Zippers and Data Type Derivatives

Simon Roßkopf
Department of Informatics
TUM
rosskops@in.tum.de

June 2015

Abstract

In this paper, an efficient and convenient way of traversing and modifying a data structure, called the Zipper, is demonstrated. This is done by denoting a position in the structure by a context surrounding it and a hole which data can be plugged into. Furthermore, it is demonstrated that zippers can be automatically generated for a subset of the algebraic data types by an operation analogous to the derivative known from Calculus.

1 Introduction

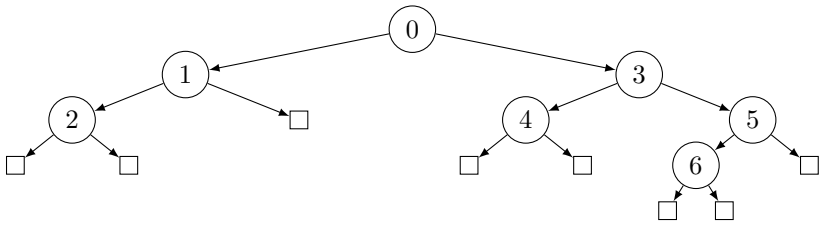
In everyday programming, the need of editing an already existing data structure may arise. This is no problem in classic imperative languages, as long as one knows a way of navigating to the position in the data structure that will change. Then, one can just update the corresponding place in memory. As functional languages do not allow destructive updates to already existing structures, the naive way of performing an update is to create a entirely new structure with just the one position changed. If we recreate the entire structure, this operation is obviously linear in the size of the structure. Luckily, many modern functional programming languages allow the sharing of equal parts. Is it possible to use this to perform better?

Consider for example an algebraic data type, the binary tree with values saved in the nodes. All code examples I will give in this paper will be written in the purely functional language Haskell.

```
data BinaryTree a =
          Leaf
        | Node a (BinaryTree a) (BinaryTree a)
```

Using this definition, one can create a small example tree, containing for example Integers, displayed in figure 1:

Figure 1: A small example tree




```
exampleTree :: BinaryTree Integer
exampleTree = Node 0
        (Node 1
                Leaf
                (Node 2 Leaf Leaf))
        (Node 3
                (Node 4 Leaf Leaf)
                (Node 5 Leaf Leaf))
```

If one wants change a value in this tree, one has to select a position in the tree to change. One could specify this position by giving a list of directions to take from the root node. In the case of the binary tree, there are clearly only two choices at every step, going into the left or right subtree. This allows to denote a position in a binary tree as a pair of the tree and a list of directions to the location.

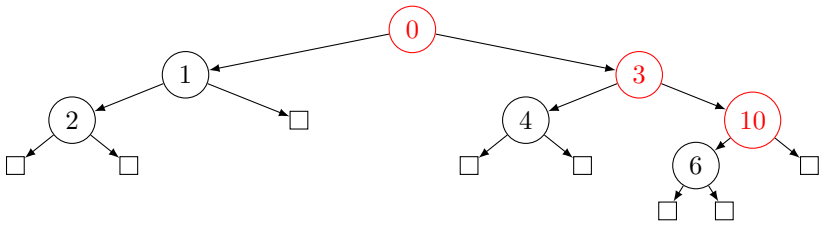
```
data Direction = L | R
type Position a = (BinaryTree a, [Direction])
```

One can use this to write a small update function for binary trees. Note that the update function can fail if one enters an incorrect list of directions. For the sake of keeping the algorithms short and understandable, no proper error handling is implemented here. As soon as the function reaches a leaf, it is not possible to reach a value to update any more (as going up is not possible), so an error has occurred. If the list of directions is empty, one has arrived and the new value is inserted. If there are still elements left in the list, remove the head and use it to decide whether to descend into the left or the right subtree and continue there recursively.

```
update :: a -> Position a -> BinaryTree a
update _ (Leaf, _) = error "Cannot update a leaf"
update new (Node _ left right, []) =
        Node new left right
update new (Node v left right, (L:ds)) =
        Node v (update new (left, ds)) right
update new (Node v left right, (R:ds)) =
        Node v left (update new (right, ds))
```

In Figure 2 you can see the tree after the update operation has been performed to change the value 5 to 6. It is interesting to note that only the nodes along the path one travelled down (marked red) to reach the '5' have changed.

