Tabulation with Zippers

Marcos Viera[1][0000−0003−2291−6151], Alberto Pardo[1][0000−0003−3336−9755], and João Saraiva[2][0000−0002−5686−7151]

[1] Instituto de Computación, Universidad de la República, Montevideo, Uruguay
{mviera,pardo}@fing.edu.uy
[2] Dep. of Informatics & HASLab/INESC TEC, University of Minho, Braga, Portugal
saraiva@di.uminho.pt

Abstract. Tabulation is a well-known technique for improving the efficiency of recursive functions with redundant function calls. A key point in the application of this technique is to identify a suitable representation for the table. In this paper, we propose the use of zippers as tables in the tabulation process. Our approach relies on a generic function `zipWithZipper`, that makes strong use of lazy evaluation to traverse two zippers in a circular manner. The technique turns out to be particularly efficient when the arguments to recursive calls are closely situated within the function domain. For example, in the case of natural numbers this means function calls on fairly contiguous values. Likewise, when dealing with tree structures, it means functions calls on immediate sub-trees and parent nodes. This results in a concise and efficient zipper-based embedding of attribute grammars.

Keywords: Zipper · Tabulation · Generics · Attribute Grammars

1 Introduction

The evaluation of recursive functions may require multiple computations of identical function calls. A classical example of this behaviour is the recursive definition of the Fibonacci function as show in Figure 1a. Although this definition is simple and easy to understand, it is very inefficient due to the repeated computation of identical recursive calls; a call to `fib n` requires two evaluations of `fib(n-2)`, three evaluations of `fib(n-3)`, and so on.

One possible solution to improve this inefficiency is the use of *tabulation* [2], a well-known technique that uses a bottom-up scheme in which function calls are computed once and stored in a *table* for future reuse. Function calls in the program code are then changed by lookups in the table. In Figure 1b we present a tabulation-based version of `fib`, where we rely on lists to model the table.

In this case the table `fibTab` is given by a list of suspensions, each one corresponding to the computation of a Fibonacci number. Recursive calls in the original function definition are replaced by requests to other positions in the table by using list indexing `(!!)`. The necessary sharing of the table in order to avoid multiple recomputations of its entries is achieved thanks to lazy evaluation and

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

(a) Recursive

```
tab_fib n = fibTab !! n
  where fibTab = map (fibm fibTab) [0..]
        fibm _ 0 = 0
        fibm _ 1 = 1
        fibm t n = t !! (n−1) + t !! (n−2)
```

(b) Tabulation-based

Fig. 1: Fibonacci

a circular definition of the table. In fact, the suspended computations stored in the table entries form a sort of dependency graph in such a way that the request of certain position of the table triggers the evaluation of the other entries (computations) it depends on. Lazy evaluation ensures that, once an entry is evaluated, its value is available without the need for recomputation.

Despite the use of tabulation, a drawback of `tab_fib` is the use of a list for implementing the table, as list indexing requires to traverse the list from the beginning each time an element is required. This implies that each call to `tab_fib` causes multiple traversals through the list to pick up the values that are required for the computation of the different positions. Using an array instead of a list improves efficiency by allowing direct access to any position.

Fibonacci turns out to be a good motivating example to show the power of tabulation, but on the other hand it relies on a simple, linear data structure to model the table. There are algorithms, however, that require more elaborated tables to be used with tabulation. Indeed, several algorithms may use an underlying inductive structure, for example a tree, to store values on the nodes so that they can be reused in future computations.

In this paper we focus on the design of the table. Our aim is to provide a generic and efficient solution for the table design and manipulation that can be uniformly used in many applications of the tabulation technique. Concretely, the generic solution we propose is to represent the table as a *zipper* [5], a data structure that provides efficient navigation through tree structures, allowing to move *left*, *right*, *up*, or *down* within a tree.

Once we have the table represented as a zipper, our technique provides the ways to move along the table in order to build the suspensions that are stored in the table and reflect the outcomes of the original function. The assembly of the table is carried out by a function, called `zipWithZipper`, which performs a sort of `zipWith` between the table (the zipper) and the (virtual) structure of function arguments that is traversed by the original function to compute each particular result (e.g. the list of naturals that Fibonacci traverses when called on a certain argument `n`). Again, a sophisticated use of lazy evaluation and circularity proves to be an essential aspect to achieve an effective and elegant solution.

In the case of Fibonacci, the solution based on a zipper looks as follow:

```
tab_fib_Z n = focusL (fibTab !> n)
  where fibTab = zipWithZipperL fibm fibTab [0..]
        fibm _ 0 = 0
        fibm _ 1 = 1
        fibm z n = focusL (z <! 1) +  focusL  (z <! 2)
```

