Memoized Zipper-based Attribute Grammars

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

[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
jas@di.uminho.pt

Abstract. Attribute Grammars are a powerfull, well-known formalism to implement and reason about programs which, by design, are conveniently modular.
In this work we focus on a state of the art Zipper-based embedding of Attribute Grammars and further improve its performance through controlling attribute (re)evaluation by using memoization techniques.
We present the results of our optimization by comparing their impact in various implementations of different, well-studied Attribute Grammars.

Keywords: Embedded domain specific languages, Attribute Grammars, Zipper data structure, Memoization

1 Introduction

Attribute Grammars (AGs) are a declarative formalism that allows us to implement and to reason about programs in a modular and convenient way. This formalism was proposed by Knuth [12] in the late 60s, and a concrete AG relies on a context-free grammar to define the syntax of a language, while adding *attributes* to it so that it is also possible to define its semantics.

AGs have been used in practice not only to specify real programming languages, like for example Haskell [6], but also to specify powerful pretty printing algorithms [22], deforestation techniques [8] and powerful type systems [17].

When programming with AGs, modularity is achieved due the possibility of defining and using different aspects of computations as separate attributes. Attributes are distinct computation units, tipically quite simple and modular, that can be combined into elaborated solutions to complex programming problems. They can also be analyzed, debugged and maintained independently which eases program development and evolution.

AGs have proven to be particularly useful to specify computations over trees: given one tree, several AG systems such as [7, 13, 24] take specifications of which values, or attributes, need to be computed on the tree and perform these computations. The effort put into the creation, improvement and maintenance of these AG systems, however, is tremendous, which often is an obstacle to achieving the success they deserve.

An increasingly popular alternative approach to the use of AGs relies on embedding them as first class citizens of general purpose programming languages [5, 15, 18, 21, 25, 3]. This avoids the burden of implementing a totally new language and associated system by hosting it in state-of-the-art programming languages. We want to exploit the modern constructions and infrastructure that are already provided by those languages and focus on the particularities of the domain specific language that we are developing.

In this paper we focus on the embedding proposed in [15] for Haskell, which we revise in Section 2. This choice is motivated by the fact that this embedding ensures a notation that closely resembles AGs , and even if it relies on a simple navigation engine, it has shown sufficient expressive power to incorporate state-of-the-art extensions to the AG formalism such as the possibility of defining: i) higher-order attributes [20, 26], ii) references [14], iii) circular attributes [15, 21], and iv) bidirectional transformations [16].

In spite of its elegancy and expressive power, the embedding of [15] does not ensure that attributes are computed only once on a given node. As will become clearer in the next section, the same attribute can be evaluated many times on the same node which causes unnecessary overhead on computations.

The first contribution of this paper is that we take the embedding of [15] and show how it can be extended in such a way that all attributes in an AG are evaluated only once. This extension is achieved with a memoization strategy that can systematically be applied to all embedded AGs in the setting of [15]. This contribution is introduced in Section 3.

A second and final main contribution of the paper is that we analyze the impact of memoization, in terms of efficiency, on several well known and well studied AG examples from the literature. This is detailed in Section 4. We conclude in Section 5.

2 Zipper-based Attribute Grammars

In this section we describe by means of an example the embedding of AGs proposed in [15]. The example we consider, which is used as running example throughout the paper, is the *repmin* problem [4]. This is a well-known example that has been extensively used in the literature, for the same reason we have chosen it here: it is a simple, easy to understand problem which clearly illustrates the modular nature of AG and the difficulties on implementing and scheduling its computations. The goal of *repmin* is to transform a binary leaf tree of integers into a new tree with the exact same shape but where all leaves have been replaced by the minimum leaf value of the original tree. Concretely, we consider the following definition of binary leaf trees:

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

```
  -- Inherited
globmin :: AGTree Int
globmin t = case constructor t of
                 C_Root    → locmin  (tree t)
                 C_Leaf _ → globmin (up   t)
                 C_Fork    → globmin (up   t)
  -- Synthesized
locmin :: AGTree Int
locmin t = case constructor t of
               C_Leaf l → l
               C_Fork   → min (locmin (left t)) (locmin (right t))
replace :: AGTree Tree
replace t =  case constructor t of
                 C_Root    → replace (tree t)
                 C_Leaf _ → Leaf  (globmin t)
                 C_Fork    → Fork (replace (left t)) (replace (right t))
```

Fig. 1: Repmin defined using a Zipper-based AG

In order to solve *repmin*, we may define an AG with three attributes: i) one inherited attribute, *globmin*, so that all nodes in a tree may know and use the global minimum of the tree; and two synthesized attributes: ii) *locmin*, to compute the local minimum of each node in a tree, and iii) *replace*, to compute at each node the *repmin* of the tree under it. These attributes should be scheduled according to the computation: we need to find the minimum value contained in the tree with *locmin*, distribute this value across all the nodes of the tree with *globmin* and analyze the structure and traverse the tree to create a new one with *replace*.