Figure 2: The example tree after changing a value. Nodes that cannot be shared marked red



The rest of the nodes are untouched and could be shared and only the path to the changed location has to be recreated. This means the update operation can run in $\mathcal{O}(depth(tree))$ time.

Furthermore, as updates often occur in the neighbourhood of a certain location, it would be nice to have functions to quickly move around the current position. One could define functions to shift the current position around. As at the moment, the position in a tree is marked by a list of directions, movement is done by manipulating this list. These functions work by appending a direction if one wants to go down a subtree, or remove the last direction to move up a level.

```
up :: Position a -> Position a
up (t, []) = (t, [])
up (t, ds) = (t, init ds)

left :: Position a -> Position a
left (t, ds) = (t, ds ++ [L])

right :: Position a -> Position a
right (t, ds) = (t, ds ++ [R])
```

However, all of those functions must walk the entire direction list each time they are called. This means all of them are in $\mathcal{O}(depth(tree))$ time as well. It would be nice to have a purely functional way to update and move in the tree in constant time.

2 Huet's Zipper

The following solution to this problem was first published by Huet [1]. He however remarked, that he was not the first to discover this technique and considered it some kind of obscure programming "folklore" [1].

2.1 The Data Structure

The idea is that instead of storing the tree and a path to the location one wants to change, one can keep the value currently subject to change in focus and see the rest of the tree as some kind of context around this value.

```
type Zipper a = (a, Context a)
```

The context describes the structure of the tree around the currently focused value. Naturally, it has to contain both children of the focussed node. Furthermore, as we are not walking down from the root any more, it has to store all the rest of the tree 'above', too.

```
data Context a = Context
        (BinaryTree a) --Left child
        (BinaryTree a) --Right child
        (AboveContext a) --Rest of the tree
```

The parent of node holding the currently focussed value of course needs to be in the above context. Depending on whether the current location is in the left or the right subtree of the parent, one has to store the other one. Furthermore, the parent node may be in a context itself, which means the above context type must contain itself recursively. At the root, there is no more context above to be stored.

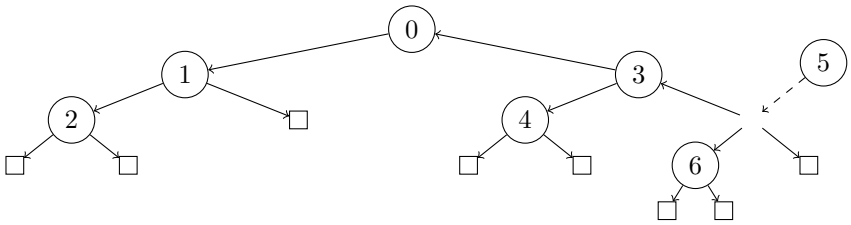
```
data AboveContext a = Top
        | LSubTree  a (BinaryTree a) (AboveContext a)
        | RSubTree a (BinaryTree a) (AboveContext a)
```

Overall the context type describes a tree (albeit in a slightly complicated notation) with one value missing. This is why one calls it a 'One-Hole-Context'. However, the direction of some edges is inverted. Instead of pointing downwards from the root to the hole, the context above is structured upwards from the hole to the root. A decomposition of the example tree from section 1 into a focussed value (the 5) and context can be seen in figure 3. In Haskell, the Zipper focusing on the value five would look like this:

```
exampleFocusOn5 = (
    5, --Focused value
    Context
        (Node 6 Leaf Leaf) --Left subtree
        Leaf --Right subtree
        (RSubTree --Focused value is in right subtree
            3 --Value of parent node
            (Node 4 Leaf Leaf) --Other child of parent
            (RSubTree --Parent also is a right subtree
                0 --Value of parent's parent
                (Node --Other child of parent's parent
                    1
                    (Node 2 Leaf Leaf)
                    Leaf
                )
                Top --No more context above the root
            )
        )
)
```

One can also arrive at Huet's idea by thinking why the problems with the first proposed solution from section 1 arise. The problem comes from always

Figure 3: The example tree decomposed into a value and the context



having to walk the full list of directions, whether it is in order to append or remove the last element or to navigate from the root to a location that needs updating.

