was recorded. Since it moved away from ②, it recorded that node as well.

The rest of the tuple also changed. The new focus node is ⑤, and since it has no children, the zipper recorded NIL.

We've reached the node we wanted to modify. Let's modify this node by calling modify(10). The zipper looks like this:



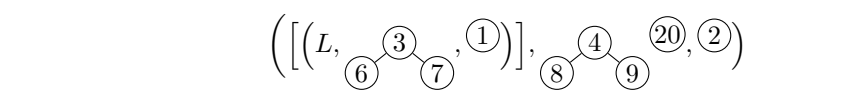
Let's modify it again by calling modify(20). The zipper looks like this:

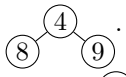
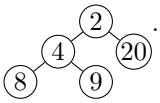


We can continue modifying this particular node, and this modification will be done in $O(1)$, because we're only changing one entry in the tuple, the focus node. We're no longer paying the heavy price by having to copy all the nodes along the path to our target node. Hurray! We've achieved our goal.

After we've done all the necessary modifications to ⑤ (it became ⑳), suppose we decide to go back up to the root of the tree. We simply call up() twice. Because the zipper saved all the branches it did not move to, as it moved to the target node, the zipper will be able to reconstruct the tree as it moves up.

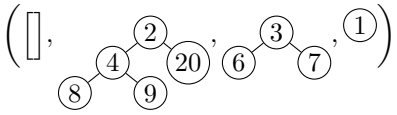
If we call up(), the zipper looks like this:

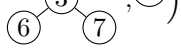


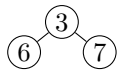
The focus node was ⑳, and by calling up() the zipper moved to the parent node ②. By moving to ②, the zipper reconstructed the entire subtree rooted at ②. It was able to do that by using the head entry in the list, . The entry tells the zipper that the parent node of ⑳ is ②. The left branch of ② is  . The R marker indicates that the zipper moved to the right from ② to get to ⑳, which means ⑳ is the right branch of ②. Putting all this together the zipper reconstructed the subtree rooted at ② as:  .

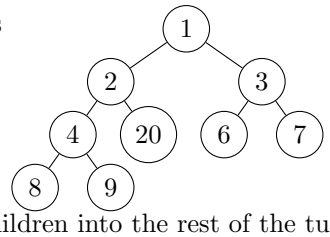
The zipper recorded ② as the new focus node and its children into the rest of the tuple.

Let's call up() again. This time the result is:



The zipper moved to the parent node ① of the focus node ②. By doing so, it reconstructed the entire subtree rooted at ①, which happens to be the tree itself. Once again, it used the head entry in the list,  .

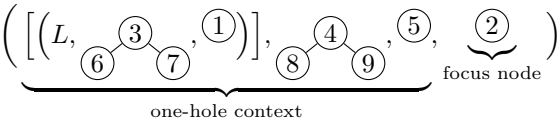
The marker L indicates that the zipper moved left to get to ②. The entry also tells us the zipper moved away from ①. This means the subtree rooted at ② is the left branch of ①. In addition, the entry also has the right branch of ①, that is 

. Putting all this together, the reconstructed subtree rooted at ① is 

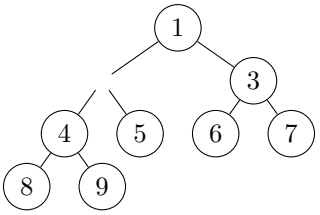
The zipper then put the new focus node and its children into the rest of the tuple.

Going up the tree is called zipping up the tree. It's a metaphor. The list of entries in the zipper contains separated branches. As the zipper moves up the tree, it puts those branches back together until we get the tree back.

Let's introduce some terminology.



Looking at a zipper above, we've already given the fourth element in the tuple a special name, the focus node. Everything else in the tuple is called the *one-hole context*. The one-hole context represents everything around the focus node. That is, it's the tree itself without the focus node, the hole. Here's a way of showing the one-hole context:



This is how the zipper is able to reconstruct the tree as it moves up. It uses the one-hole context, and the focus node to rebuild the tree.

Since this paper is about types, let's write down the types of the data structures we've seen so far:

$$BTree = 1 + Int * BTree * BTree$$
$$BTreeZipper = [(2 * BTree * Int)] * BTree * BTree * Int$$

Conor's paper [2] tells us that the type of $BTree$ and the type of the one-hole context are related by the rules of differentiation. In particular, we can derive

the type of the one-hole context, $[(2 * BTree * BTree)] * BTree * BTree$, from the type of $BTree$. This means that we can derive the type of the zipper of $BTree$ as well. In fact, we can derive the type of the zipper of any "regular" type.