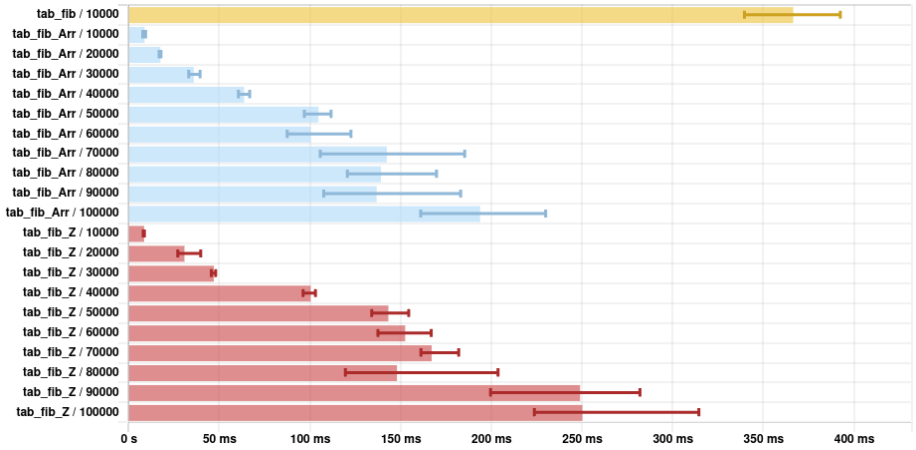


Fig. 2: Performance of the Zipper-based Fibonacci

where the operators `(<!)` and `(!>)` make it possible to move left and right (resp.) on the zipper a given number of times. Function `focusL` returns the zipper focus, while `zipWithZipperL` (introduced in Section 3) denotes the particular case of `zipWithZipper` where the construction of the table is performed by zipping a zipper that navigates on a list and a list of indexes.

It is interesting to mention that, as shown in Figure 2, the zipper-based version of Fibonacci is competitive even with the array-based implementation.

The rest of the paper is structured as follows. In Section 2 we show how this technique can be applied to optimize functions that navigate through tree-shaped terms, delving into a concise and efficient embedding of *Attribute Grammars*. In Section 3, we introduce the function `zipWithZipper`, which forms the backbone of the technique. We also provide some generic instances of the function, that we use in Section 4 to implement a complete example of Attribute Grammar. In Section 5 we analyze the performance of the programs obtained using this technique, identifying some cases when it is worth using it and others where it is not. Finally, we discuss related work and conclude.

2 Zipper Tabulation of Functions on Trees

Zippers are mostly used to navigate tree-shaped terms. This leads us to the question whether we can take advantage of the tabulation technique using zippers in the context of functions on trees.

As an example, let us consider a solution to the classical *repmin* problem, where a zipper is used to navigate through the tree. The goal of *repmin* is to transform a binary leaf tree containing integers into a structurally identical tree, where all leaves are replaced by the original's tree minimum value.

Given a datatype representing binary leaf trees:

```
data Tree = Leaf Int | Fork Tree Tree
```

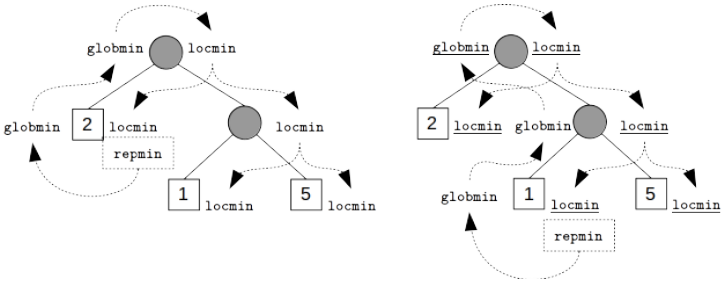


Fig. 3: Recomputations of the Zipper-based repmin

we can define a zipper to navigate through trees of this type. The navigation functions `downT`, `upT`, `leftT` and `rightT` move the focus in the given direction, such that `downT` moves to the left-most child, `upT` moves to the parent, and `leftT` and `rightT` move to the left and right sibling (resp.) if they exist. With the function `focusT` we get the subtree currently being visited.

The following code implements a zipper-based solution to *repmin*:

```
repmin z = case focusT z of
  (Leaf _)   → Leaf (globmin z)
  (Fork _ _) → Fork (repmin (downT z)) (repmin ((rightT . downT) z))

globmin z = if isTopT z then locmin z else globmin (upT z)

locmin z = case focusT z of
  (Leaf x)   → x
  (Fork _ _) → min (locmin (downT z)) (locmin ((rightT . downT) z))
```

Function `repmin` computes a new tree with the same shape as the original one, but replacing the value of each leaf by the global minimum. The (global) minimum of the original tree is computed bottom-up by `locmin`, and distributed top-down by `globmin`. Notice the use of the zipper functions to navigate to the children in `repmin` and `locmin`, and to the parent in the case of `globmin`.

This implementation implies plenty of recomputations of the functions `locmin` and `globmin`. In Figure 3 we show the function calls to compute `repmin` to the tree `Fork (Leaf 2) (Fork (Leaf 1) (Leaf 5))` at the leaf with value 2 (left) and at the leaf with value 5 (right). We underlined the function calls that are recomputed in the latter, which are a major source of inefficiency. The idea is then to apply a technique like the one applied to Fibonacci in order to avoid those re-computations.