Huet proposes to reverse this. Instead of walking from the root to the location, one should consider starting at a position and storing the way back to the root.

Then the context is very similar to the direction list shown before, but reversed. The context 'above' can be attached to the direction list. If one is taking a left turn, save the right tree and vice versa. If one has reached the top, there is no more context, so the list is empty. Additionally, one also has to save the two child trees of the current location again. So, only the context above changes:

```
type AboveContext a = [(Direction, a, BinaryTree a)]
```

One can transform the first version into the second. The first step is condensing the two recursive constructors into one, by adding another field specifying the which subtree one descended into.

```
data AboveContext a = Top
        | Up a Direction (BinaryTree a) (AboveContext a)
```

Now `AboveContex a` is a linked list of 3 tuples of `Direction`, value of type `a` and `BinaryTree a`. `Top` is used in place of the empty list constructor and `Up` is used for pre pending a value. Extracting this list from the data type and writing it explicitly leads to the second version. I will be using this second version for the rest of this paper.

2.2 Operations on the Zipper

With this new data structure, it is now time to revisit the function up, left, right and update from Section 1. Using the Zipper, those can be rewritten to operate in a faster way.

Direct access to the value as the first component of the Zipper tuple saves walking down from the root each time. Additionally, as the context is never touched, one can reuse all of the rest tree. This means the update function now runs in constant time.

```
update :: a -> Zipper a -> Zipper a
update new (_, c) = (new, c)
```

Figure 4: Updating the node with value 5 to 10

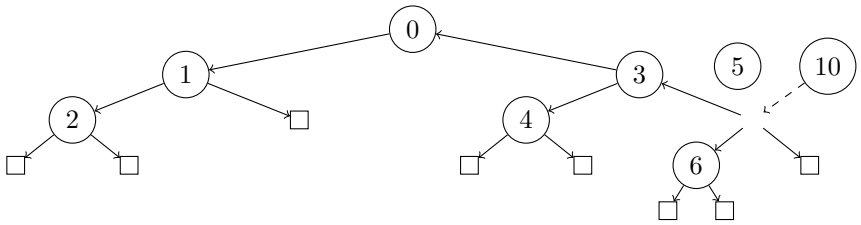
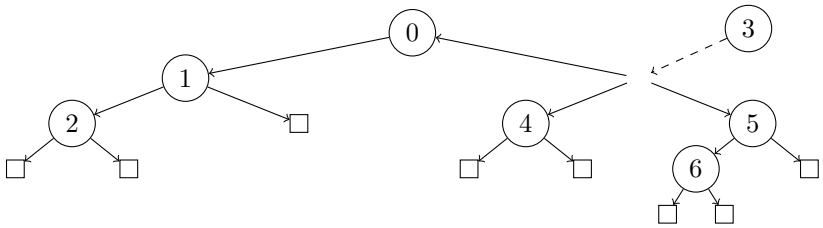


