Science of Computer Programming 173 (2019) 71–94

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Memoized zipper-based attribute grammars and their higher order extension

João Paulo Fernandes [a,*], Pedro Martins [b,*], Alberto Pardo [c,*], João Saraiva [d,*], Marcos Viera [c,*]

[a] CISUC & Universidade de Coimbra, Portugal
[b] University of California, Irvine, USA
[c] Universidad de la Republica, Uruguay
[d] HASLab/INESC TEC & Universidade do Minho, Portugal

ARTICLE INFO

Article history:
Received 19 March 2017
Received in revised form 24 September 2018
Accepted 8 October 2018
Available online 12 November 2018

Keywords:
Embedded domain specific languages
Zipper data structure
Memoization
Attribute grammars
Functional programming

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 classic attribute grammars and higher-order attribute grammars. We improve their execution performance through controlling attribute (re)evaluation by means of memoization techniques. We present the results of our optimizations by comparing their impact in various implementations of different, well-studied, attribute grammars and their Higher-Order extensions.

1. Introduction

Attribute Grammars (AGs) are a declarative formalism that was proposed by Knuth [23] in the late 60s and allows the implementation and reasoning about programs in a modular and convenient way. A concrete AG relies on a context-free grammar to define the syntax of a language, and on attributes associated to the productions of the grammar to define the semantics of that language. AGs have been used in practice to specify real programming languages, like for example Haskell [10], as well as powerful pretty printing algorithms [40], deforestation techniques [14] and powerful type systems [28].

When programming with AGs, modularity is achieved due to the possibility of defining and using different aspects of computations as separate attributes. Attributes are distinct computation units, typically 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 [32,11,24,43] take specifications of which values, or attributes, need to be computed on the tree and perform these computations. The design and coding efforts 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.

* Corresponding authors.
E-mail addresses: jpf@dei.uc.pt (J.P. Fernandes), pribeiro@uci.edu (P. Martins), pardo@fing.edu.uy (A. Pardo), saraiva@di.uminho.pt (J. Saraiva), mviera@fing.edu.uy (M. Viera).

https://doi.org/10.1016/j.scico.2018.10.006
0167-6423/

An increasingly popular alternative approach to the use of AGs relies on embedding them as first class citizens of general purpose programming languages [9,26,29,38,44,5]. This avoids the burden of implementing a totally new language and associated system by hosting it in state-of-the-art programming languages. Following this approach one then exploits the modern constructions and infrastructure that are already provided by those languages and focus on the particularities of the domain specific language being developed.

In this paper we focus on the embedding of AGs in Haskell proposed in [26], which we revise in Section 2. This embedding ensures a notation that closely resembles AGs, and even though 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 not only defining classical (first-order) attribute grammars but also Higher-Order Attribute Grammars (HOAGs) [36,26,27]. This elegant AG embedding, however, has a severe performance problem: it does not ensure that attributes are computed only once on a given node. As will become clearer in the next section, the same attributes can be evaluated many times on the same node which causes unnecessary overhead on computations. Also, under a strictly formal point of view, recomputation of attribute values is against the principles of AG systems.

To avoid unnecessary computation, this paper makes an essential contribution: we incorporate memoization into the zipper-based embedding of AGs. Thus, we introduce memoized zippers to provide a navigational engine that guarantees that all attributes in classical AGs are evaluated only once. This contribution is presented in Section 3.

Classical AGs, however, require that the underlying abstract tree is fixed during attribute evaluation. This limits the expressiveness of the AG formalism. For example, when we need to define an algorithm that cannot be easily expressed on the inductive structure of the underlying abstract tree, a more appropriate structure can not be computed first! HOAGs offer this expressiveness by extending AGs with higher-order attributes, which are attributes that are trees, and as such, we may in turn associate attributes to them. HOAGs provide component based programming style for AGs [34]: AG components are easily *plugged-into* an AG specification via higher-order attributes. In Section 4 we show how memoization can be extended to HOAGs, in order to avoid attribute recomputation in higher-order attributes. Our approach memoizes all attribute values in memo tables, thus avoiding attribute recomputation.

Our motivation to avoid attribute recomputation is aligned with the goals of the seminal work on incremental computation that was originally proposed by [31]. Incremental computation, however, is concerned with avoiding attribute recomputation when the underlying tree structure changes; and so, otherwise, even attributes whose values can be shown not to change as a consequence of the structure change would have to be recomputed. Instead, we seek to avoid attribute recomputation in the first decoration of the tree structure.[1] Actually, as other attribute grammar systems such as [12,38], our approach is closely related with the classical approach of [18], in the sense that, similarly, we evaluate attributes using a memoized recursive algorithm.

In the context of attribute evaluators, it has been shown that their performance can be further improved by using a selective memoization strategy [37,39,7]. In Section 3.3 we show how selective memoization is achieved in our zipper-based setting.

Our embedding is parametric in type of the memo table, and as a consequence, we can easily change the memoization strategy without having to change the HOAG specification. Our goal of having a memoization strategy that is not coupled with the attribute grammar specification itself is shared with [39], where it is possible to choose which attributes to memoize without changing the underlying specification.

To assess the impact of memoization, we present extensive benchmarks both for classical AGs (Section 3.4) and for HOAGs (Section 4.3). We also, present preliminary results of selective memoization evaluators both for first-order and higher-order AGs.

Our benchmarks show that our memoized zipper-based engine improves the performance of both AGs and HOAGs, often by various different orders of magnitude. We believe the detailed benchmarks we have conducted are relevant to show up to which extent the memoized versions are better, and that it is possible to combine lazy evaluation and memoization without compromising efficiency.

The implementation of the AGs used for the benchmarks are available at a GitHub repository.[2]

In Section 5 we discuss related work whereas in Section 6 we present the conclusions.

2. Zipper-based attribute grammars

In this section we describe by means of an example the embedding of AGs in Haskell we proposed in [26]. The example we consider, which is used as running example throughout the paper, is the *repmin* problem [6]. 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.

[1] Actually, the current version of our work does not support tree structure editions, so we consider the first and single decoration of attributed trees.