In Fibonacci the table was indexed by a sequence of integers and therefore could be implemented with a list. In contrast, in this case we want to store computations on each node of a tree. To do so let us define a type `TTree` to represent the table, which is isomorphic to the original tree but with values attached at every node.

```
data TTree t = TLeaf t | TFork (TTree t) (TTree t) t
```

Now, using a function `zipWithZipperT` (introduced in Section 3), which zips a zipper that navigates on trees of type `TTree` with a tree of type `Tree`, we can implement a tabulation-based version of *repmin* (following the same receipt

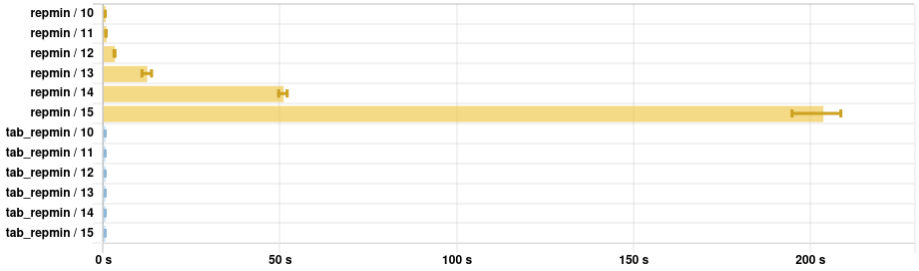


Fig. 4: Performance of the tabulated repmin

applied to Fibonacci), which generates a table with the suspended computations and uses the zipper to navigate through it:

```
repmin t = focusRmin rmTab
  where rmTab = zipWithZipperT (app (gmin,lmin,rmin)) rmTab t

        rmin z (Leaf _)   = Leaf (focusGmin  z)
        rmin z (Fork _ _) = Fork ((focusRmin . downT)  z)
                                 ((focusRmin . rightT . downT) z)

        gmin z _          = if isTopT z then focusLmin  z
                                        else (focusGmin . upT) z

        lmin _ (Leaf x)   = x
        lmin z (Fork _ _) = min ((focusLmin . downT)  z)
                                ((focusLmin . rightT . downT) z)
```

In this case we have three mutually recursive functions instead of just one. Consequently, each node of the table stores a triple of computations (one corresponding to each function). Functions `focusRmin`, `focusGmin` and `focusLmin`, get the focus and project the desired component of the triple. With `app` we define a function that takes a triple of functions `(gmin,lmin,rmin)` and applies each one to a zipper and a tree.

In this new version of *repmin*, we avoid all duplicated computations, resulting in a significant improvement in efficiency. Figure 4 presents the execution times of both versions of *repmin* for complete trees with depths ranging from 10 to 15. While the former grows exponentially, the latter presents times that are relatively negligible.

2.1 Efficient Zipper-based Attribute Grammars

Some readers may have already noticed that we have the ingredients to define an embedding of *Attribute Grammars* [6, 7]. Indeed, the zipper-based version coincides with the zipper-based AG embedding proposed in [11]. Attribute Grammars are a formalism used to define and compute attributes of nodes in a tree-like structure. Attributes can be categorized as *inherited* and *synthesized*, where the former propagate values downwards the tree, and can be seen as arguments of the (evaluator) functions, while the latter compute values upwards the tree, and correspond to results of the (evaluator) function. In the case of *repmin*, `repmin` and `locmin` are synthesized attributes, while `globmin` is an inherited attribute.

However, in order to have a proper AG embedding we must not only provide a zipper that walks through a tree-shaped table, but also a zipper that (in parallel) walks through the original tree, since some computations may depend on their context. For instance, to compute an inherited attribute we might need to access the parent and siblings of a given node. To motivate this kind of data dependency let us define the following minimal expression language with let bindings:

```
data Expr = Lit Int | Var String | Add Expr Expr | Let String Expr Expr
```

An expression in this language can be either an integer literal, a variable, the sum of two expressions or a let expression. We define the evaluation semantics in terms of two attributes: a synthesized attribute, *eval*, that contains the value obtained by evaluating the expression and an inherited attribute, *env*, that distributes the variable environment, adding the bindings defined on let expressions.

```
eval zt ze = case focusE ze of
    (Lit x)     → x
    (Var v)     → (slookup v . focusEnv) zt
    (Add _ _)   → focusEval (zt .$ 1) + focusEval (zt .$ 2)
    (Let _ _ _) → focusEval (zt .$ 3)

env  zt ze
  | isTopE ze = []
  | otherwise = case ((focusE . parentE) ze, parent zt) of
    (Let x _ _,zp) → if ze .? 3
                     then (x,focusEval (zp .$ 2)) : focusEnv zp
                     else focusEnv zp
    (_         ,zp) → focusEnv zp
```

Function `eval` implements the attribute of the same name. The arguments `zt` and `ze` are both zippers, the former for the table and the latter for the expression. By getting the focus of `ze` we obtain the node and thus determine the kind of expression we are evaluating. For literals, the result is simply their value. In case we use the value of an attribute, we get it from `zt`. The operator `(.$)` navigates to the given child, and `(.?)` returns true if the given child exists. For addition, the result is the sum of the *eval* attributes of the subexpressions, whereas for `let` the result is the *eval* attribute of the third subexpresion. In case of variables, we have to search the value of the variable in the environment *env*.

For the *env* attribute we have to analyze the zipper of the expression. If we are at the top, then we return the empty environment. Otherwise, the environment to return depends on the parent. If the parent is a let expression, and we are computing the environment of the third subexpression (`ze .? 3`), then the resulting environment is the inherited *env* of the parent extended with the new binding, which associates the variable `x` with the value of *eval* for the second subexpression. In any other case the environment is just inherited from the parent.