In the setting of [15] we may define the AG for *repmin* by the embedding in Haskell shown in Figure 1. We see that, e.g., at a *Leaf* node, the global minimum of a tree is inherited from its parent node (*up t*), and that the local minimum of a *Fork* node is given by the minimum of the local minimums of the child nodes (*left t* and *right t*). Notice that the attributes are represented as functions.

Finally, *repmin* is obtained by computing the *replace* attribute on the topmost node of a tree:

```
repmin :: Tree → Tree
repmin t = replace (mk_AG t)
```

The embedding of [15] relies on the *zipper* data structure [9] to provide the means to navigate on a tree and to define the values of attributes in terms of other attributes on neighbour nodes. An AG computation on a *Tree* is actually a function that takes a *Zipper* and returns the result of the computation:

$\textbf{type}\ \mathit{AGTree}\ a = \mathit{Zipper} \to a$

A zipper can be regarded as a tree together with its context:

$\textbf{type}\ \mathit{Zipper} = (\mathit{Tree}, \mathit{Cxt})$

$\textbf{data}\ \mathit{Cxt} = \mathit{Root} \mid \mathit{Top} \mid L\ \mathit{Cxt}\ \mathit{Tree} \mid R\ \mathit{Tree}\ \mathit{Cxt}$

To construct a zipper, we mark a *Tree* as being at the *Root* node:

$\mathit{mk}_{AG} :: \mathit{Tree} \to \mathit{Zipper}$

$\mathit{mk}_{AG}\ t = (t, \mathit{Root})$

Constructor *Root* is artificially added as a context, since we need to distinguish the topmost tree from all the other (sub)trees. In fact, we need to bind the local minimum of the topmost tree with the global minimum of that same tree.[1]

In order to inspect the node under focus, we define a new datatype, with an associated pattern-matching function:

$\textbf{data}\ \mathit{Cons} = C_{\mathit{Root}} \mid C_{\mathit{Fork}} \mid C_{\mathit{Leaf}}\ \mathit{Int}$

$\mathit{constructor} :: \mathit{Zipper} \to \mathit{Cons}$

$\mathit{constructor}\ (\_, \mathit{Root}) = C_{\mathit{Root}}$

$\mathit{constructor}\ (\mathit{Leaf}\ l, \_) \quad = C_{\mathit{Leaf}}\ l$

$\mathit{constructor}\ (\mathit{Fork}\ \_\ \_, \_) = C_{\mathit{Fork}}$

Now, we have defined all it takes to navigate through concrete trees. Going down on a (non topmost) tree, for example, can be implemented as follows:

$\mathit{left}\ :: \mathit{Zipper} \to \mathit{Zipper}$

$\mathit{left}\ (\mathit{Fork}\ l\ r, c) = (l, L\ c\ r)$

$\mathit{right}\ :: \mathit{Zipper} \to \mathit{Zipper}$

$\mathit{right}\ (\mathit{Fork}\ l\ r, c) = (r, R\ l\ c)$

while trying to go down the topmost tree simply creates a zipper whose (real) context is *Top*:

$\mathit{tree}\ :: \mathit{Zipper} \to \mathit{Zipper}$

$\mathit{tree}\ (t, \mathit{Root}) = (t, \mathit{Top})$

Going up on a location on a tree may also be performed in a simple way, which actually inverts the behavior of functions *left*, *right* and *tree* shown above:

$\mathit{up} :: \mathit{Zipper} \to \mathit{Zipper}$

$\mathit{up}\ (t, \mathit{Top}) = (t, \mathit{Root})$

$\mathit{up}\ (t, L\ c\ r) = (\mathit{Fork}\ t\ r, c)$

$\mathit{up}\ (t, R\ l\ c) = (\mathit{Fork}\ l\ t, c)$

Finally, we define a function that applies a transformer to the tree under focus:

$\mathit{modify} :: \mathit{Zipper} \to (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Zipper}$

$\mathit{modify}\ (t, c)\ f = (f\ t, c)$

Despite its clear syntax and expressive power, the described embedding does not ensure that attributes are computed only once on a given node. We may notice that on the *repmin* solution presented earlier, the global minimum of a tree is computed as many times as the number of leaves that tree has.