[2] https://github.com/pedromartins4/memo-AG.

```
  – 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 implement *repmin*, we consider the following definition of binary leaf trees:

$$\textbf{data}\ Tree = Leaf\ Int \mid Fork\ Tree\ Tree$$

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 [26] we may define the AG for *repmin* by the embedding in Haskell shown in Fig. 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:

$$repmin :: Tree \to Tree$$
$$repmin\ t = replace\ (mk_{AG}\ t)$$

The embedding of [26] relies on the *zipper* data structure [15] to provide the mechanisms 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:

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

Navigation on a tree is possible with a zipper due to the way tree locations are represented: a *location* in a tree consists of a subtree (regarded as the current focus of attention) together with the context surrounding that subtree (given by the path from the subtree to the root of the tree).

$$\textbf{type}\ Zipper = (Tree, Cxt)$$
$$\textbf{data}\ Cxt = Root \mid Top \mid L\ Cxt\ Tree \mid R\ Tree\ Cxt$$

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

$$mk_{AG} :: Tree \to Zipper$$
$$mk_{AG}\ t = (t, Root)$$

Constructor *Root* has been added artificially as a context as we need to identify and treat the topmost tree t in two different ways: i) as the *de facto* topmost tree, and in such case we have the zipper $(t, Root)$; and ii) as any other (sub)tree, and in such case we have the zipper (t, Top). This distinction is necessary, for example, as we need to bind the local minimum of the topmost tree with the global minimum of that same tree.[3] For the calculation of the global minimum, we

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

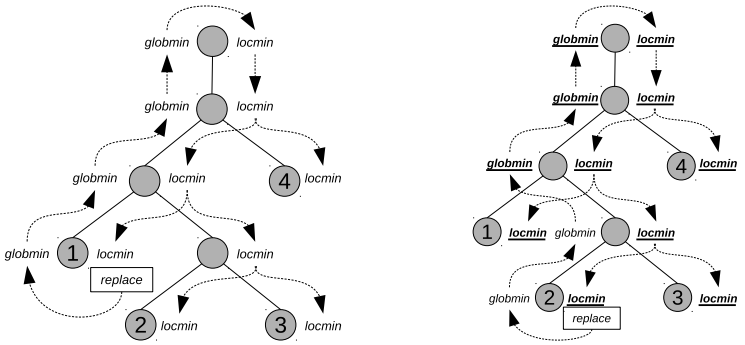


Fig. 2. Function (attribute) calls to evaluate *replace* in a leaf.

need to identify whether we are at the topmost tree, but once having reached that tree, we calculate its local minimum as we do for any other tree.

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{Cons} = \mathit{C_{Root}} \mid \mathit{C_{Fork}} \mid \mathit{C_{Leaf}}\ \mathit{Int}\\
&\mathit{constructor} :: \mathit{Zipper} \to \mathit{Cons}\\
&\mathit{constructor}\ (\_, \mathit{Root}) \quad\ = \mathit{C_{Root}}\\
&\mathit{constructor}\ (\mathit{Leaf}\ l, \_) \quad = \mathit{C_{Leaf}}\ l\\
&\mathit{constructor}\ (\mathit{Fork}\ \_\ \_, \_) = \mathit{C_{Fork}}
\end{aligned}
$$

Notice that $\mathit{constructor}\ (t, \mathit{Top})$ will pattern match on t, and will check whether t is a Leaf or a Fork.[4]

Tree navigation is based on the following primitives. Going down on a (non topmost) tree can be implemented as follows:

$$
\begin{aligned}
&\mathit{left} \quad :: \mathit{Zipper} \to \mathit{Zipper}\\
&\mathit{left} \quad (\mathit{Fork}\ l\ r, c) = (l, L\ c\ r)\\
&\mathit{right}\ :: \mathit{Zipper} \to \mathit{Zipper}\\
&\mathit{right}\ (\mathit{Fork}\ l\ r, c) = (r, R\ l\ c)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{tree}\ :: \mathit{Zipper} \to \mathit{Zipper}\\
&\mathit{tree}\ (t, \mathit{Root}) = (t, \mathit{Top})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{up} :: \mathit{Zipper} \to \mathit{Zipper}\\
&\mathit{up}\ (t, \mathit{Top})\ = (t, \mathit{Root})\\
&\mathit{up}\ (t, L\ c\ r) = (\mathit{Fork}\ t\ r, c)\\
&\mathit{up}\ (t, R\ l\ c) = (\mathit{Fork}\ l\ t, c)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{modify} :: \mathit{Zipper} \to (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Zipper}\\
&\mathit{modify}\ (t, c)\ f = (f\ t, c)
\end{aligned}
$$

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 the tree has.

As a concrete example of this, in Fig. 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* (the one on the leaf labelled with 1) every attribute is computed only once, in the second case (the computation of *replace* on the leaf labelled with 2) we see that some calls to *globmin* are new, but then we reach a

[4] So t will be treated as a *normal* tree.

```
  – Inherited
globmin :: (Memo Globmin m Int, Memo Locmin m Int) ⇒ AGTree_m 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 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 m Tree
replace = memo Replace $ λz → case constructor_m z of
                                C_Root   → replace .@. 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.

point in which we start to repeat the steps that have already been taken (the underlined attributes in the figure), 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 proper attribute grammar embedding.

Although we use *repmin* as a running example, the strategy we analyze has been tested and assessed in other well-known problems in the AG domain, some of which are presented in Section 3.4.

3. Memoized AGs

Fig. 3 shows an alternative solution to the one previously given in Fig. 1. 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.

```
data Tree_m m = Fork_m m (Tree_m m) (Tree_m m)
              | Leaf_m m Int
```

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

```
build_m :: Tree → m → Tree_m m
build_m (Leaf n)    mt = Leaf_m mt n
build_m (Fork l tr) mt = Fork_m mt (build_m l mt) (build_m r mt)
```

The evaluation of an AG using memoization requires the inspection and update of the memo tables attached to the nodes of a $Tree_m$, in particular, the table attached to the root node:

```
getMemoTable :: Tree_m m → m
getMemoTable (Leaf_m mt _)   = mt
getMemoTable (Fork_m mt _ _) = mt
updMemoTable :: (m → m) → Tree_m m → Tree_m m
updMemoTable f (Leaf_m mt i)   = Leaf_m (f mt) i
updMemoTable f (Fork_m mt l r) = Fork_m (f mt) l r
```

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

$$\begin{aligned}
&\textbf{type } Zipper_m\ m = (Tree_m\ m, Cxt_m\ m)\\
&\textbf{data } Cxt_m\ m = Root_m\\
&\qquad\qquad\ \mid Top_m\\
&\qquad\qquad\ \mid L_m\ m\ (Cxt_m\ m)\ \ (Tree_m\ m)\\
&\qquad\qquad\ \mid R_m\ m\ (Tree_m\ m)\ (Cxt_m\ m)
\end{aligned}$$

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:

$$\begin{aligned}
&up_m :: Zipper_m\ m \to Zipper_m\ m\\
&up_m\ (t, Top_m) \quad\ = (t, Root_m)\\
&up_m\ (t, L_m\ m\ c\ r) = (Fork_m\ m\ t\ r, c)\\
&up_m\ (t, R_m\ m\ l\ c) = (Fork_m\ m\ l\ t, c)
\end{aligned}$$

3.1. Memo tables

A memo table contains *Maybe* elements corresponding to the attributes, 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:

$$\begin{aligned}
&\textbf{data } Globmin = Globmin\\
&\textbf{data } Locmin\ \ = Locmin\\
&\textbf{data } Replace\ = Replace
\end{aligned}$$

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

$$\begin{aligned}
&\textbf{class } Memo\ att\ m\ a\ \textbf{where}\\
&\quad mlookup :: att \to m \to Maybe\ a\\
&\quad massign\ :: att \to a \to m \to m
\end{aligned}$$

The intended meaning of $massign\ att\ v\ m$ is to assign the value v to the attribute att stored in the table m. 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 possible representation is in terms of tuples. In our example, the tuple stores values corresponding to *globmin* (*Int*), *locmin* (*Int*) and *replace* (*Tree*).

$$\textbf{type } MemoTable = (Maybe\ Int, Maybe\ Int, Maybe\ Tree)$$

The use of tuples to represent memo tables imposes an important drawback: 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 [22] defined in the HList[5] library. In our repository we include an alternative version of our embedding 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:

$$\begin{aligned}
&\textbf{instance } Memo\ Globmin\ MemoTable\ Int\ \textbf{where}\\
&\quad mlookup\ _\ \ (g, _, _) = g\\
&\quad massign\ _\ v\ (g, l, r)\ \ = (Just\ v, l, r)
\end{aligned}$$

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.