Finally we "tie the knot", as we did in the previous examples. In this case we use a version of `zipWithZipper` that navigates through two zippers in parallel.

```
semExpr :: Expr → Int
semExpr e = focusEval tab
  where tab = zipWithZipper (app (eval, env)) tab (toZipper e)
```

Notice that this version of `zipWithZipper` is more general than the previous ones, in the sense that all the previous examples could have been written following this same approach, by just applying the function to get the focus in the cases we do not need the zipper. For example, `tab_fib_Z` (shown in Section 1) can be rewritten as follows (assume `fm` is some function):

```
tab_fib_Z n = focus ( fibTab !> n )
  where fibTab = zipWithZipper fibm' fibTab [0 ..]
        fibm' z zn = fm z (focusL zn)
        ...
```

3 Zip With Zipper

In this section, we develop the `zipWithZipper` function to traverse two zippers in parallel, potentially circularly. We demonstrate specific implementations to identify common patterns before generalizing it.

In Section 1 we used `zipWithZipperL` to zip a zipper with a list, while constructing the resulting zipper. This function can be implemented as follows:

```
zipWithZipperL :: (ZipperL a → b → c) → ZipperL a → [b] → ZipperL c
zipWithZipperL f _ []     = emptyL
zipWithZipperL f z (x:xs) = consL (f z x) $
                            zipWithZipperL f ((fromJust . rightL) z) xs
```

where `emptyL` represents the zipper corresponding to the empty list, whereas `consL` returns a new zipper which has `(f z x)` as focus.

Notice that, unlike in previous sections, the navigation function `rightL` now returns the zipper as part of a `Maybe` type. In previous examples, we did not want to introduce the noise of using error handling, but from now on, we will use safe versions. The use of `fromJust` makes the function partial, as it is only defined in cases where the length of the path through the list zipper (`ZipperL`) is greater than or equal to the length of the list `(x:xs)`. That is not a problem in the context in which we use the function, as the zipper is constructed in a circular manner from the list. Therefore, we say that the list traversal *guides* the zipper traversal. It would indeed be a problem if we attempt to inspect the zipper before building it. For instance, by defining the second equation in the following way we would have made the function total:

```
zipWithZipperL f z (x:xs) = consL (f z x) zs
  where zs = maybe [] (λz' → zipWithZipperL f z' xs) (rightL z)
```

but would not allow circularity, since we have to inspect the zipper we are constructing in order to determine if the `rightL` movement returns `Just` or `Nothing`, generating thus a loop. However, if we want to be total anyway, allowing the use of `zipWithZipperL` for contexts other than our own, a possible solution is to change `(fromJust . rightL)` by `(maybe z id . rightL)`, using for the last calls the same zipper `z`. Notice that in doing so we are not inspecting the zipper while traversing, we are just generating suspended computations that will provide the function `f` access to a specific place in the zipper when necessary.

Now, let us analyze `zipWithZipperT`, which was the function we used in Section 2 to implement *repmin*.

```
zipWithZipperT :: (ZipperTTree a → Tree → b) → ZipperTTree a → Tree
                  → ZipperTTree b
zipWithZipperT f z t@(Leaf _)   = toZipperT $ TLeaf (f z t)
zipWithZipperT f z t@(Fork l r) = toZipperT $ TFork (fromZipperT zl')
                                                    (fromZipperT zr')
                                                    (f z t)
    where zl = (fromJust . downT)  z
          zr = (fromJust . rightT) zl
          zl' = zipWithZipperT f zl l
          zr' = zipWithZipperT f zr r
```

Like in the case of lists, here it is the traversal on the tree that guides the recursion. For the resulting zipper we are just generating computations that will navigate through it once needed by the function `f`. In fact, both `zipWithZipperL` and `zipWithZipperT` have the same structure: they recursively produce new zippers for the children of the third argument and combine them in a new zipper with the result of applying `f` to the actual zipper and value.

As we explained at the end of Section 2, a more general version of this last function can be defined by using a zipper (`ZipperTree`) instead of a tree (`Tree`).

```
zipWithZipperTZ :: (ZipperTTree a → ZipperTree → b) → ZipperTTree a
                   → ZipperTree → ZipperTTree b
zipWithZipperTZ f z zt = case children zt of
   []    → toZipperT $ TLeaf (f z zt)
   [l,r] → toZipperT $ TFork (fromZipperT zl') (fromZipperT zr')
                             (f z zt)
    where zl = (fromJust . downT)  z
          zr = (fromJust . rightT) zl
          zl' = zipWithZipperTZ f zl l
          zr' = zipWithZipperTZ f zr r

children z  = maybe [] g (downT z)
   where g z' = [z', (fromJust . rightT) z']
```

With function `children` we obtain either the list of zippers pointing to the (two) children or, in the case of a leaf, the empty list. Thus, we use the navigation functions (`downT` and `rightT`) of the second zipper to guide the recursion.

3.1 `zipWithZipper`

We have already observed that the various versions of `zipWithZipper` essentially perform the same operation: recursion over the children of the second structure, providing a way to navigate the corresponding children in the first structure, and combining the results with the result of applying function `f` to both structures. This leads us to a unified implementation, where the variations that may occur depending on the zipper used will be implemented as class instances.

