TODO: *What's the title?*

João Paulo Fernandes[1], Pedro Martins[2], Alberto Pardo[3], João Saraiva[4], Marcos Viera[3], and Tom Westerhout[5]

[1] LISP/Release – Universidade da Beira Interior, Portugal jpf@di.ubi.pt
[2] University of California, Irvine, USA pribeiro@uci.edu
[3] Universidad de la República, Uruguay {pardo,mviera}@fing.edu.uy
[4] Universidade do Minho, Portugal saraiva@di.uminho.pt
[5] Radboud University, The Netherlands twesterhout@student.ru.nl

Abstract. **TODO:** *What's the abstract?*

Keywords: Embedded Domain Specific Languages · Zipper data structure · Memoization · Attribute Grammars · Higher-Order Attribute Grammars · Functional Programming

1 Introduction

2 Functional Zippers

Zipper is a data structure commonly used in functional programming for traversal with fast local updates. The zipper data structure was originally conceived by Huet[?] in the context of trees. We will, however, first consider a simpler problem: a bidirectional list traversal.

Suppose that we would like to update a list at a specific position:

```
modify :: (a → [a]) → Int → [a] → [a]
modify f i xs = helper [] xs 0
  where helper before (x : after) !j
          | j ≡ i     = before ++ f x ++ after
          | otherwise = helper (before ++ [x]) after (j + 1)
        helper _ [] _ = error "Index out of bounds."
```

Here `modify` takes an update action `f`[6], an index `i`, and a list `xs` and returns a new list with the `i`'th element replaced with the result of `f`.This function "unpacks" a list, modifies one element, and "packs" the result into a list. If we do a lot of updates, we end up unpacking and packing the list over and over again – very time-consuming for long lists. Explicitly working with the unpacked representation is bug-prone. A list zipper simplifies this.

A zipper consists of a focus (alternatively called a hole) and surrounding context:

[6] `f` returns a list rather than a single element to prevent curious readers from suggesting to use a boxed array instead of a list.

```
data Zipper a = Zipper { _hole :: a, _cxt :: !(Context a) }
  deriving (Show, Eq)
data Context a = Context [a] [a]
  deriving (Show, Eq)
```

where the `ListContext` keeps track of elements to the left and to the right of the focus. We can now define movements:

```
left :: Zipper a → Maybe (Zipper a)
left (Zipper _    (Context [] _))        = Nothing
left (Zipper hole (Context (l : ls) rs)) = Just $ Zipper l $ Context ls (hole : rs)

right :: Zipper a → Maybe (Zipper a)
right (Zipper _    (Context _ []))        = Nothing
right (Zipper hole (Context ls (r : rs))) = Just $ Zipper r $ Context (hole : ls) rs
```

and functions for entering and leaving the zipper:

```
lzEnter :: [a] → Maybe (Zipper a)
lzEnter []       = Nothing
lzEnter (x : xs) = Just $ Zipper x (Context [] xs)

lzLeave :: Zipper a → [a]
lzLeave (Zipper hole (Context ls rs)) = reverse ls ++ hole : rs
```

Finally, we define a local version of our `modify` function (**TODO:***Boy, is this function ugly...*)

```
lzModify :: (a → [a]) → Zipper a → Maybe (Zipper a)
lzModify f (Zipper hole (Context ls rs)) = case f hole of
  (x : xs) → Just $ Zipper x (Context ls (xs ++ rs))
  []       → case rs of
    (r : rs') → Just $ Zipper r (Context ls rs')
    []        → case ls of
      (l : ls') → Just $ Zipper l (Context ls' rs)
      []        → Nothing
```

using which we can perform multiple updates efficiently and with minimal code bloat[7]:

```
modifyExample :: IO ()
modifyExample = print $
  lzEnter @Int >=> right
               >=> right
               >=> lzModify (const [])
               >=> lzModify (return ∘ (+1))
               >=> left
               >=> lzModify (return ∘ negate)
               >=> return ∘ lzLeave $
    [1, 2, 3, 4, 5]
```

Consider now a binary tree data structure:

```
data Tree a
  = Fork (Tree a) (Tree a)
  | Leaf !a
  deriving (Show, Eq)
```

A binary tree zipper is slightly more insteresting than the list zipper, because we can move up and down the tree as well. The zipper again consists of a hole (a subtree we are focused on) and its surrounding context (a path from the hole to the root of the tree):

```
data Zipper a = Zipper { _hole :: (Tree a), _cxt :: !(Context a) }
  deriving (Show, Eq)

data Context a
  = Top
  | Left !(Context a) (Tree a)
  | TreeRight (Tree a) !(Context a)
  deriving (Show, Eq)
```