[5] https://hackage.haskell.org/package/HList.

```
  – Inherited
globmin :: (Memo Globmin m Int, Memo Locmin m Int) ⇒ AGTree_m m Int
globmin = memo Globmin $ do c ← constructorM_m
                            case c of
                              C_Root  → locmin treeM_m
                              C_Leaf _ → atParentM globmin
                              C_Fork  → atParentM globmin
  – Synthesized
locmin :: (Memo Locmin m Int) ⇒ AGTree_m m Int
locmin = memo Locmin $ do  c ← constructorM_m
                           case c of
                             C_Leaf v → return v
                             C_Fork  → min <$> locmin leftM_m
                                       <*> locmin rightM_m
replace :: (Memo Replace m Tree, Memo Globmin m Int, Memo Locmin m Int) ⇒ AGTree_m m Tree
replace = memo Replace $ do c ← constructorM_m
                            case c of
                              C_Root  → replace treeM_m
                              C_Leaf _ → Leaf <$> atParentM globmin
                              C_Fork  → Fork <$> replace leftM_m
                                        <*> replace rightM_m
```

Fig. 4. Repmin defined using the monadic and applicative interface.

In order 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:

$$\textbf{data}\ \mathit{MemoTable} = \mathit{MTFork}\ \mathit{MemoTableFork} \mid \mathit{MTLeaf}\ \mathit{MemoTableLeaf}$$

It is then necessary to define *build* and the corresponding instances of the *Memo* class taking into account such 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:

$$\textbf{type}\ \mathit{AGTree}_m\ m\ a = \mathit{Zipper}_m\ m \to (a, \mathit{Zipper}_m\ m)$$

Notice that this could also be represented in terms of a *State* monad, but we will not take that alternative in this paper. Fig. 4 shows the running example implemented using a monadic and applicative interface, where each Zipper operation was replaced by its monadic version (e.g. $tree_m$ by $treeM_m$).

The function *memo*, used in every attribute definition of Fig. 3, 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@(t, _) =
  case mlookup attr (getMemoTable t) of
    Just v    → (v, z)
    Nothing → let (v, z') = eval z
              in (v, modify_m z' (updMemoTable (massign attr v)))
```

First of all, the memo table is obtained (by *getMemoTable*) from the focus tree. 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 continuous movement of the computation focus. This means that the location where the computation of an attribute is taking place is continuously 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 attribute because in some cases it is necessary to return to the original location after moving.

To see an example suppose we implement *locmin* of Fig. 3 in the following way:

$$
\begin{aligned}
locmin = memo\ Locmin\ \$\ \lambda z \rightarrow\ & \textbf{case}\ constructor_m\ z\ \textbf{of} \\
& C_{Leaf}\ v \rightarrow (v, z) \\
& C_{Fork} \rightarrow\ \textbf{let}\ (left, z') = locmin\ (left_m\ z) \\
& \qquad\qquad\ (right, z'') = locmin\ (right_m\ z') \\
& \qquad\ \textbf{in}\ (min\ left\ right, z'')
\end{aligned}
$$

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 *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 n the given child, and then the focus goes back to the parent using up_m:

$$
\begin{aligned}
& (.@.) :: AGTree_m\ m\ a \rightarrow AGTree_m\ m\ a \\
& eval\ .@.\ z = \textbf{let}\ (v, z') = eval\ z \\
& \qquad\qquad\ \textbf{in}\ \ (v, up_m\ z')
\end{aligned}
$$

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.

$$
\begin{aligned}
& atParent\ eval\ z = (v, (back\ z)\ z') \\
& \quad \textbf{where} \\
& \qquad (v, z') = eval\ (up_m\ z) \\
& \qquad back\ (\_, Top_m) \quad\ = tree_m \\
& \qquad back\ (\_, L_m\ \_\ \_\ \_) = left_m \\
& \qquad back\ (\_, R_m\ \_\ \_\ \_) = right_m
\end{aligned}
$$

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

$$
\begin{aligned}
& repmin :: Tree \rightarrow Tree \\
& repmin\ t = t' \\
& \quad \textbf{where} \\
& \qquad (t', \_) = replace\ z \\
& \qquad z \quad\ = mkAG_m\ (build_m\ t\ emptyMemo)
\end{aligned}
$$

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

$$
emptyMemo = (Nothing, Nothing, Nothing)
$$

3.3. Selective memoization

In the previous sections, we have detailed the main ingredients of our memoized zipper-based AG embedding. The ambition to avoid the recomputation of attribute values, however, is not new: it was first proposed by Reps and Teitelbaum in the context of incremental computation [31,32]. This seminal work did have a great impact in the AG community: several AG systems were developed to produce evaluators that compute each attribute value a single time [32,24,12]. Similar to our approach, this is achieved by memoizing attribute values, either in the tree's nodes [32,12], or in a global memo table [24].

In general, however, these systems use a predefined attribute memoization strategy (usually, they memoize all attribute values) which can not be modified by the AG writer. The exception to this is [7], where it is possible to define strategies to avoid the overhead of caching for cases where it would actually not pay off [39]. And indeed, as also shown in [37, 7], the performance of attribute evaluators can be further improved by using a selective memoization strategy, instead of memoizing all attributes! In our embedding, since both the tree and attribute computations are parametric on the type of the memo table, changing the memoization strategy only requires changing the memo table! Indeed, it does not require changing the specification of the attribute grammars, which is aligned in spirit with [39] not to couple the memoization strategy with the attribute grammar specification.

A version of *repmin* that only memoizes *globmin* can be obtained by defining a new type for the memo table:

```
type SelMemoTable = Maybe Int
```

with its corresponding instances of the *Memo* class:

```
instance Memo Globmin SelMemoTable Int where
  mlookup _  g = g
  massign _ v g = Just v
instance Memo Locmin SelMemoTable Int where
  mlookup _  _ = Nothing
  massign _ _ g = g
instance Memo Replace SelMemoTable Tree where
  mlookup _  _ = Nothing
  massign _ _ g = g
```

and the semantics function:

```
repminSel :: Tree → Tree
repminSel t = t'
  where
    (t', _) = replace z
    z       = mkAG_m (build_m t emptyMemoSel)
    emptyMemoSel :: SelMemoTable
    emptyMemoSel = Nothing
```

Thus, our embedding provides a flexible framework for experimenting with different memoization strategies for the same AG.

3.4. Results

In this section, we assess in terms of efficiency the memoization approach we presented in this paper against the original, non-optimized embedding of [26]. For this assessment, we test the optimized and non-optimized versions of *repmin* together with two other well known AG examples from the literature. The results are presented as running times and memory consumption.

For the benchmarks we are presenting, we compiled the different programs 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).

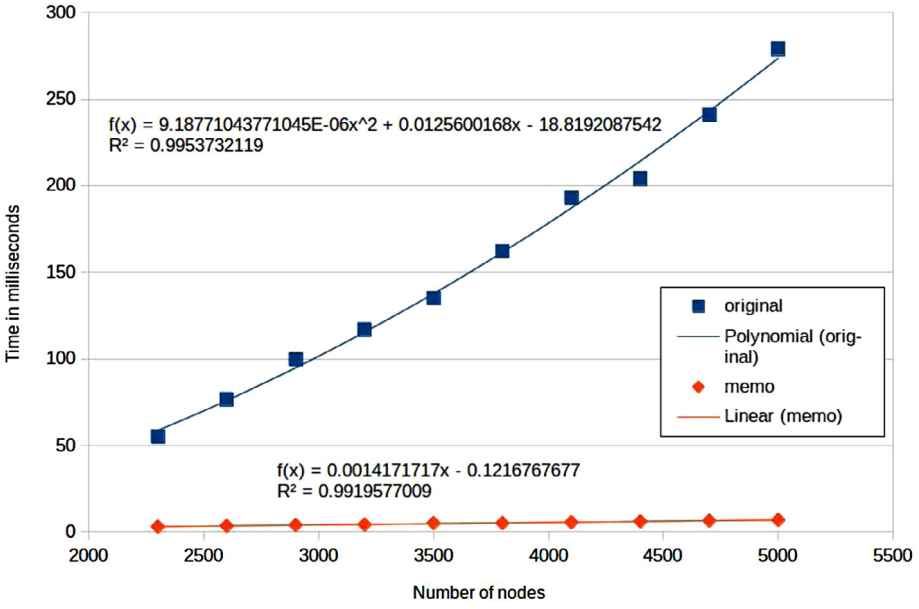
3.4.1. Repmin

We started by benchmarking the running example of the paper: *repmin*. For this test, we used increasingly larger balanced binary leaf trees with a number of nodes ranging from 2300 to 5000, represented on Fig. 5a in the *x*-axis.

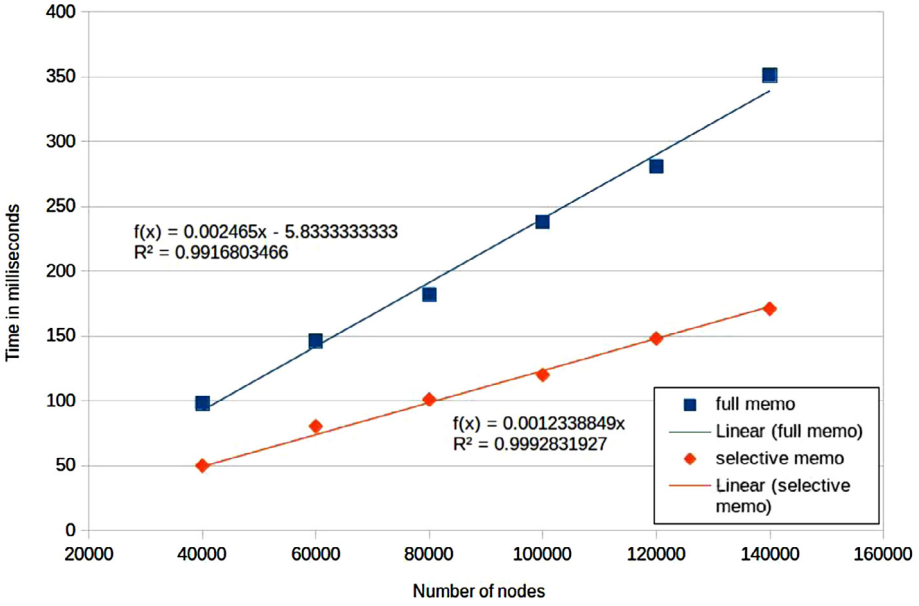
The performance results of the implementations with and without memoization allow us to observe that the memoized version significantly improves the performance of the original version. Indeed, when we reach the 5000 nodes there is a clear gap in the time required to run *repmin* between the original and the memoized versions. As we grow from 2300 to 5000 nodes, the memoized version shows only a slight increase in running time, while the original approach takes proportionally more and more processing time. By applying curve fitting to the results, we notice that the non-memoized version behaves as a quadratic function (R^2 of 0.995), while with memoization we fit a linear function (R^2 of 0.992) with a very small factor.

Another interesting result is how well the memoized version scales. In Fig. 5b we show the performance results of the memoized version for a larger number of inputs (ranging from 40000 to 140000). In this case, we compare full memoization with the selective memoization introduced in Section 3.3. In both cases linear curves are obtained, although with selective memoization we get a smaller coefficient. Notice, from Figs. 5a and 5b, that the version that uses selective memoization takes, for an input of size 140000, the same time the non memoized version takes to process an input of 8000.

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 Fig. 6. The tests were performed with a balanced tree with 150,000 nodes. As



(a) Non memoized vs Full memoization.

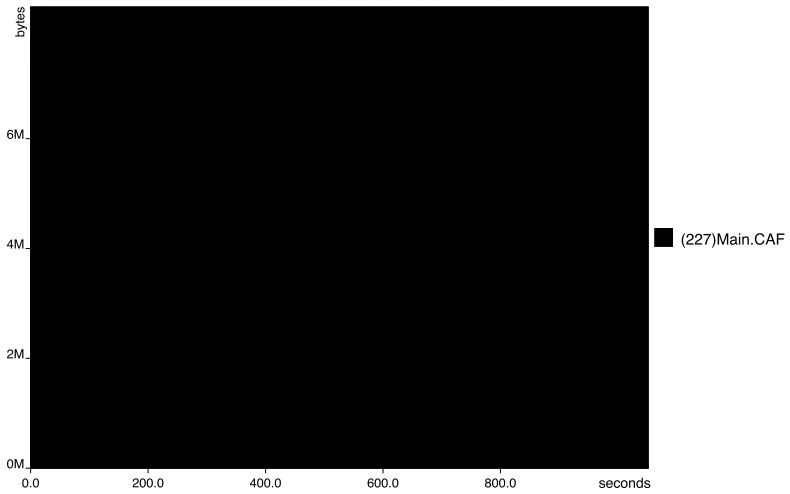


(b) Full memoization vs Selective memoization.

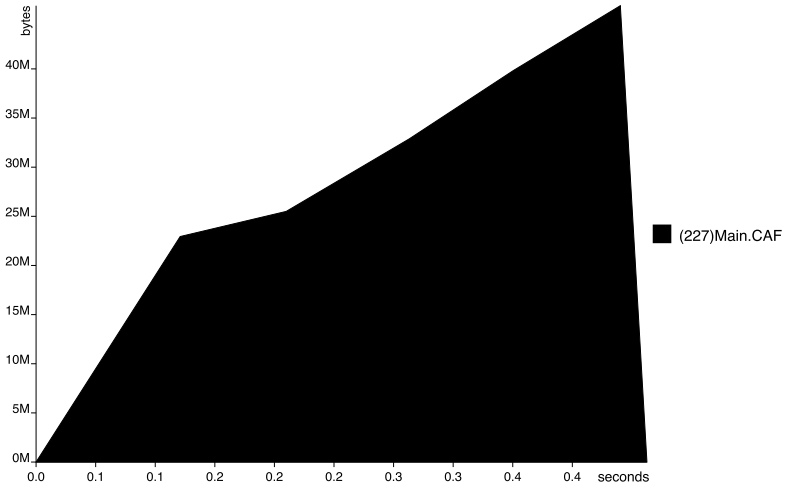
Fig. 5. Performance of the *repmin* implementations.

expected, we observe that the original version (Fig. 6a) has the lowest peak of memory consumption, of slightly more than 8 Mbytes. The full memoized version (Fig. 6b) reaches the highest peak of consumed memory, of around 45 Mbytes. By reducing the size of the memo tables, the selective memoization version (Fig. 6c) enables an important memory saving, with peaks of around 16 MBytes.