The class `Zipper` contains the minimal interface we require of a zipper: a way to construct a zipper from a structure, and four navigation functions.

```
class Zipper z where
  type Root z
  toZipper :: Root z → z
  up, down, left, right :: z → Maybe z

class Zipper z ⇒ CombineZipper z where
  type Info z
  combine :: Info z → [z] → z
```

The associated type `Root z` is the type of the structure we want to navigate with the zipper `z`. Note that, despite being present in all zippers, the class does not include functions for retrieving or modifying the focus. This is due to two reasons: firstly, we do not need them for implementing `zipWithZipper`, and secondly, their types can vary among different zippers. For example, the type of the focus in a generic zipper does not have to be the same as the type of the root.

The class `CombineZipper` represents a zipper we can construct by the combination of some information (`Info`) with a list of zippers representing the substructures (i.e. the children).

Based on these two classes and the pattern we identified in the previous subsection, we can define a generic version of `zipWithZipper`:

```
zipWithZipper :: (CombineZipper z, Zipper z')
              ⇒ (z → z' → Info z) → z → z' → z
zipWithZipper f z zt = combine (f z zt) (zipWith ($) fzs zs)
 where fzs = mapZipper (flip (zipWithZipper f)) zt
       zs  = (fromJust . down) z : map (fromJust . right) zs

mapZipper :: Zipper z ⇒ (z → b) → z → [b]
mapZipper f z  = maybe [] g (down z)
 where g z' = f z' : maybe [] g (right z')
```

As mentioned before, we combine the result of applying `f` to the actual zippers with the recursive calls to the children. With `mapZipper` we apply the recursive call to all the immediate children of the zipper `zt`, obtaining a list of functions `fzs`. The functions in `fzs` will then be applied to the corresponding elements of the list `zs`, which is the (potentially infinite) list of zippers that are obtained by descending one level in zipper z. This list contains the zipper with the focus on the first child of z's focus, the focus on the second child, and so on. Sharing makes that navigation to each element of the list starts from the previous one and not each time from the root.

3.2 Tying the Knot

Finally, we can also abstract the tabulation process.

```
tabulate :: (CombineZipper z, Zipper z')
         ⇒ (z → a) → (z → z' → Info z) → Root z' → a
tabulate get f x = get tab
   where tab = zipWithZipper f tab (toZipper x)
```

3.3 A Generic Zipper Instance of `zipWithZipper`

Now that we have a general interface for zippers and a corresponding implementation of `zipWithZipper` based on it, we can define some useful instances. The Generic Zipper [1] is a zipper implementation based on the *Scrap your Boilerplate*[8–10] framework which operates over any[3] possibly heterogeneous tree-shaped type. Thus, if we provide an instance of our `Zipper` class for the Generic Zipper, we can then use our approach for any structure. The instance is immediate:

[3] Provided the type is an instance of `Data`.

```
instance Data t ⇒ Zipper (GZ.Zipper t)  where
  type Root (GZ.Zipper t) = t

  toZipper = GZ.toZipper
  up       = GZ.up
  down     = GZ.down'
  left     = GZ.left
  right    = GZ.right
```

3.4 A Generic Table

In the examples of Section 2 we had to define new datatypes, isomorphic to the original trees, in order to represent the tables. This can be done in a more generic way, by representing the tables with a Rose Tree.

```
data Rose t = Node t [Rose t]
```

This is somehow related to *type erasure* [9], where the generic function `gmapQ` is used to render terms as trees, preserving their shapes. In our case the work is done by the `zipWithZipper` function.

The instance of `Zipper` for `Rose` presents no surprises. The context stores the left and right siblings of the focus and the context of the parent.

```
data ZipperRose a = ZR (Rose a) (CtxR a)
data CtxR a       = TopR | CtxR [Rose a] a (CtxR a) [Rose a]

instance Zipper (ZipperRose t) where
 type Root (ZipperRose t) = Rose t
 ...
```

Then, for instance, to go left we obtain the focus from the list of left siblings and store the former focus in the list of right siblings. To go down involves moving the focus to the first element of the list of children of the node, and store the rest as right siblings on the context. To go up we restore the `Node` based on the information of the context.

The `CombineZipper` instance is also straightforward.

```
instance CombineZipper (ZipperRose m)  where
  type Info (ZipperRose m) = m

  combine m zs = toZipper (Node m (map fromZipperR zs))

fromZipperR :: ZipperRose a → Rose a
fromZipperR (ZR t TopR) = t
fromZipperR z           = (fromZipperR . fromJust . up) z
```

4 The Expression Language Revisited

Now that we have all the ingredients defined, we use them in a more complex example that involves heterogeneous trees. We extend the expression language of Section 2 now allowing let expressions to include a list of bindings. Additionally, we divided the `Expr` type in four different types.

```
data Prog = Prog Expr
data Expr = EAtom Atom | EAdd Expr Expr | ELet [Bind] Expr
data Atom = Lit Int | Var String
data Bind = Bind String Expr
```