As a concrete example of this, in Figure 2 we show the function call chains that activate the computation of attributes *replace* on leaves labelled with 1 (left) and 2 (right). As defined earlier, *replace* in a leaf will call *globmin* on the same node, then *globmin* will call *globmin* at its parent, and so on, calling then *locmin* from the root to the leaves. So, while in the first computation of *replace* every attribute is computed only once, in the second case we see that some calls to

[1] This binding can be seen in the definition of *globmin*, in $C_{\mathit{Root}} \to \mathit{locmin}\ (\mathit{tree}\ t)$.

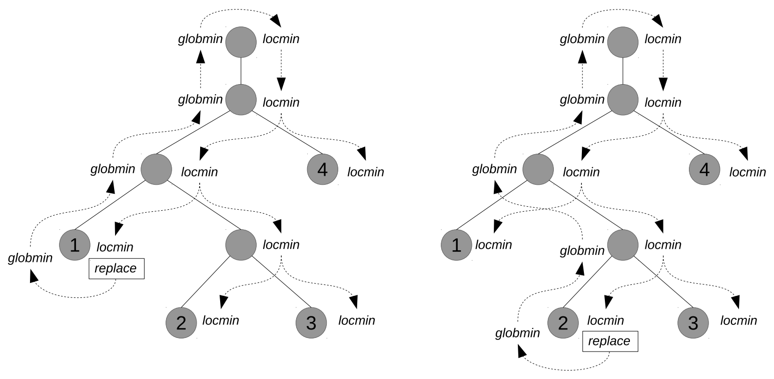


Fig. 2: Function (attribute) calls to evaluate *replace* in a leaf

globmin are new, but then we reach a point in which we start to repeat the steps that have already been taken, therefore duplicating computations and creating an unnecessary overhead, which grows proportionally with the number of leaves.

One contribution of this paper is the introduction of a strategy for solving this efficiency issue, which is presented in the next section. This is achieved by memoizing attribute computations, improving that way the performance of the solution, and allowing us to say that we provide, under a formal perspective, a real attribute grammar embedding.

Although we use *repmin* as a running example, the strategy we study has also been applied and assessed in other problems that are well know in the AG domain, some of which are presented in Section 4.

3 Memoized AGs

As an alternative to the solution given in Figure 1 we present the one in Figure 3. The structure of the new code is quite similar to the old one. Without delving into details now, it can be seen that the main differences are the use of a *memo* function, which introduces *memoization* in the evaluation of the attribute grammar, and the use of **let** to pass around a changing tree.

In order to avoid attribute recomputations, we attach a table to each node of a tree to store the value of the attributes associated to the node. We do so by transforming the original tree into a new one of same shape and with a memo table attached to each node. The new tree type is now parametric on the type m of the memo table.

$$\begin{aligned}\textbf{data}\ \mathit{Tree}_m\ m &= \mathit{Fork}_m\ m\ (\mathit{Tree}_m\ m)\ (\mathit{Tree}_m\ m)\\ &\mid\ \mathit{Leaf}_m\ m\ \mathit{Int}\end{aligned}$$

A new version of the Zipper has to be defined to be able to navigate through a tree of type Tree_m.

```
  -- Inherited
globmin :: (Memo Globmin m Int, Memo Locmin m Int) ⇒ AGTree_m Int
globmin = memo Globmin $ λz → case constructor_m z of
                                    C_Root   → locmin .@. tree_m z
                                    C_Leaf _ → globmin `atParent` z
                                    C_Fork   → globmin `atParent` z
  -- Synthesized
locmin :: (Memo Locmin m Int) ⇒ AGTree_m Int
locmin = memo Locmin $ λz → case constructor_m z of
                                 C_Leaf v → (v, z)
                                 C_Fork   → let (left,  z')  = locmin .@. left_m   z
                                                (right, z'') = locmin .@. right_m  z'
                                            in  (min left right, z'')
replace :: (Memo Replace m Tree, Memo Globmin m Int, Memo Locmin m Int)
        ⇒ AGTree_m Tree
replace = memo Replace $ λz → case constructor_m z of
                                 C_Root   → replace_m .@. tree_m z
                                 C_Leaf _ → let (mini, z') = globmin z
                                            in  (Leaf mini, z')
                                 C_Fork   → let (l, z')  = replace .@. left_m   z
                                                (r, z'') = replace .@. right_m  z'
                                            in  (Fork l r, z'')
```