To move the zipper down, we "unpack" the current hole:

```
down :: Zipper a → Maybe (Zipper a)
down (Zipper (Leaf _) _) = Nothing
down (Zipper (Fork l r) cxt) = Just $ Zipper r (TreeRight l cxt)
```

`TreeContext` stores everything we need to reconstruct the hole, and `tzUp` does exactly that:

[7] Operator `>=>` comes from `Control.Monad` module in `base` and has the following signature:

```
(>=>) :: Monad m => (a → m b) → (b → m c) → a → m c
```

```
up :: Zipper a → Maybe (Zipper a)
up (Zipper _ Top) = Nothing
up (Zipper l (Left cxt r)) = Just $ Zipper (Fork l r) cxt
up (Zipper r (TreeRight l cxt)) = Just $ Zipper (Fork l r) cxt
```

Implementation of `tzLeft`, `tzRight`, and `tzEnter` is very similar to the List zipper case and is left as an exercise for the reader. `tzLeave` differs slightly in that we now move all the way up rather than left:

```
leave :: Zipper a → Tree a
leave z = case up z of
  Just z' → leave z'
  Nothing → _hole z
```

The list and binary tree zipper we have considered here are homogeneous zippers: the type of focus does not change upon zipper movement. Such zippers can be a very useful abstraction. For example, a well-known window manager XMonad[**?**] uses a rose tree zipper to track the window under focus. For other tasks, however, one might need to traverse heterogeneous structures. A zipper that can accomodate such needs is usually called a generic zipper as it relies only on the generic structure of Algebraic Data Types (ADTs). One can view an ADT as an Abstract Syntax Tree (AST) where each node is a Haskell constructor rather than a syntax construct.

The generic zipper we will use is very similar to the one presented in[**?**]. The most common technique in Haskell for supporting heterogeneous types is Existential Quantification. However, not every type can act as a hole. To support moving down the tree, we need the hole to be *dissectible*, i.e. we would like to be able to dissect the value into the constructor and its arguments. Even though `Data.Data.gfold` allows us to achieve this, we define out own typeclass which additionally allows us to propagate down arbitrary constraints:

```
class Dissectible (c :: Type → Constraint) (a :: Type) where
  dissect :: a → Left c a

data Left c expects where
  LOne  :: b → Left c b
  LCons :: (c b, Dissectible c b) => Left c (b → expects) → b → Left c expects
```

For example, here is how we can make `Tree` an instance of `Disssectible`

```
instance c (Tree a) => Dissectible c (Tree a) where
  dissect (Fork l r) = LOne Fork `LCons` l `LCons` r
  dissect x = LOne x
```

We can unpack a `Fork` and the zipper will thus be able to go down. `Leaf`s, however, are left untouched and trying to go down from a `Leaf` will return `Nothing`.

To allow the zipper to move left and right, we need a means to encode arguments to the right of the hole. Following Adams et al, we define a GADT representing constructor arguments to the right of the hole:

```
data Right c provides r where
  RNil  :: Right c r r
  RCons :: (c b, Dissectible c b) => b → Right c provides r → Right c (b → provides) r
```

For example, for a tuple `(Int, Int, Int, Int, Int, Int)`, we can have

```
lefts = LOne (,,,,,) `LCons` 1 `LCons` 2 `LCons` 3
hole = 4
rights = 5 `RCons` 6 `RCons` RNil
```

generalising this a little, we arrive at:

```
data LocalContext c hole rights parent =
  LocalContext !(Left c (hole → rights)) !(Right c rights parent)

data Context :: (Type → Constraint) → Type → Type → Type where
  RootContext :: ∀ c root. Context c root root
  (:>) :: ∀ c parent root hole rights. (c parent, Dissectible c parent)
       => !(Context c parent root)
       → {-# UNPACK #-} !(LocalContext c hole rights parent)
       → Context c hole root
```

And just like before the `Zipper` is a product of the hole and context:

```
data Zipper (c :: Type → Constraint) (root :: Type) =
  ∀ hole. (c hole, Dissectible c hole) =>
    Zipper { _zHole :: !hole
           , _zCxt  :: !(Context c hole root)
           }
```

Implementation of movements is quite straightforward and is left out. Please, refer to (**TODO:** *github repo*) for complete code.

We now consider a rather interesting application of generic zipper: embedding of attribute grammars.