We do not need to define new instances of `Zipper` and `CombineZipper` to implement this example. We use the defined instances for `GZ.Zipper` and `ZipperRose`, which are generic. We only need to derive the `Data` instances for the expression datatypes and define the different *focus* functions in order to provide type information to select the instances. For instance, `focus` is just:

```
focus :: Typeable a ⇒ GZ.Zipper t → Maybe a
focus = GZ.getHole
```

Concerning the attributes, the synthesized attribute *attEval* for expression evaluation is defined in the following way.

```
attEval = close $ evalP `ext` evalE `ext` evalA

evalP zt zp = do p ← focus zp
                 return $ case p of
                   Prog _ → focusEval (zt .$ 1)

evalE zt ze = do e ← focus ze
                 return $ case e of
                   EAtom _ → focusEval (zt .$ 1)
                   EAdd _ _ → focusEval (zt .$ 1) + focusEval (zt .$ 2)
                   ELet _ _ → focusEval (zt .$ 2)

evalA zt za = do a ← focus za
                 return $ case a of
                   Lit x → x
                   Var v → (slookup v . focusIEnv) zt
```

We use the Maybe monad in order to be able to define the different possible alternatives for the attribute. Since the `focus` function will return `Nothing` in case the type of the element in the focus does not coincide with the expected type, we can implement all the alternatives in different functions, in a modular way, and combine them using the `Alternative` instance of `Maybe`. This is performed by the `ext` function, that also applies the two alternative functions to both zippers. Once we have all the definitions, we `close` the attribute, assuming that it will produce a `Just` value.

```
ext f g z t = f z t <|> g z t
close f z t = fromJust (f z t)
```

Inherited attributes like *attIEnv* are computed from the parent to the children. The function `parents` is used to go up in both the zipper of the expression and the zipper of the tables, and get the focus of the first one, to distinguish the different cases.

```
attIEnv = close $ iEnvP `ext` iEnvE `ext` iEnvL `ext` iEnvCpy

iEnvP zt z = do (Prog _,_) ← parents z zt
                return [ ]

iEnvE zt z = do (p,ztp) ← parents z zt
                return $ case p of
                  ELet _ _ → if z .? 2 then focusSEnv ztp
                                       else focusIEnv ztp
                  _        → focusIEnv ztp

iEnvL zt z = do (p,ztp) ← parents z zt
                return $ case (p :: [Bind]) of
                  _ : _ → if z .? 2 then focusSEnv (ztp .$ 1)
                                    else focusIEnv ztp
```

```
iEnvCpy zt _ = return $ (focusIEnv . parent) zt
```

In case of a let expression, the inherited environment is distributed unchanged to the list of bindings, while a new environment (*senv*) produced by the bindings is distributed to the sub-expression. The behaviour is similar for the cons of binding lists, meaning that the expressions in new bindings can use the bindings previously defined.

The synthesized environment attribute *sEnv* is defined to collect the bindings in the list.

```
attSEnv = close $ sEnvE `ext` sEnvL `ext` sEnvB

sEnvE zt ze = do e ← focus ze
                 return $ case e  of
                   ELet _ _ → focusSEnv (zt .$ 1)

sEnvL zt zl = do l ← focus zl
                 return $ case l  :: [Bind]  of
                   _ : _ → focusSEnv (zt .$ 2)
                   []    → focusIEnv zt

sEnvB zt zb = do b ← focus zb
                 return $ case b  of
                   Bind x _ → (x, focusEval (zt .$ 2)) : focusIEnv zt
```

We start at the end of the list (case `[]`) with the inherited environment *ienv*, and then on each cons we add the new binding.

Finally, we define the semantic function using `tabulate`.

```
semProg p  = tabulate focusEval (app (attEval, attIEnv, attSEnv)) p
```

5 Performance

In this section we analyze the performance of zipper-based tabulation in different scenarios, and compare it with existing alternatives. The idea is to identify cases where it is worth using this technique and cases where it is not.

In Section 1 we showed that the execution time of zipper-based versions of functions like Fibonacci is comparable to the obtained using arrays. An elegant aspect of lazy tabulation is that it enables the construction of a data structure consisting of deferred computations, activating only the necessary ones when demanded. This facilitates the representation of functions that utilize "future" computations (i.e., accessing values further ahead in the table) as long as the dependencies between computations are non-circular. We show a function called *Foobonacci* that exhibits these characteristics.

```
foo n | n <= 6           = 1
      | n `mod` 5 == 0 = foo (n-8)
      | n `mod` 5 == 1 = foo (n-9)
      | otherwise        = foo (n+1) + foo (n+2)
```