Fig. 3: Repmin defined using memoization

```
type Zipper_m m = (Tree_m m, Cxt_m m)
data Cxt_m m = Root_m | Top_m
             | L_m m (Cxt_m m)  (Tree_m m)
             | R_m m (Tree_m m) (Cxt_m m)
```

The combinators $mkAG_m$, $constructor_m$, $tree_m$, $left_m$, $right_m$, up_m and $modify_m$ that work on $Zipper_m$ are analogous to the ones defined in Section 2 for the original *Zipper* type. For example, up_m is defined as:

```
up_m :: Zipper_m m → Zipper_m m
up_m (t, Top_m)       = (t, Root_m)
up_m (t, L_m m c r) = (Fork_m m t r, c)
up_m (t, R_m m l c) = (Fork_m m l t, c)
```

3.1 Memo Tables

A memo table will contain *Maybe* elements corresponding to the attribues, where *Nothing* is used to mean that the value of an attribute has not been computed yet. In our example, we store *Maybe* values for the attributes Globmin, Locmin and Replace.

We define singleton datatypes to refer to each attribute in a table:

```
data Globmin = Globmin
data Locmin  = Locmin
data Replace = Replace
```

By means of a multi-parameter type class *Memo* we define functions to lookup and modify the value (of type a) of a given attribute *att* in a memo table of type m.

```
class Memo att m a where
  mlookup :: att → m → Maybe a
  mmodify :: att → (Maybe a → Maybe a) → m → m
```

The intended meaning of *mmodify att f m* is the update of the value v of attribute *att* stored in the table m by $f\ v$. The benefit of defining this class is that we can have memoized implementations of AGs that are generic in the representation of the memo tables.

There are different alternatives in how we can implement a memo table. One possibile representation is in terms of tuples. In our example, the tuple stores values corresponding to Globmin (*Int*), Locmin (*Int*) and Replace (*Tree*).

```
type MemoTable = (Maybe Int, Maybe Int, Maybe Tree)
```

The use of tuples to represent memo tables imposes an important drawback because it requires to close the universe of attributes for defining the tuple corresponding to the memo table. Consequently, the addition of a new attribute to the AG leads to the redefinition of the memo table and its associated operations. In other words, the solution with tuples is not extensible.

One way to solve this problem is by replacing tuples by some implementation of extensible records, like the heterogeneous strongly typed lists [11] defined in the HList[2] library. In our repository[3] we include an alternative version that represents the memo tables as extensible records.

Once we have decided the representation of the memo table we are in conditions to define an instance of the *Memo* class for each attribute. For example, the instance for *Globmin* for the representation in terms of tuples is as follows:

```
instance Memo Globmin MemoTable Int where
  mlookup _   (g, _, _) = g
  mmodify _ f (g, l, r)  = (f g, l, r)
```

A $Tree_m$ can be generated from an input tree by attaching a given memo table to each node.

[2] https://hackage.haskell.org/package/HList
[3] https://hackage.haskell.org/package/ZipperAG

$build_m :: Tree \to m \to Tree_m\ m$
$build_m\ (Fork\ l\ tr)\ mt = Fork_m\ mt\ (build_m\ l\ mt)\ (build_m\ r\ mt)$
$build_m\ (Leaf\ n)\quad mt = Leaf_m\ mt\ n$

We make a final remark concerning the representation of memo tables. Our representation assumes uniformity on all nodes of the AG in the sense of all having the same attributes. However, this is not the case in every AG. Different types of nodes may have different attributes and consequently different types of memo tables. To admit this case one possible solution is to declare *MemoTable* as a sum type with one type of memo table for each kind of node:

data $MemoTable = MTFork\ MemoTableFork \mid MTLeaf\ MemoTableLeaf$

It is then necessary to define corresponding instances of the *Memo* class taking into account the alternative memo tables.