3 Attribute Grammars

Attribute grammars (AGs) are an extension of context-free grammars that allow to specify context-sensitive syntax as well as the semantics. AGs achieve it by associating a set of attributes with each grammar symbol. These attributes are defined using evaluation rules assiciated with production rules of the context-free grammar.

Attributes are then usually divided into two disjoint sets: synthesized attributes and the inherited attributes. Such distinction is required for the construction of a dependency graph. It is then used for specification of the evaluation order and detection of circularity. In the zipper-based embedding of attribute grammars we make no use of a dependency graph and thus do not divide attributes into classes.

Let us consider the repmin[8] problem as an example of a problem that requires multiple traversals. The classical solution is the following circular program:

```
repmin :: Tree Int → Tree Int
repmin t = t'
  where (t', m') = go t m'
        go (Leaf x    ) m = (Leaf m, x)
        go (Fork xs ys) m = (Fork xs' ys', min mx my)
          where (xs', mx) = go xs m
                (ys', my) = go ys m
```

Although quite elegant, the code lacks modularity and is very difficult to reason about. Attribute Grammars provide a more modular approach. Viera et al[**?**] identified three steps for solving repmin: computing the minimal value, passing it down from the root to the leaves, and constructing the resulting tree. We can associate each step with an attribute[**?**]:

- A synthesized attribute `localMin :: Int` represents the minimum value of a subtree. Computing the minimal value thus corresponds to evaluation of the `localMin` attribute for the root tree.
- An inherited attribute `globalMin :: Int` is used to pass down the minimal value.
- Finally, a synthesized attribute `updated :: Tree Int` is the subtree with leaf values replaced by values of their `globalMin` attributes. The solution is thus the value of `updated` attribute for the root tree.

The obtained AG is presented in figure **??**. **TODO:***Do we actually need to explain the algorithms here? It seems a little bit childish to explain how to compute the minimum of a binary tree... If we absolutely have to explain stuff, maybe just put it in the caption.*

We now move on to embed this attribute grammar into Haskell. Semantic rules simply become functions, which, given a zipper, return values of the attributes. For example,

```
localMin :: Zipper (WhereAmI Position) (Tree Int) → Int
localMin z@(Zipper hole _) = case whereami hole of
  C_Leaf  → let Leaf x = hole in x
  C_Fork  → let Just l = child 0 z; Just r = child 1 z
              in min (localMin l) (localMin r)
```

Apart from the type signature, the code is pretty straightforward and closely mirrors the AG we defined earlier. `whereami` function allows us to "look around"

[8] The repmin problem:

> Given a tree of integers, replace every integer with the minimum integer in the tree, in one pass.

```
SYN Tree Int [localMin : Int]
SEM Tree Int | Leaf lhs.localMin = @value
             | Fork lhs.localMin = min @left.localMin
                                       @right.localMin

SYN Tree Int [updated : Tree Int]
SEM Tree Int | Leaf lhs.updated = Leaf @lhs.globalMin
             | Fork lhs.updated = Fork @left.updated
                                       @right.updated


INH Tree Int [ globalMin : Int ]
SEM Tree Int | Fork left.globalMin  = @lhs.globalMin
                    right.globalMin = @lhs.globalMin

DATA Root | Root tree : Tree Int
SEM  Root | Root tree.globalMin = @tree.localMin
```

Fig. 1: Attribute grammar for repmin. The syntax is closely mirrors the one used in[?]. `SYN` and `INH` introduce synthesized and inherited attributes respectively. `SEM` is used for defining semantic rules. A new data type *Root* is introduced as it is common in the AG setting to "connect" `localMin` with `globalMin`.

and returns the position of the zipper. Thus the **case** corresponds to the pattern matches on the left of the vertical bars on figure ??.

Position of the zipper is encoded using the following GADT which can be generated automatically using Template Haskell:

```
data Position :: Type → Type where
  C_Leaf  :: Position (Tree Int)
  C_Fork  :: Position (Tree Int)
```

Parametrization on the type of the hole allows the code like **let** *Leaf* `x = hole` **in** `x` to typecheck, even though the generic zipper itself knows close to nothing about the type of the hole. It might seem trivial at first, because the binary tree zipper is in fact homogeneous. The "position trick" however extends also to heterogeneous zippers which we will encounter in more advanced examples.

```
child :: Int → Zipper cxt root → Maybe (Zipper cxt root)
```

`child n` moves the zipper to the `n`'th child, if there is one.

4 Related Work

5 Conclusion

Acknowledgements