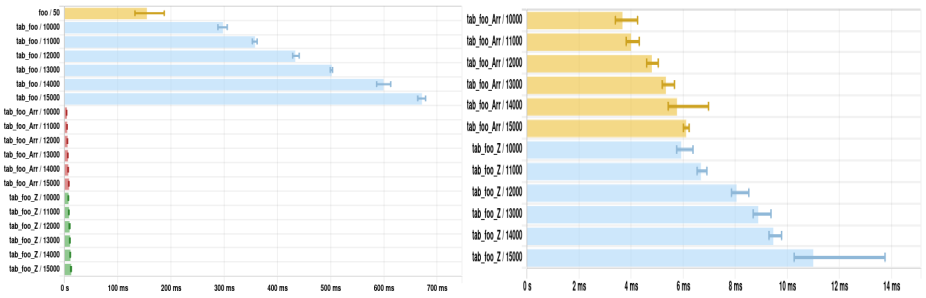


Fig. 5: Performance of Foobonacci

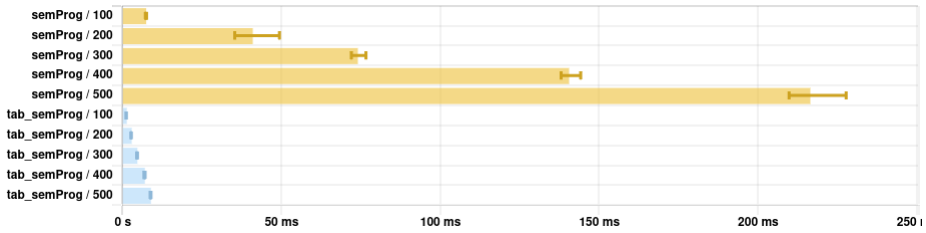


Fig. 6: Performance of `semProg` when tabulation is needed

In Figure 5, it can be seen that also in the case of Foobonacci the zipper version presents much better times than those using lists. We are still quite competitive with the array version, which performs better but it has the flaw that it cannot be produced trivially, since some care has to be taken when choosing the array boundaries. This is an important advantage of using zippers compared to arrays: there is no need to set bounds, allowing us to work with (potentially) infinite structures.

With respect to tree traversals, we have shown in Figure 4 of Section 2, that in the case of *repmin* avoiding duplicated computations has an important gain in performance.

Of course, the use of tabulation also has its costs. On the one side we can observe in Figure 6 the comparison of execution times of a tabulated and a non-tabulated version of `semProg` (from Section 4) when the expression represents a program with the form: `let x1 = 1 ... xn = n in x1 + ... + xn`. In the non-tabulated version, every occurrence of a variable involves recomputing the environment, thus the tabulated version performs significantly better. On the other hand, in expressions where no recomputations are needed, like `let x1 = 1 ... xn = n in 1 ... + n`, the non-tabulated version clearly outperforms the tabulated one (see Figure 7). However, it is important to note that for similar expressions the worst case for the non-tabulated version is approximately 200ms, whereas for the tabulated version is of around 3ms. So, the cost associated with tabulation is worth paying in order to avoid worse performance issues.

The zipper-based approach is useful when long list searches can be substituted by short zipper traversals in the neighbourhood of the focus. In other cases,

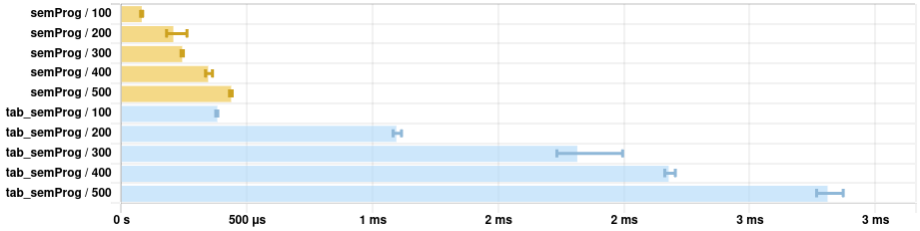


Fig. 7: Performance of `semProg` when tabulation is not needed

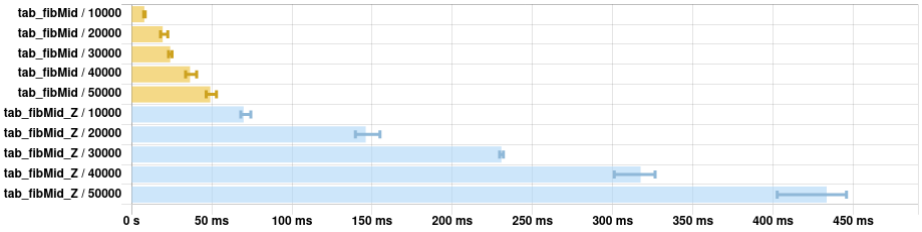


Fig. 8: Performance of `fibMid`

the cost of using a zipper is not justified. To illustrate this, we define another Fibonacci-like function that instead of calling the recursion with $n-1$ and $n-2$ it halves n. Figure 8 shows the results.

```
fibMid 0 = 0
fibMid 1 = 1
fibMid n = fib (n `div` 2) + fib (n `div` 2 -1)
```

Another case where it does not seem wise to use the zipper-based approach is when tabulation involves a matrix, since the zipper is not an efficient structure to move through it. For instance, if the matrix is implemented as a list of rows, moving up and down involves moving all along the actual row to the left and then back right in the upper row. A well-known example of such case is the *knapsack problem*, which involves finding the maximum profit obtained by introducing some subset of n objects with given weights (ws) and profits (ps) in a knapsack with capacity c.

```
knapsack 0 c = 0
knapsack n c | c<ws!n     = knapsack (n-1) c
             | otherwise = max (knapsack (n-1) c)
                               (knapsack (n-1) (c-ws!n) + ps!n)
```

We show in Figure 9 the times of executing `snackpack` with 100 objects, respective weights `[1..100]`, profits `[100..1]`, and capacity 500. It is clearly more efficient to use a list of lists as table than to use a zipper of them.