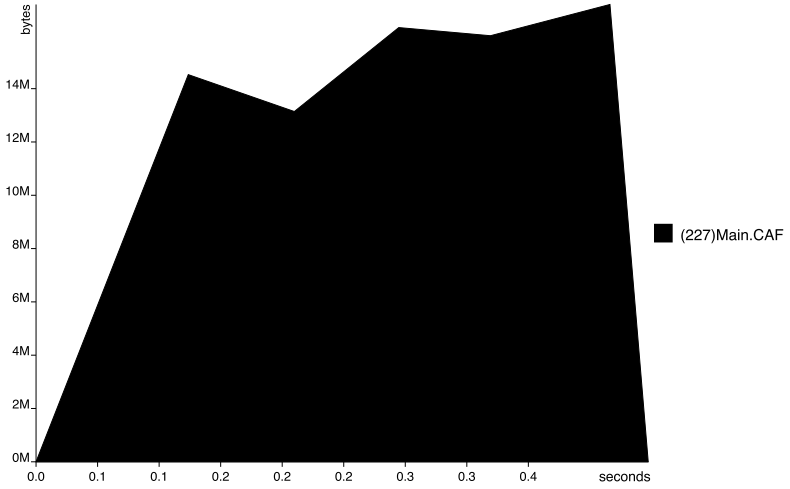
The time spent in garbage collection (GC) is clearly different between memoized and non-memoized versions. The non-memoized version spends 8.4% of the running time in GC, while the full and selective memoized versions spend 56.4% and 51.1%, respectively. As mentioned earlier, the original implementation of *repmin* is an extremely heavy example of semantics requiring a large number of 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, even though they spend half of the time in GC. The next examples are focused on real situations.



(a) Non memoized.



(b) Full Memoization.



(c) Selective Memoization.

Fig. 6. Heap Profile on Repmin (values in Mbytes).

3.4.2. Algol-68 scope rules

In this section we benchmark an implementation of the Algol 68 scope rules [33,14,26]. 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 [21] (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 containing either use or declaration of names, or a nested block. An example of a program is:

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

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. Moreover, to make it easier to detect which variables are being incorrectly used, we require that the list of invalid variables follows the sequential structure of the input program. Thus, the semantic meaning of processing p is $[w, x]$.

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. Moreover, a straightforward solution would build an environment in a first traversal (and detecting duplicated declarations), so that in a second traversal it could detect invalid uses (use of variables in the computed environment). As a consequence, this algorithm forces the programmer to explicitly pass errors detected in the first traversal to the second one, so that they are combined according to sequential structure of the input.

In Fig. 7a, we show the results we obtained when running the different implementations on Algol programs with an increasing number of enclosing blocks. The x-axis ranges from 60 to 150 enclosing blocks. Similarly to the previous example, memoization shows better processing times. The behaviour of the original version fits to a quadratic curve, while the memoized version is linear, with a very slight increase. With a larger number of enclosed blocks (Fig. 7b), the curve of the memoized solution becomes quadratic but still with a much smaller coefficient than the non-memoized version. This shows that the memoized version scales. For example, for an input of 1500 enclosed blocks, the memoized version takes the same time as the non-memoized version for an input of around 140.

Fig. 8 compares the memory consumption of the two versions with an input of 1500 enclosed blocks. Like in the *repmin* example, the memoized version consumes much more memory than the other (peaks of 1 Mbyte against 10 Mbytes). In this case, there is a minor difference in garbage collection times. The non-memoized version spends 7.3% of the running time in GC, while the memoized version spends 20.5%.

3.4.3. HTML table formatter