3.2 Attribute computation

An attribute computation computes a value, as before, but now it may also apply modifications to memo tables contained in the tree:[4]

type $AGTree_m\ m\ a = Zipper_m\ m \to (a, Zipper_m\ m)$

The function *memo*, used in every attribute definition of Figure 3, is who puts the memoization mechanism to work. It takes as input a reference to an attribute and an $AGTree_m$, representing the computation of that attribute, and returns as result a new $AGTree_m$ where the computation of the attribute is memoized.

```
memo :: Memo attr m a ⇒ attr → AGTree_m m a → AGTree_m m a
memo attr eval z =
  case mlookup attr (getMemoTable z) of
    Just v    → (v, z)
    Nothing → let (v, z′) = eval z
                  in  (v, modify_m z′ (mmodify attr (const $ Just v)))
```

First of all, the memo table is obtained (by *getMemoTable*). Then the given attribute is searched in the memo table to see whether it was already computed. In the affirmative case, the stored value of the attribute is directly returned. Otherwise, we have to compute the value of the attribute at the current location of the zipper and modify the $Tree_m$ by storing the computed value in the corresponding memo table. Notice the use of $modify_m$ to update the $Zipper_m$ that will be passed to future computations.

One effect of attribute computation by memoization is a continuos movement of the computation focus. This means that the location where the computation of an attribute is taking place is continuosly changing. Changes in the computation focus correspond to location changes in the zipper. Those movements in the zipper need to be taken into account when defining the computation of an

[4] This could also be represented in terms of a *State* monad, but we will not take that alternative in this paper.

attribute because in some cases it is neccesary to return to the original location after moving. To see an example suppose we implement *locmin* of Figure 3 in the following way:

```
locmin = memo Locmin $
              λz → case constructor_m z of
                      C_Leaf v → (v, z)
                      C_Fork   → let (left,  z')  = locmin (left_m z)
                                     (right, z'') = locmin (right_m z')
                                 in  (min left right, z'')
```

In the C_{Fork} case, the focus is first moved to the left child where *locmin* is computed. Then, the intention is to compute *locmin* at the right child of the original Fork. However, this is not the case, since it is actually computed at the right child of the left child of the original Fork (if that location even exists). In summary, this definition of *locmin* is not correct. The reason of the failure is that once we move the focus to another position, using e.g. $left_m$ or $right_m$, it does not return to the original one.

To cope with this problem we define two new combinators (.@.) and *atParent* to move the focus of the $Zipper_m$ to an immediate position to compute an attribute there, returning the focus to the original location afterwards. By using (.@.) an attribute is computed in the given child, and then the focus goes back to the parent using up_m:

```
(.@.) :: AGTree_m m a → AGTree_m m a
eval .@. z = let (v, z') = eval z
             in  (v, up_m z')
```

Moving the focus to the parent adds the complication of knowing the position of the child to which we have to return. This is easily solved by inspecting the context of the zipper from which we started.

```
atParent eval z = (v, (back z) z')
  where
    (v, z') = eval (up_m z)
    back (_, Top_m)       = tree_m
    back (_, L_m _ _ _)   = left_m
    back (_, R_m _ _ _)   = right_m
```

Finally, to evaluate the AG defined in Figure 3 we compute *replace* at the initial $Tree_m$ (with empty tables at each node), ignoring the final $Tree_m$.

```
repmin :: Tree → Tree
repmin t = fst (replace (mkAG_m (build_m t emptyMemo)))
```

If, for example, we adopt the memo table representation in term of tuples then the empty table for this AG is given by:

```
emptyMemo = (Nothing, Nothing, Nothing)
```

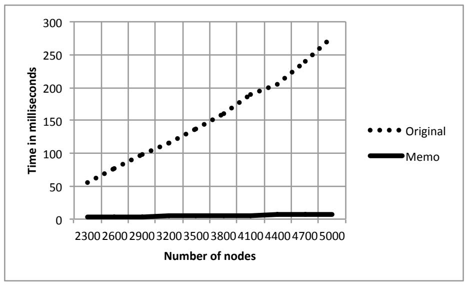


Fig. 4: Performance of the *repmin* implementations.

4 Results

In this section, we assess in terms of efficiency the memoization approach we followed in this paper against the original, non-optimized embedding of [15]. For this assessment, we test the optimized and non-optimized versions of Repmin together with three well known AG examples from the literature. The results are presented as running times and memory consumption.