Figure 5: Moving the focus from the value 5 to its parent's value 3

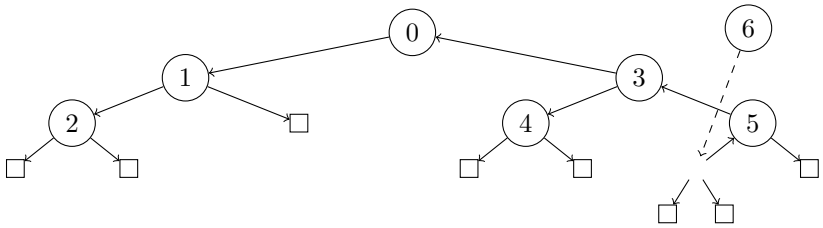


Moving upwards in the tree is relatively straightforward. It is impossible when already being at the top. If there is still context left, one can shift the focus to the parent's value. One of the children in the new context can be constructed from the old focused values and its children. The other one was saved with the parents value in the context list. Which of those is the left/right one is decided by the direction value in the context list. The context list towards the top just loses its head as one is moving a step towards the root. During all those operations, only the first element of the context list is used. Furthermore, no intrusion into the child trees happens; only two of them are glued together by a new shared parent node. This means all of them can be reused, which makes the up operation run in constant time.

```
up :: Zipper a -> Zipper a
up (t, Context _ _ []) = error "already at root"
up (t, Context l r ((L, p, s):cs)) =
        (p, Context (Node t l r) s cs)
up (t, Context l r ((R, p, s):cs)) =
        (p, Context s (Node t l r) cs)
```

The `left` and `right` functions put the value of the left/right subtree as the new focus. As one has moved down one level, the context needs to grow by a new element representing the node previously focused. It can be constructed from the previously focused value, the subtree one did not descend into and the direction one has taken. As this element is just pre-pended to the context list and no intrusion into the subtrees occurs, the operation works in $\mathcal{O}(1)$ time.

Figure 6: Moving the focus from the value 5 to its left child 6




```
left :: Zipper a -> Zipper a
left (old, Context (Node new l r) o cs) =
        (new, Context l r ((L, old, o):cs))
left _ = error "hit a leaf"

right :: Zipper a -> Zipper a
right (old, Context o (Node new l r) cs) =
        (new, Context l r ((R, old, o):cs))
right _ = error "hit a leaf"
```

Lastly, we need functions to convert between the Zipper and ordinary trees. The `toZipper` function just puts the focus on the root node (no context above). If the focus is on the root, the `fromZipper` function can reconstruct the tree from the root value and the two children of the root. Otherwise, one keeps to shift the focus up, till the root is reached.

```
fromZipper :: Zipper a -> BinaryTree a
fromZipper (t, Context l r []) = Node t l r
fromZipper c = fromZipper (up c)
```

3 Other Zippers

Of course there are more data structures than just binary trees which could benefit from efficient navigation and local updates. If one considers, for example, ternary trees, one can define a similar Zipper:

```
data TTree a = Leaf | Node a (TTree a) (TTree a) (TTree a)

data Direction = Left | Down | Right
data Context a = [(Direction, Tree a, Tree a)]
type Zipper a = (TTree a, Context a)
```

This structure can be manipulated with functions similar to those presented in section 2, one just has to account for three subtrees and direction values instead of two. This means a function `down` has to exist next to `left` and `right`. Furthermore, the `up` function now needs to distinguish between three different cases, depending which of the three subtrees one has chosen.

Another very important data structure is the linked list, for which it is also possible to create a Zipper. If one makes a hole at a certain location in a list, the context, of course, contains the following elements. The above context is

simpler than in trees, as there are no different direction to take. This means it consists only of the values before, stored in reverse order. So a list Zipper consists of the currently focused value, the following elements and the preceding elements in reverse order.

```
data List a = Nil | Cons a (List a)
type Zipper a = (List a, a, List a)
```

For navigation, only two functions are needed. One to move forward (corresponding to `left` and `right`) and one to move backwards (corresponding to `up`) in the list. Note that contrasting the tree example, the lack of different directions and choices, leads to them just moving values around. The update, toZipper and fromZipper functions are working analogously to the tree example.

```
forward :: Zipper a -> Zipper a
forward (_, _, Nil) = error "reached the end"
forward (ls, f, Cons r rs) = (Cons f ls, r, rs)

backward :: Zipper a -> Zipper a
backward (_, _, Nil) = error "reached the beginning"
backward (Cons l ls, f, rs) = (ls, l, Cons r rs)
```

If one looks at Haskell's abstract data types, one can see that all of them are either finite(there are no recursive type declarations involved) or have a 'tree-like' structure. In this, the constructors containing no recursive type declarations act as leaves whereas constructors which (indirectly) reference their own types count as nodes. This prompts the question whether it is possible to find a uniform way of deriving Zippers for abstract data types.