We now analyze an example from [40]: 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. Fig. 9 shows an example of a possible input (left) and correspondent output (right).

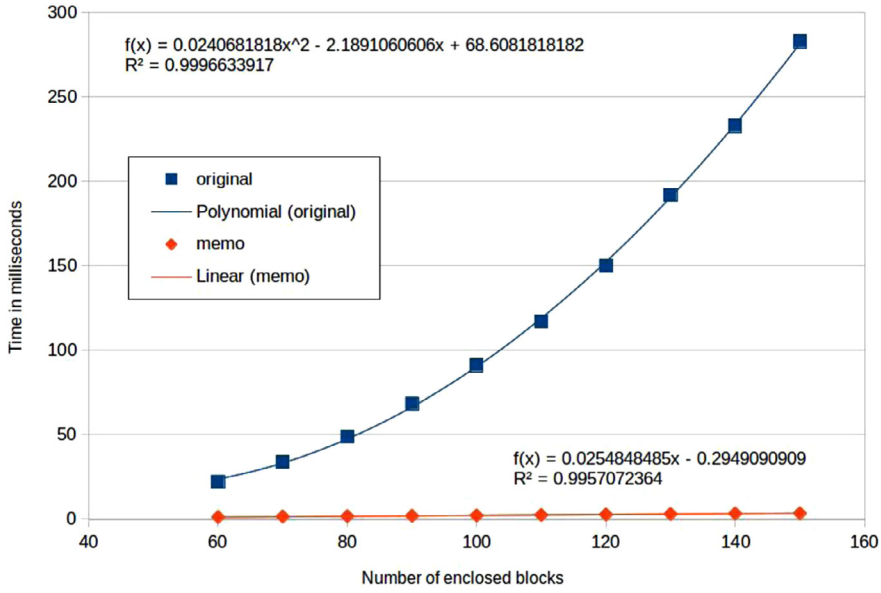
Notice that in the output, all the lines have the same number of columns and the columns have the same length. None of these features are required in the HTML language.

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 the definition of gluing data types to pass the width/height (blue subscripts/superscripts in Fig. 9) 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.

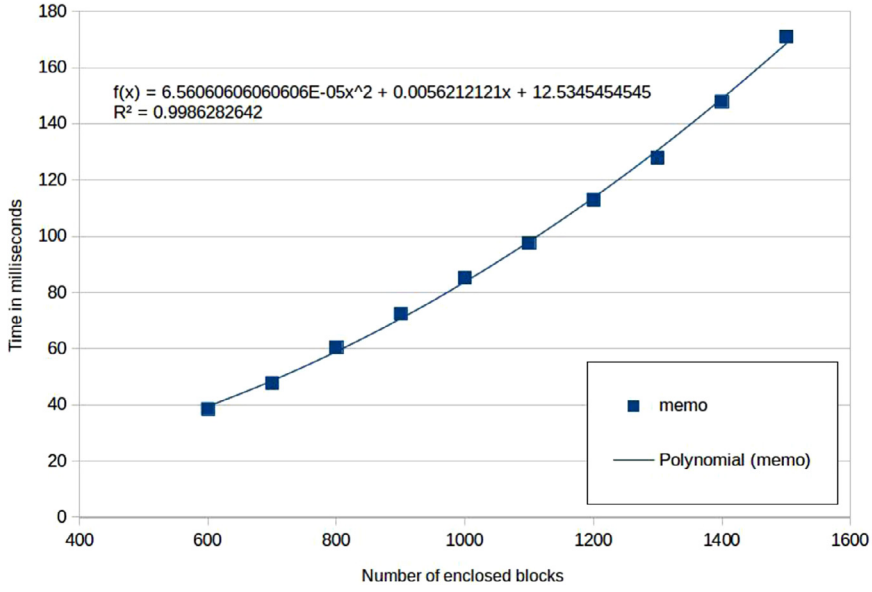
To test this AG, we computed trees representing HTML tables with the same number of rows (50) and an increasing number of columns. All the cells of the tables include the same text, excepting for the ones in the last column, which include nested tables with, recursively, the same shape but half the number of rows and columns of the containing table. The results are presented in Figs. 10 (time) and 11 (memory consumption), where the x-axis represents the number of columns, ranging from 10 to 100.

It can be observed from these results that there is a considerable reduction in execution time, going from a cubic curve to a linear one. Although the regression curve in the non-memoized case does not fit accurately the runtime data (R^2 of 0.957), the difference of slopes in the curves can be seen clearly; for example with linear regression we get the function $136.23x - 526.67$ with a R^2 of 0.922.

The memoized evaluator does consume much more memory; Fig. 11 shows a peak of around 900 Mbytes in the memoized version and 4 MBytes in the non-memoized one, for a table of 50 rows and 50 columns. This leads to more time spent in garbage collection; while the non-memoized version spends 12.6% of the time in GC, when applying memoizaton the GC time grows to 60.1%. Note that, for large inputs this evaluator produces large strings. As a result, in the memoized version,



(a) Non memoized vs Memoized.



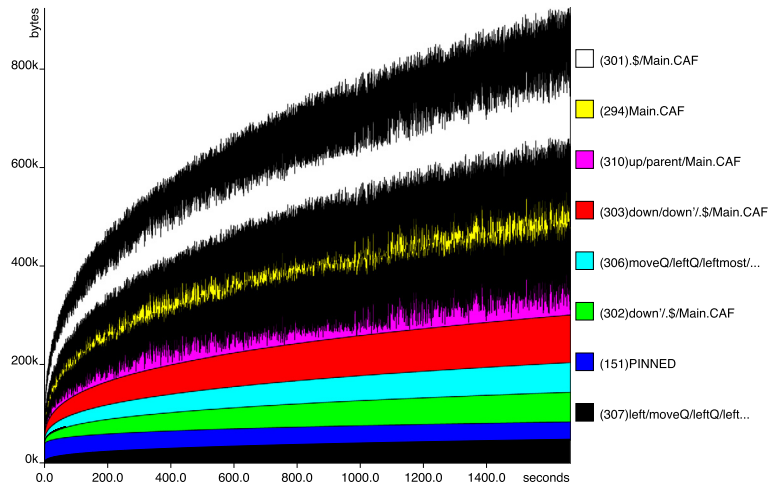
(b) Memoized.

Fig. 7. Performance of the Algol implementations.

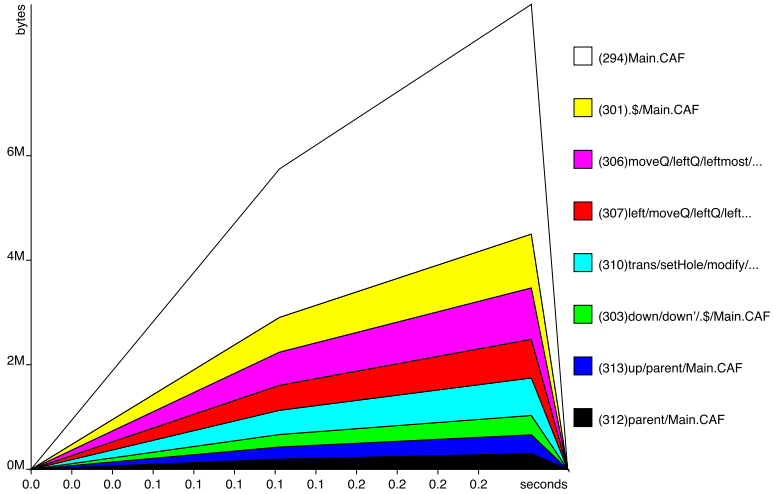
partial results are kept in the memo table: large strings in this case. As usual in memoization techniques the gain in runtime is obtained by using additional memory: the memo table. In fact, memory consumption can result in a scalability problem in certain cases. To reduce this problem several AG techniques to purge entries from a memo table have been proposed [33], which can be used in our zipper-based setting. Moreover, we may use selective memoization where some attributes are not memoized at all.

4. Memoized higher-order AGs

Higher-Order Attribute Grammars (HOAGs) [45,41] represent an important extension to traditional AGs. This extension allows the definition of higher-order attributes, which are attributes whose value is itself not only a (higher-order) tree, but a tree where (possibly higher-order) attributes can be defined.



(a) Non memoized.



(b) Memoized.

Fig. 8. Heap Profile on Algol (values in Mbytes). (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