For the benchmarks we are presenting in this section, we compiled the different approaches with the Glasgow Haskell Compiler (ghc), version 7.8.4, using the `-O2` optimization flag. The computer used was a `1.3 GHz Intel Core i5` with `8 GB 1600 MHz DDR3` RAM memory (mid 2013 stock MacBook Air with RAM upgrade).

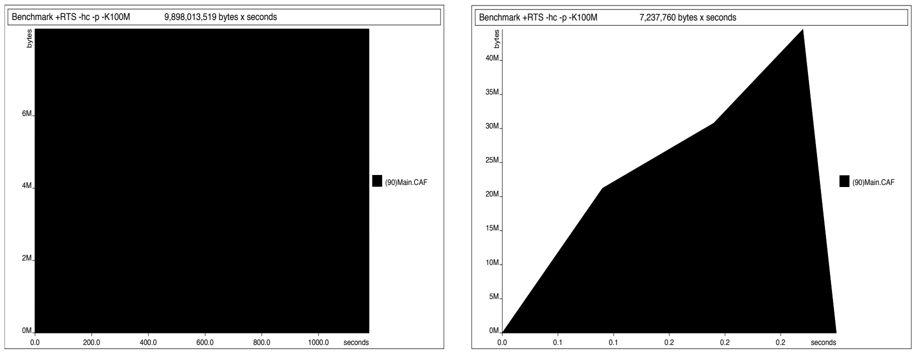
4.1 Repmin

We have started by benchmarking the running example presented throughout this paper. For this test, we used increasingly larger balanced binary leaf trees with a number of nodes ranging from 2300 to 5000, represented on Figure 4 in the x-axis.

The performance results of the implementations with and without memoization allow us to observe that the memoized version significantly improves the performance of the original version. Indeed, when we reach the 5000 nodes there is a clear gap in the time required to run *repmin* between the original and the memoized versions.

Another interesting result is how well the memoized version scales. As we grow from 2300 to 5000 nodes, almost 50%, the memoized shows only a slight increase in running time, while the original approach takes proportionally more and more processing time.

The use of memoization strategies in programming often trades off memory consumption to achieve better runtime performance. This is also evidenced from the memory consumption comparison we performed on the different implementations of Repmin, which is presented in Figures 5a and 5b. Both times we ran repmin with a balanced tree with 150,000 nodes.



(a) Non memoized. (b) Memoized.

Fig. 5: Heap Profile on Repmin (values in Mbytes).

As expected, we observe that it is the original version that throughout its execution has the lowest peak of memory consumption, of slightly more than 8 Mbytes. And it is the memoized version, which is the fastest in terms of runtime performance, that reaches the highest peak of consumed memory, of around 45 Mbytes. A large difference but a burden expected by the use of the memo tables.

It is worth mentioning the time gap between the two versions. The original took around 1500 seconds (around 25 minutes) while the memoized version took around 0.3 seconds. Even though these tests cannot be compared to the performance ones of Figure 4, because of the overhead introduced when analysing memory consumption (by the ghc profiler), among themselves there is a huge gap in run time, confirming the exponential behavior of the non memoized version.

A final note, as mentioned earlier, the original implementation of Repmin is an extremely heavy example of semantics requiring a large number of necessary recomputations of attributes (the minimum value of the tree is constantly being required). So, it comes as no surprise that the memoized versions perform significantly better here. The next examples are focused on real situations.

4.2 Algol-68 Scope Rules

In this section we benchmark an implementation of the Algol 68 scope rules [19, 8, 15]. Algol 68 holds central characteristics of widely-used programming languages, such as a structured layout and mandatory but unique declarations of names which are used.

The semantics requirements are therefore the same as some real examples, like the ones on the Eli system [10] (to define a generic component for the name analysis task of a compiler), or the `let-in` construct of the Haskell programming language.

Algol 68 is a simple block structure language that does not require a *declare-before-use* scope rule discipline. A program consists of a block with a list con-

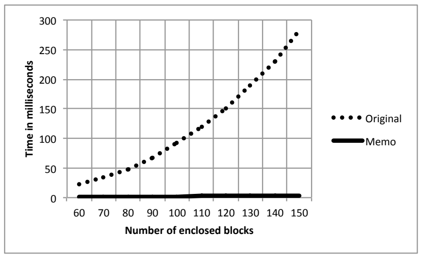


Fig. 6: Performance of the Algol implementations.

taining either use or declaration of names, or a nested block. An example of a program is:

$$
\begin{aligned}
p = [\,&use'\ y; decl'\ x;\\
&\quad [\,decl'\ y; use'\ y; use'\ w;\,]\\
&decl'\ x; decl'\ y;\,]
\end{aligned}
$$

In this language a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x. In the latter case, the definition of x in the local scope hides the definition in the global one. In a block an identifier may be declared at most once.