4 Deriving a Zipper

I will now show an automatic way of deriving a Zipper for a subset of the algebraic data types similar to McBride [3]. Please note that the following aims to provide an intuition rather than a rigorous formal construction of the method. Furthermore, all the following can easily break when allowing undefined values ($\bot, undefined$). So I will assume that bottom values do not occur.

4.1 Regular Tree Types

I will only show a way of deriving the Zipper for a subset of the algebraic data types, which I will call, following [3], the regular tree types. Those are all the types which can be constructed combining only the following.

The basic blocks are the empty type `Void`, which does not have any constructors and the `Unit` type with exactly one constructor but no arguments. In Haskell the `Unit` type and its only value are normally written as `()`.

```
data Void
data Unit = Unit
```

The following two types combine those basic blocks to create new types. The `Either a b` type offers choice between two types. It is similar to a tagged

union in other programming languages. The `Pair a b` type builds new types by combining two existing types. One could also use a 2-tuple or a struct with two members.

```
data Either a b = Left a | Right b
data Pair a b = Pair a b
```

A simple example is defining a type similar to Haskell's `Bool`. A boolean can have two values, false and true. Both carry no extra information, so they can be represented using a `Unit` value, with the `Either` type providing choice. This means `Left Unit` corresponds to `False` and `Right Unit` to `True`.

```
type Bool' a = Either Unit Unit
```

Another example is the following regular tree type, isomorphic to `Maybe a`. Here one is using the `Either` type to provide the choice between the `Nothing` value (`Left Unit`) and the `Just` values (`Right a`).

```
type Maybe' a = Either Unit a
```

So far, the shown regular tree type only had finitely many possible values. In order to change this, one also allows recursive data type definitions, which means a definition in which the name of the defined type also appears in its definition.

With this, for example a regular tree type isomorphic to `[a]` can be constructed. Breaking down the example, the `Either` allows to choose between an empty list (a `Unit` value) or a pair of a value and another list. The `newtype` used in this example is just here to satisfy Haskell's type checker.

```
newtype List' a = List' (Either Unit (Pair a (List' a)))
```

However, it is necessary to note, that one cannot create all the types with these regular tree types. For example it is not possible to use functions types. Furthermore types that need to fulfil certain invariants in addition to their structure like sets can not be encoded.

4.2 A mathematical notation for regular tree types

One can relate those regular tree types to numbers, by *counting* the number of different values that can be created for each type [4]. This is interesting, as it allows a new view on data types, which will allow to see relationships and laws more easily. I will use the notation $|a| = n$ to denote that a type `a` contains exactly n values.

As `Void` has no constructors there, can be no values of type `Void`, $|\texttt{Void}| = 0$. The `Unit` type only has a single constructor without arguments, which can create exactly the one value `Unit`. So $|\texttt{Unit}| = 1$.

How many Values can be constructed for the type `Either a b`? The type offers a choice to use values of type `a` or type `b`. So `Either a b` contains all values of `a` tagged with `Left` and all values of `b` tagged with `Right`. This means $|\texttt{Either a b}| = |\texttt{a}| + |\texttt{b}|$ and justifies relating Either to the operator '+'.

One can argue similarly to count the values of the type `Pair a b`, which contains all pairs of values from a and b. From combinatorics one knows that it is possible to create $|\texttt{a}| \cdot |\texttt{b}|$ values.

Because of this, one can adopt the notation for numbers also for the types. To go back to my earlier examples, one can write $Bool' = 1+1$, $Mabye'(a) = 1+a$ and $List'(a) = 1 + a \cdot List'(a)$.