6 Related Work

Tabulation is a well-known and commonly used optimization technique in the context of dynamic programming. Our work is based on what Bird [2] called *exact tabulation*, in contrast to a more fine grained tabulation technique based

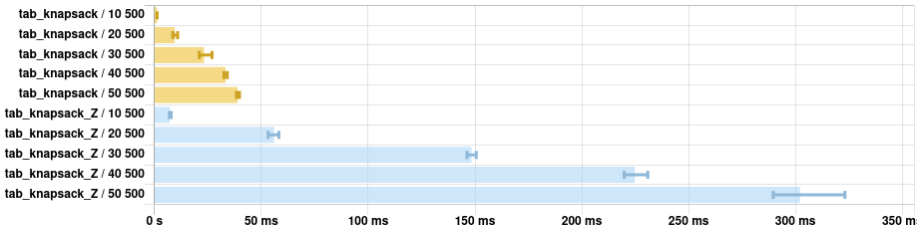


Fig. 9: Performance of `knapsack`

on the analysis of the dependency graph. Pareja-Flores et. al. [12] proposed a transformation tactic to convert recursive functions into tabulated ones based on arrays. In the case of exact tabulation (that they call total tabulation), it corresponds to the technique we showed in the introduction. In our work we adapted the standard tabulation technique to be used with zippers as tables, changing from absolute to relative addressing on the tables. To deal with the genericity of the tables we introduced the classes `Zipper` and `CombineZipper` and a function `zipWithZipper` to construct them. Based on a different approach, Bird [3] investigated the systematic derivation of tabulation versions for recursive functions that satisfy certain conditions. Similar to us he put the focus of his analysis on the tree structure one can form with the successive arguments to the recursive calls.

The use of zippers to embed attribute grammars in a functional setting is proposed in [11] as the ZipperAG embedding, which has been extended with memoization to avoid attribute re-computations [4]. In those works a set of domain-specific zipper-based combinators were introduced to express attribute grammars. Our zipper-based tabulation technique offers for free the expressiveness of those AG combinators. As we showed in sections 2 and 4 we can express AGs directly in our setting without having to introduce AG specific combinators/notation. In our embedding attributes have to be *closed* in order to be evaluated and stored in the table, while ZipperAG implements the attribute calls as function applications. On the other hand, ZipperAG needs to model attribute computations as computations in the State monad in order to perform memoization.

7 Conclusions

We have adapted the tabulation technique to be used with zippers as tables. It showed to be particularly useful to represent recursive functions whose recursive calls are performed on closely situated arguments within the function domain. We have also defined generic instances of the needed ingredients in order to avoid having to define boilerplate code. This technique can also be applied to avoid duplicated computations of functions that use the zipper to traverse tree-shaped terms. This delves to a concise and efficient zipper-based embedding of attribute grammars.

References

1. Adams, M.D.: Scrap your zippers: A generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming. p. 13–24. WGP '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863495.1863499
2. Bird, R.S.: Tabulation techniques for recursive programs. ACM Comput. Surv. **12**(4), 403–417 (dec 1980), https://doi.org/10.1145/356827.356831
3. Bird, R.S.: Zippy tabulations of recursive functions. p. 92–109. MPC '08, Springer-Verlag, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-70594-9_7
4. Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M.: Memoized zipper-based attribute grammars and their higher order extension. Science of Computer Programming **173**, 71–94 (2019), https://doi.org/10.1016/j.scico.2018.10.006
5. Huet, G.P.: The zipper. J. Funct. Program. **7**(5), 549–554 (1997), https://doi.org/10.1017/s0956796897002864
6. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2), 127–145 (June 1968), https://doi.org/10.1007/BF01692511, correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
7. Knuth, D.E.: The genesis of attribute grammars. In: Deransart, P., Jourdan, M. (eds.) Attribute Grammars and their Applications. pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
8. Lämmel, R., Jones, S.P.: Scrap your boilerplate: A practical design pattern for generic programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. p. 26–37. TLDI '03, Association for Computing Machinery, New York, NY, USA (2003), https://doi.org/10.1145/604174.604179
9. Lämmel, R., Jones, S.P.: Scrap more boilerplate: Reflection, zips, and generalised casts. SIGPLAN Not. **39**(9), 244–255 (sep 2004), https://doi.org/10.1145/1016848.1016883
10. Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: Extensible generic functions. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming. p. 204–215. ICFP '05, Association for Computing Machinery, New York, NY, USA (2005), https://doi.org/10.1145/1086365.1086391
11. Martins, P., Fernandes, J.P., Saraiva, J., Van Wyk, E., Sloane, A.: Embedding attribute grammars and their extensions using functional zippers. Science of Computer Programming **132**(P1), 2–28 (Dec 2016), https://doi.org/10.1016/j.scico.2016.03.005
12. Pareja, C., Pena, R., Velázquez-Iturbide, J.Á.: A tabulation transformation tactic using haskell arrays. In: Alpuente, M., Sessa, M.I. (eds.) 1995 Joint Conference on Declarative Programming, GULP-PRODE'95, Marina di Vietri, Italy, September 11-14, 1995. pp. 485–496 (1995)