According to these rules, p above contains two errors: a) at the outer level, the variable x has been declared twice, and b) the use of the variable w, at the inner level, has no binding occurrence at all.

Implementing a validator for Algol implies not only checking each individual block for double declarations of variables, but also constantly analysing outer blocks for the declaration of variables whose definition can not be found in the current block, forcing multiple tree traversals.

In Figure 6, we show the results we obtained when running the different implementations on Algol programs with an increasing number of enclosing blocks. The x-axis of Figure 6 ranges from 60 to 150 enclosing blocks, an increase of more than 50%. Similarly to the previous examples, memoization shows better processing times.

4.3 HTML Table Formatter

We now analyze an example from [22]: we want to format HTML style tables. Namely, we want our AG to receive an abstract data type of an HTML table and to print a geometrically well defined table. Figure 7 shows an example of a possible input (left) and correspondent output (right).

Notice that in the output, all the lines have the same number of columns and the columns have the same length. None of these features are required in the HTML language.

An entry in the table can be a string or a nested table, thus, the straightforward algorithm to express this table formatting requires two traversals and

⟨TABLE⟩
⟨TR⟩⟨TD⟩$^{1}_{14}$`The first line` ⟨/TD⟩⟨TD⟩$^{1}_{4}$`of a` ⟨/TD⟩⟨/TR⟩
⟨TR⟩⟨TD⟩$^{7}_{12}$⟨TABLE⟩
⟨TR⟩⟨TD⟩$^{1}_{4}$`This` ⟨/TD⟩⟨TD⟩$^{1}_{2}$`is` ⟨/TD⟩⟨/TR⟩
⟨TR⟩⟨TD⟩$^{1}_{7}$`another` ⟨/TD⟩⟨TD/⟩⟨/TR⟩
⟨TR⟩⟨TD⟩$^{1}_{5}$`table` ⟨/TD⟩⟨TD/⟩⟨/TR⟩
⟨/TABLE⟩
⟨/TD⟩⟨TD⟩$^{1}_{5}$`table` ⟨/TD⟩⟨/TR⟩
⟨/TABLE⟩

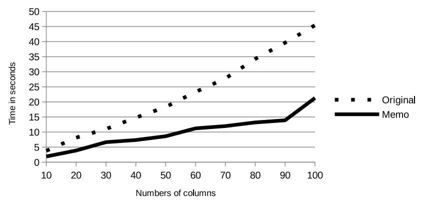
```
|--------------------|
|The first line|of a |
|--------------------|
||----------|  |table|
||This   |is|  |     |
||----------|  |     |
||another|  |  |     |
||----------|  |     |
||table  |  |  |     |
||----------|  |     |
|--------------------|
```

Fig. 7: HTML Table Formatting

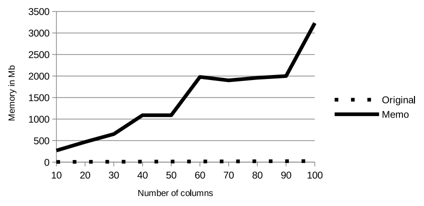
the definition of gluing data types to pass the width/height (blue subscripts / superscripts in Figure 7) of nested table from the first to the second traversal. Simplifying, it is required to know the sizes of inner tables in order to resize the outer ones.

To test this AG, we computed trees representing HTML tables with the same number of rows (50) and an increasing number of columns. All the cells of the tables include the same text, excepting for the ones in the last column, which include nested tables with, recursively, the same shape but half the number of rows and columns of the containing table. The results are presented in figures 8a (time) and 8b (memory consumption), where the x-axis represents the number of columns, ranging from 10 to 100.

It can be observed from these results that, although the reduction in execution time is great, the memoized evaluator does consume much more memory. Note that, for large inputs this evaluator produces large strings. As a result, in the memoized version, partial results are kept in the memo table: large strings in this case. As usual in memoization techniques the gain in runtime is obtained by using additional memory: the memo table. In fact, memory consumption can result in a scalability problem in certain cases. To reduce this problem several AG techniques to purge entries from a memo table have been proposed [19], which can be used in our zipper-based setting.