It is interesting to note, that a lot of the rules for numbers carry over to the regular tree types. However, one must note that the given equalities between types do not mean that Haskell would consider the types equal but the types are isomorphic. For example, 0 is also the additive identity($a + 0 = 0 + a = a$). Because there are no values for `Void` , one can only create values of `a`(with the extra `Left` or `Right` tag). $0 \cdot a = a \cdot 0 = 0$ because if one cannot create a value for one of the components of the tuple one also cannot create the whole tuple. $1 \cdot a = a \cdot 1 = a$ because forming a tuple with a single Unit value is just adding another tag to the value.

The classic associative, commutative and distributive laws hold (up to isomorphism) for the regular tree types. One can prove this, for example, by providing functions to convert between the two types. I will exemplary provide such functions for the commutative and associative law of addition $a+b = b+a$ and $a + (b + c) = (a + b) + c$, however all the other laws follow similarly [4].

```
sumCom :: Either a b -> Either b a
sumCom (Left a)  = Right a
sumCom (Right b) = Left b

sumAssL :: Either a (Either b c) -> Either (Either a b) c
sumAssL (Left x) = Left (Left x)
sumAssL (Right (Left x)) = Left (Left x)
sumAssL (Right (Right x)) = Right x

sumAssR :: Either (Either a b) c -> Either a (Either b c)
sumAssR (Right x) = Right (Right x)
sumAssR Left (Right x) = Right (Left x)
sumAssR Left (Left x) = Left x
```

These laws help to understand that one did not really lose anything by only allowing to choose between two types or pair two objects. Choices between or values of multiple types can be simulated by chaining the pairwise versions together.

In addition to those simple rules, there are a lot more laws that can be proven to also hold for types. As this can be very tedious to do by hand, one can use that the regular tree types form a semiring. This is interesting, as it allows us to transfer other laws from the complex numbers to the types, if certain conditions are met [2]. For the rest of this paper I assume that the laws I am using are transferred and valid.

4.3 Deriving Types

If one wants to create a Zipper for a data structure, one has to divide the data structure into the focused value and a context representing the rest of

the structure around this value. Another way to say this, the context is the structure with a hole in the position to which the focussed value belongs. In this section I will explore how to punch such a hole in a structure. I will use the notation $\partial_a b$ to denote a context around a hole of type a in a structure of type b.

The basic types `Void` and `Unit` do not have ways of storing an `a`. So the type of the contexts around an `a` those two types create must not contain any values and is therefore `Void`. This means that $\partial_a 0 = \partial_a 1 = 0$.

More generally, if a type contains no field of type a (it is constant in a), there is no place where one can make a hole of type a. This means that for all types not containing `a`, the context they provide for an `a` is `Void`.

If one tries to make a hole of type a in an a, there is only one possibility. So $\partial_a a = 1$.

A more interesting case is the sum of the type $u + v$. Remember that this type gives the choice of either using a value from u or v. So if one wants to make a hole of type a in $u + v$, this hole can either be in a value of u or of v. So the type of $\partial_a(u + v) = \partial_a u + \partial_a v$.

If one considers a pair of two data structures (u, v) making a hole is a bit more complicated. One can make the hole either in the first or the second component. If one makes the hole in the first component, the context contains the second component unchanged. If one makes a hole in the second component, one keeps the first component. So $\partial_a(u \cdot v) = (\partial_a u) \cdot v + u \cdot (\partial_a v)$.

Looking at this, one can realize, that this is exactly the Leibniz rule for differentiating a product. Furthermore, all the previous rules for constants, values and addition also seem consistent with differentiation. This leads to the conjecture that making a hole in a data type corresponds to differentiation. One can now check whether other differentiation rules also hold for regular tree types.

One important rule is the power rule $x^n = n \cdot x^{(n-1)}$. In type language, this would give a way of making a hole in an n-tuple of as. One can start with a one tuple. A one tuple is just a single a, which generates a one-hole-context of 1. Using the power rule one gets $\partial_a a^1 = 1 \cdot a^0 = 1$ which is consistent. If one looks at a 2-tuple, one can use the Leibniz rule. $\partial_a(a \cdot a) = 1 \cdot a + a \cdot 1 = 2 \cdot a$, which one can also generate with the power rule $\partial_a a^2 = 2 \cdot a$. One can say, that the one-hole-context is a two-valued field, specifying where in the tuple the hole is and the other field of the tuple. Similarly, for a 3-tuple the power rule gives $\partial_a a^3 = 3 \cdot a^2$. This makes sense, the one-hole-context consists of a three valued field specifying where the hole is and the other two components of the tuple.

