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[1] 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.

[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.

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

```
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 update efficiently and with minimal code bloat:

```
modifyExample :: IO ()
modifyExample = print $
  lzEnter@Int >=> right
```

```
    >=> right
    >=> lzModify (const [])
    >=> lzModify (return ∘ (+1))
    >=> left
    >=> lzModify (return ∘ negate)
    >=> return ∘ lzLeave $
    [1, 2, 3, 4, 5]
```

Application to binary trees...

```
data Tree a
  = Fork (Tree a) (Tree a)
  | Leaf !a
data Path a
  = Top
  | Left !(Path a) (Tree a)
  | TreeRight (Tree a) !(Path a)
data Zipper a = Zipper !(Path a) (Tree a)
```

Application to lists...
Generic zipper...
An application of generic zipper that we will consider is embedding of attribute grammars.

3 Attribute Grammars

What attribute grammars are...
Repmin as two traversals...
Repmin as a circular program...
Repmin as an AG...

4 Related Work

5 Conclusion

Acknowledgements

References

References

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