(a) Time



(b) Memory

Fig. 8: Performance of the HTML table formatters.

5 Conclusion

This paper shows how memoization is introduced in zipper-based embeddings of AGs. Regarding the programs we finally end up with, we argue that they maintain the elegance of the embedding we build upon, and in most cases show better performance, often by various different orders of magnitude.

There is a range of AG applications where this technique does not necessarily yield an advantage. For example, AGs that only have one tree traversal or heavily rely on local attributes or semantic operations on their leaves should not (greatly) benefit from the use of memoization. However, real applications of AGs do not fit into this category. In fact, we have seen through a series of standard AG examples that there is a range of problems where memoization provides real noticeable benefits modulo the memory consumption.

As a possible direction of future research, we would like to test the approach suggested here with other embeddings of AGs such as the ones of [23, 1, 2]. This comparison should be performed whenever possible (for example, it might be hard to perform with specific AG systems such as [7, 13, 24]), but other embeddings have different strategies to deal with attribute recomputation (for example, lazy evaluation). Further tests are required to see how this compares to our memoized approach.

Another line of work, could be the use of type-level programming techniques to make the AG system extensible in the sense of adding new productions to the grammars.

References

1. Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Yet another implementation of attribute evaluation. Research Report RR-6315, INRIA, 2007.
2. Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Attribute grammars as recursion schemes over cyclic representations of zippers. *Electronic Notes Theory Computer Science*, 229(5):39–56, 2011.
3. Florent Balestrieri. *The Productivity of Polymorphic Stream Equations and The Composition of Circular Traversals*. PhD thesis, University of Nottingham, 2015.
4. Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf*, 21:239–250, 1984.
5. Oege de Moor, Kevin Backhouse, and Doaitse Swierstra. First-Class Attribute Grammars. In *3rd. Workshop on Attribute Grammars and their Applications*, pages 1–20, Ponte de Lima, Portugal, 2000.
6. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell Symposium*, pages 93–104, 2009.
7. Atze Dijkstra and Doaitse Swierstra. Typing Haskell with an Attribute Grammar (Part I). Technical Report UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University, 2004.
8. João Paulo Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *Symposium on Partial Evaluation and Program Manipulation*, pages 102–111. ACM, 2007.
9. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

10. Uwe Kastens, Peter Pfahler, and Matthias T. Jung. The Eli System. In *Int. Conference on Compiler Construction*, pages 294–297. Springer-Verlag, 1998.
11. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Workshop on Haskell*, pages 96–107. ACM, 2004.
12. Donald Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2), June 1968. *Correction: Mathematical Systems Theory* 5 (1), March 1971.
13. Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In *International Conference on Compiler Construction*, pages 298–301. Springer-Verlag, 1998.
14. Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Science Computer Programming*, 68(1):21–37, 2007.
15. Pedro Martins, João Paulo Fernandes, and João Saraiva. Zipper-based attribute grammars and their extensions. In *Brazilian Symposium on Programming Languages*, pages 135–149. Springer, 2013.
16. Pedro Martins, João Paulo Fernandes, João Saraiva, Eric Van Wyk, and Anthony Sloane. Embedding attribute grammars and their extensions using functional zippers. *Science of Computer Programming*, 2016. In Press.
17. Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Iterative type inference with attribute grammars. In *International Conference on Generative Programming*, pages 43–52. ACM, 2010.
18. Ulf Norell and Alex Gerdes. Attribute Grammars in Erlang. In *Workshop on Erlang*, 2015, pages 1–12. ACM, 2015.
19. João Saraiva. *Purely Functional Implementation of Attribute Grammars.* PhD thesis, Utrecht University, The Netherlands, December 1999.
20. João Saraiva and S. Doaitse Swierstra. Generating Spreadsheet-Like Tools from Strong Attribute Grammars. In *International Conference on Generative Programming*, pages 307–323. Springer, 2003.
21. Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. *Electronic Notes in Theoretical Computer Science*, 253(7):205–219, 2010.
22. Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206. Springer-Verlag, 1999.
23. Tarmo Uustalu and Varmo Vene. Comonadic functional attribute evaluation. Trends in Functional Programming, pages 145–162. Intellect Books (10), 2005.
24. Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.
25. Marcos Viera, Doaitse Swierstra, and Wouter Swierstra. Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In *International Conference on Functional Programming*, pages 245–256. ACM, 2009.
26. Harald Vogt, S. Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, June 1989.