Another interesting rule for differentiating is the chain-rule. It specifies that $\partial_a f(g(a)) = (\partial_g f)(g(a)) * (\partial_a g(a))$. This also makes sense for types. Here one looks to make a hole of type a in a structure f containing other structures of type g containing as. To do this, one must change the value in the inner structure, so one needs a context of this changed structure. However, one also needs to know where to put this changed structure in the outer structure. So one makes a hole for a g in the outer structure.

It is possible to extend this classic chain rule, when realizing that even in a nested data structure, values of type a can occur not only in the inner structures, but also on the outer level. All the possibilities for holes created by the chain rule are still valid. In addition, it is also possible to just create a hole of type a in the outer structure.

In order to generate one-hole-contexts by differentiation, one is still missing the ability to differentiate recursive definitions, occuring for example in the definition of the List type. On way is to treat the definition as an equation on can first solve for the type, then differentiate this non recursive version. For example the List type $List(a) = 1 + a \cdot List(a)$ can be manipulated to first read $List(a) - a \cdot List(a) = 1$ and then $List(a) = 1/(1-a)$. This can then be differentiated to $\partial_a List(a) = 1/(1-a)^2 = (1/(1-a))^2$, which gives us the correct one-hole-context for a list, a prefix list andsummands, and taking their sum. In more explicit language a suffix list to the hole.

However, here one is relying on that all the manipulations one is doing can be safely transferred from the complexed numbers. This way also introduces a lot of new concepts and questions. For example what do subtraction and division mean on types?

Another way we can take is to 'unroll' our recursive definition by substituting repeatedly. Then one gets a representation as some kind of 'power series', which states that a list of as is either empty or has one element or two elements or three elements and so on.

$$
\begin{aligned}
List(a) &= 1 + a \cdot List(a) \\
List(a) &= 1 + a \cdot (1 + a \cdot List(a)) = 1 + a + a^2 \cdot List(a) \\
List(a) &= 1 + a + a^2 \cdot (1 + a \cdot List(a)) = 1 + a + a^2 + a^3 \cdot List(a) \\
List(a) &= 1 + a + a^2 + a^3 + \dots
\end{aligned}
$$

One can then find the closed form of this series with the geometric series formula to be $List(a) = 1/(1-a)$. This could again be differentiated. However the same problems as with the last attempt remain.

Both ways lead to $\partial_a List(a) = (1/(1-a))^2 = List(a)^2$. This means the One-Hole-Context of a list is a pair of lists. As a Zipper is a context paired with a value, the type of a List Zipper is $a \cdot List(a)^2 = List(a) \cdot a \cdot List(a)$, which is consistent to the earlier defined List Zipper.

Similarly other recursive types can also be derived by finding a closed form of their 'equation'. Then they can be differentiated to yield the type of their One-Hole-Context.

A Zipper can be created by pairing such a One-Hole-Context with a value to be placed in the hole.

5 Conclusion

In this paper a data structure supporting fast local update and movement called the Zipper was shown. It works by decomposing a data structure into a focussed value that can currently be updated and the rest of the structure forming a context around this value. Furthermore, an intuition for automatically generating Zipper structures was given. This was done by exploring the relation between types and natural numbers and relating the decomposition of a structure to the differentiation.

References

[1] G. Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[2] T. Leinster M. Fiore. Objects of categories as complex numbers. *Adv. Math*, pages 264–277.

[3] C. McBride. The derivative of a regular type is its type of one-hole contexts.

[4] A. Brent Yorgey. Species and functors and types, oh my! In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 147–158, New York, NY, USA, 2010. ACM.