```
⟨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. 9. HTML Table Formatting.

HOAGs have two main characteristics.

Firstly, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. Consequently, the AG specification does not need to rely in a (functional) notation, which is external to the AG formalism, to express computations. A typical application of higher-order attributes is to model the (recursive) lookup functions. For example, in the Algol 68 AG discussed in the previous section, the functions to lookup the environment to detect duplicated definitions and invalid uses can be defined via higher-order attributes as shown in [33].

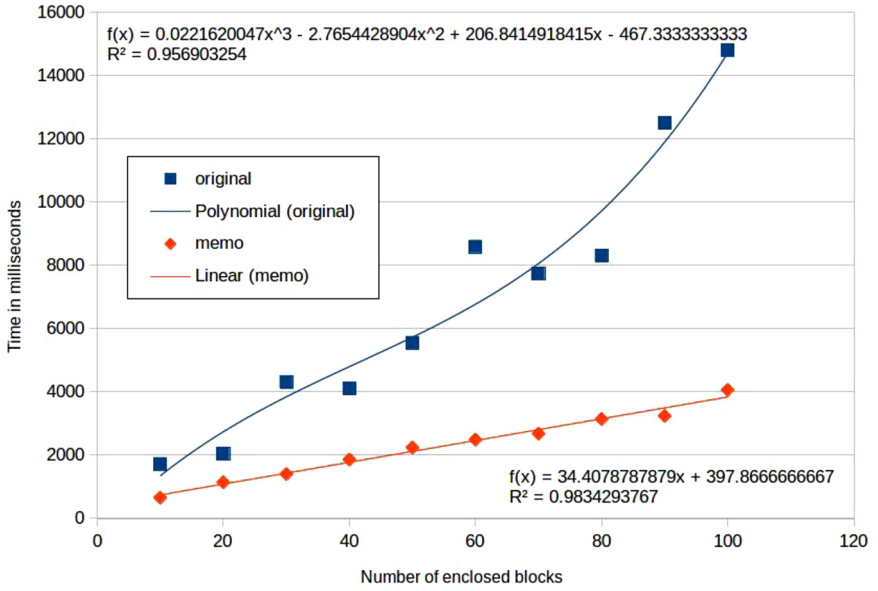


Fig. 10. Time performance of the HTML table formatters.

Consequently, such recursive computations are defined with the AG formalism and can benefit directly from well-kown AG optimization techniques, like for example attribute memoization. *Strong attribute grammars* [36] advocates this approach where the full static semantics of a language is specified via (higher-order) attributes.

Secondly, when an algorithm cannot be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed first.

Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language. Consequently, the decoration of the abstract tree is performed according to the attribution rules defined for the abstract grammar. That is, the semantic rules of the language are checked during the decoration of the abstract syntax tree.

These characteristics make HOAGs particularly suitable to define the static semantics of programming languages in a component-based programming style [34]: attribute grammar components are efficiently and easily "plugged-into" an AG specification via higher-order attributes. In this approach, one AG component defines a higher-order attribute which is decorated according to the attribute equations defined by another AG component.

HOAGs are supported by our zipper-based embedding as shown in [26,27]. Next, we show a simple *Let-In* language whose scope rules exactly follow the Algol 68 AG. As a consequence, the Algol 68 AG is "plugged-into" the Let-In specification via a higher-order attribute. Moreover, we also show how to express this HOAG using our memoized zipper-based embedding. Finally, we show the impact of memoization in the HOAG setting.

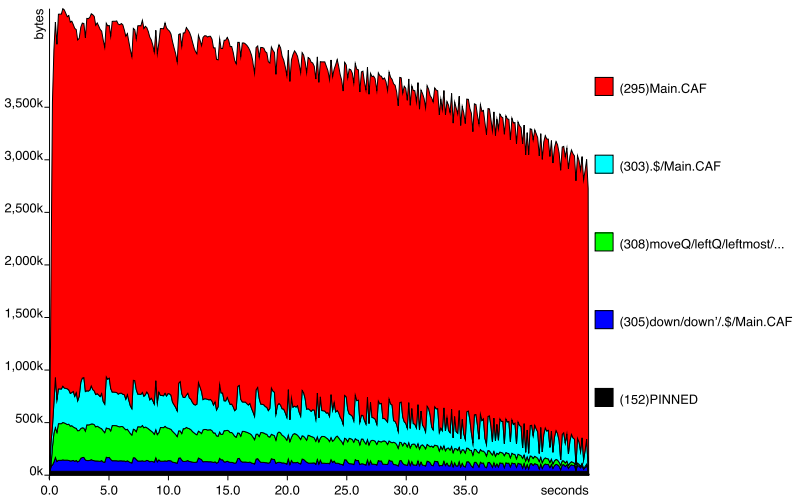
4.1. Let-In Haskell expressions

Consider the following simple example of a Haskell **let-in** construct:

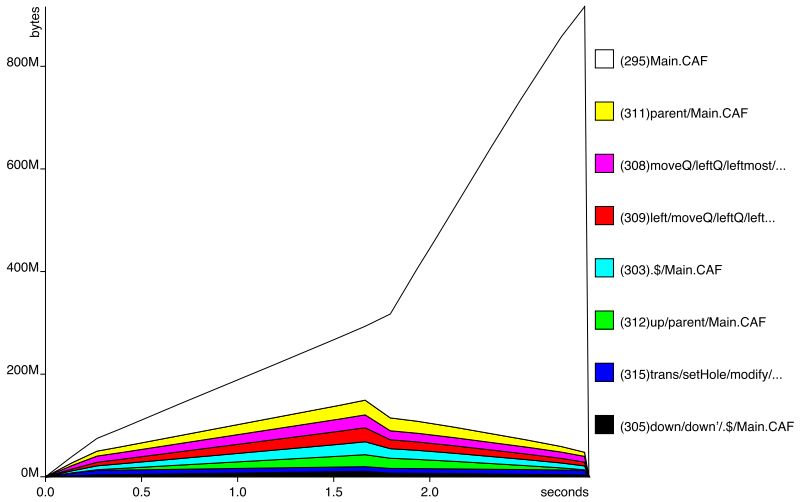
$$\begin{aligned}&\textbf{let } b = a + 3\\ &\quad\ \ a = 2\\ &\textbf{in } a + b\end{aligned}$$

It defines a valid expression that evaluates to 7. As shown by this example, in Haskell we may use a variable before it is defined (use of a when defining b). Moreover, Haskell allows nested lets as shown next:

$$\begin{aligned}&\textbf{let } b = a + 3\\ &\quad\ \ a = \textbf{let } b = 4\\ &\qquad\quad \textbf{in } \ b + 2\\ &\textbf{in } a + b\end{aligned}$$



(a) Non memoized.



(b) Memoized.