I should mention that he represents recursive types differently than what I've shown. He uses the μ operator instead. He would write the type of the tree like this:

$$BTree = \mu y.(1 + Int * y * y)$$

From the little I know about recursive type theory, if we treat μ like a lambda abstraction and repeatedly replace y with $\mu y.(1 + Int * y * y)$, then we'll get an infinite tree type. For example, if we replace once, we get:

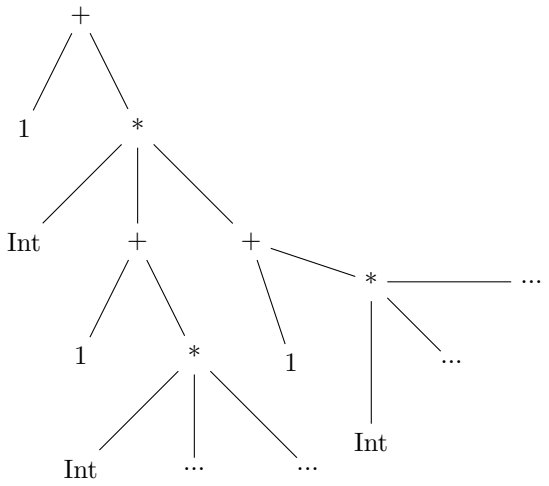
$$BTree = (1 + Int * \mu y.(1 + Int * y * y) * \mu y.(1 + Int * y * y))$$

If we continue replacing, we'll get:

$$BTree = (1 + Int * (1 + Int * (1 + Int * ... * ...) * (1 + Int * ... * ...)) * (1 + Int * (1 + Int * ... * ...) * (1 + Int * ... * ...)))$$

where "..." are further expansions.

Another way of representing the infinite tree type:



This infinite tree type is equivalent to the fixed-point of $\mu y.(1 + Int * y * y)$. That's as far as I know about recursive types, if you know more, please find me and tell me more about them. From this point on, I'm simply going to use it as notation.

Conor defines the differentiation operator as follows:

$$
\begin{aligned}
\partial_x &\mapsto 1 \\
\partial_x(y \backslash x) &\mapsto 0 \\
\partial_x 0 &\mapsto 0 \\
\partial_x(S + T) &\mapsto \partial_x S + \partial_x T \\
\partial_x 1 &\mapsto 0 \\
\partial_x(S \times T) &\mapsto \partial_x S \times T + S \times \partial_x T \\
\partial_x(\mu y.F) &\mapsto \mu z.\underline{\partial_x F|y = \mu y.F}_z + \underline{\partial_y F|y = \mu y.F}_z \times z \\
\partial_x(F|y = S) &\mapsto \partial_x F|y = S + \partial_y F|y = S \times \partial_x S \\
\partial_x \underline{T}_x &\mapsto 0 \\
\partial_x \underline{T}_{y \backslash x} &\mapsto \underline{\partial_x T}_y
\end{aligned}
$$

We can use our intuition to make sense of some of these. For example, if our data structure is $S + T$, then the one-hole context of such a data structure will clearly be the one-hole context of S, or T. It wouldn't make sense for the one-hole context of the data structure to just be the one-hole context of S, because what if we have an instance of type T?

With regards to the product rule, if we put a hole in S, then we'll have to record T. If we put a hole in T, then we'll have to record S. This is similar to what we did with our tree. The zipper moved left, that is it put a hole in the left branch, therefore it saved the right one.

Let's apply the rules to $BTree$:

$$
\begin{aligned}
\partial_{Int}(BTree) &= \partial_{Int}\mu y.(1 + Int * y * y) \\
&= \mu z.\partial_{Int}(1 + Int * y * y)|y = BTree + \partial_y(1 + Int * y * y)|y = BTree \times z \\
&= \mu z.(\partial_{Int}(1) + \partial_{Int}(Int * y * y))|y = BTree + (\partial_y(1) + \partial_y(Int * y * y))|y = BTree \times z \\
&= \mu z.\partial_{Int}(Int * y * y)|y = BTree + \partial_y(Int * y * y)|y = BTree \times z \\
&= \mu z.(y * y)|y = BTree + (\partial_y(Int) \times y * y + \partial_y(y * y) * Int)|y = BTree \times z \\
&= \mu z.(BTree * BTree) + (\partial_y(y * y) * Int)|y = BTree \times z \\
&= \mu z.(BTree * BTree) + ((\partial_y(y) * y + y * \partial_y(y)) * Int)|y = BTree \times z \\
&= \mu z.(BTree * BTree) + (y * Int + y * Int)|y = BTree \times z \\
&= \mu z.(BTree * BTree) + (2 * y * Int)|y = BTree \times z \\
&= \mu z.(BTree * BTree) + (2 * BTree * Int) \times z \\
&= [(2 * BTree * Int)] * (BTree * BTree) \\
&= [(2 * BTree * Int)] * BTree * BTree
\end{aligned}
$$

Observe that $\partial_{Int}(BTree)$ equals to the type of the one-hole context. From here, deriving the type of the zipper is simple.

In the paper we can also find the derivation of the one-hole context for the list data structure.

References

[1] Huet Gérard, *The Zipper*, J. Funct. Program., September 1997, Vol. 7 Num. 5, 1997

[2] McBride Conor, *The Derivative of a Regular Type is its Type of One-Hole Contexts (Extended Abstract)*, 2001.

[3] Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, `http://learnyouahaskell.com/zippers`, 20011.