Fig. 11. Heap Profile on HTML (values in Mbytes).

where the definition of b in the local scope hides the definition in the global one. Obviously, Haskell does not allow duplicated declarations in the same scope, nor the use of non-defined variables. According to Haskell scope rules, the next program contains three errors: the use of non-declared variables d and e and the duplicated declaration of a.

$$
\begin{array}{l}
\textbf{let } a = 2 \\
\quad c = \textbf{let } a = 4 \\
\qquad\quad \textbf{in } \ a - b + d \\
\quad b = a + 3 \\
\quad a = c * 4 \\
\textbf{in } e - a
\end{array}
$$

In order to develop a processor for **let-in** expressions we would like to compute the list of invalid variables and, whenever no errors occur, the value of the let expression as well. Like in the Algol 68 example, to make it easier to detect which variables are being incorrectly used, we require that the list of errors follows the sequential structure of the input. Thus, for the expression above the result is $[d, a, e]$.

A conventional implementation of the required analysis naturally leads to two traversals of each **let-in** block: once for processing the declarations and constructing an environment and a second time to process the uses (using the computed environment) in order to check for the use of non-declared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared

(at the same lexical level). In the affirmative case, the identifier is added to the resulting list of errors. In summary, the algorithm for the processing **let-in** expressions looks as follows:

1st Traversal	2nd Traversal
– *Collect the list of local definitions*	– *Use the list of definitions as the global environment*
– *Detect duplicate definitions*	– *Detect use of non defined names*
(using the collected definitions)	– *Combine "both" errors*

As the reader may have already noticed, this algorithm expresses exactly the scope rules of the Algol 68 AG! So, the question that arises is: can we reuse the Algol 68 AG when developing the processing of **let-in** expressions? In a HOAG the answer is yes. In fact, instead of expressing this complicated algorithm in the **let-in** abstract syntax tree, we can compute first a simpler abstract tree representation (the Algol AST) and then reuse the Algol 68 AG evaluator to synthesize from this new tree the correct list of errors. Thus, the synthesized abstract tree is a higher-order attribute (its value is a tree) of the Let-In AG; we associate attributes with such tree according to the rules of the Algol 68 AG.

Before we discuss how to specify this language analysis in the HOAG setting, it should be noticed that, the decl/use of variables of the last shown **let-in** expression can be represented by the following Algol 68 program *p*.

```
p = [ decl a;
      decl c;
        [ decl a;
          use a; use b; use d ]
      decl b; use a;
      decl a; use c;
      use e; use a ]
```

To help understanding the Let-In AG fragment that synthesizes the Algol 68 tree, we show the abstract syntax of both the Algol 68 and Let-In AGs.

```
data Algol68 = RootAlg Its
data Its     = ConsIts It Its
             | NilIts
data It      = Decl    String
             | Use     String
             | Block   Its
```

```
data LetIn = RootLet    Let
data Let   = Let        List In
data In    = In         Exp
data List  = ConsLet    String Let List
           | ConsAssign String Exp List
           | EmptyList
data Exp   = Plus       Exp Exp
           | Variable   String
           | Constant   Int
```

To simplify the presentation in this paper we consider expressions with a single operator: addition.

To develop the Let-In AG in a component-based, AG programming-style we rely on HOAGs as follows [34]: first, associated to a production applied to the root symbol of the Let-In AG, we declare a higher-order attribute named $algol_{68}$ of type *Algol68*. *Algol68* is the root nonterminal of the Algol 68 AG, and $algol_{68}$ is an attributable attribute (*ata*) [45,33], meaning that it is an attribute that can be decorated with attributes again.

Then, we instantiate this attributable attribute with its inherited attributes: since Algol 68 AG is context independent, its root symbol does not have inherited attributes. So, no inherited attributes are instantiated in this case. Next, we access the synthesized attributes of the Algol 68 AG: the list of invalid declaration/uses of variables. Below, we show this fragment of the Let-In HOAG, written in the notation used in the LRC system [24]:

```
LetIn { syn [String] errors;}          -- errors: syn attr of type [String]
LetIn -> RootLet
   ata algol68 : Algol68'              -- Declaration of the ata of type Algol68
   algol68 = RootAlg  (Let.algol68);   -- Instantiation of the ata
   LetIn.errors = algol68.errors;      -- Use of synthesized attrs (of the ata)
```

The $Let.algol_{68}$ is a regular synthesized attribute of type *Its* of the Let-In AG which computes the *Algol68* tree: it uses the constructors of the Algol 68 AG to do so. In the instantiation of the *ata* we have used the production/constructor *RootAlg* to produce the Algol tree with the desired type. We omit here the fragment of the AG where such rules are defined in an usual notation. In the next section, we show those rules expressed in a similar notation: our memoized zipper-based HOAG embedding.

4.2. Memoized zipper-based HOAG

As we have shown in Section 3, we can elegantly define attribute grammar rules in our zipper-based embedding, where we memoize attribute values in order to avoid attribute recomputations. Using this approach we can easily express the rules of the Let-In AG that synthesize the *Algol68* tree. Like we explained before, these rules use the constructors of the Algol 68 AG to build such tree.

We write the HOAG in a monadic style and using a generic version of our zipper-based library. The combinators $Child_i$ correspond to the combination of combinator (.@.) with $left_m/right_m$: they move the focus of the zipper to the *ith* child to compute an attribute there, returning the focus to the original location afterwards. Function $lexeme_{String} :: Algol68_m \rightarrow String$ takes a zipper and returns a "terminal" symbol of type *String*.

```
algol68 :: Dir → Algol68_m Its
algol68 = memo Attr_algol68$do
  constr ← constructor_m
  case constr of
    RootLet_m      → algol68 Child_1
    Let_m          → do t_1 ← algol68 Child_1
                        t_2 ← algol68 Child_2
                        return (concatIts t_1 t_2)
    In_m           → algol68 Child_1
    ConsAssign_m   → do ag ← get                    – decl var
                        var ← lexeme_String ag
                        t_1 ← algol68 Child_2
                        t_2 ← algol68 Child_3
                        return (concatIts (ConsIts (Decl var) t_1) t_2)
    ConsLet_m      → do ag ← get                    – decl var, nested block
                        var ← lexeme_String ag
                        t_1 ← algol68 Child_2
                        t_2 ← algol68 Child_3
                        return (ConsIts (Decl var) (ConsIts (Block t_1) t_2))
    EmptyList_m    → return NilIts
    Plus_m         → do t_1 ← algol68 Child_1
                        t_2 ← algol68 Child_2
                        return (concatIts t_1 t_2)
    Variable_m     → do ag ← get                    – use var
                        var ← lexeme_String ag
                        return (ConsIts (Use var) NilIts)
    Constant_m     → return NilIts
```

This zipper-based AG function synthesizes attribute $algol_{68}$ of type *Its*, as expressed by its type, while avoiding attribute recomputation. Now, we have to use an attributable attribute of type *Algol68* which is decorated using memoization by the evaluator of the Algol 68 AG (whose results using memoization were analysed in Section 3.4.2). Such HOAG fragment is expressed in our setting very much like the usual notation shown before, as we show next.

```
errors_m :: Dir → Algol68_m [String]
errors_m = memo Attr_errsAlgol$do
  constr ← constructor_m
  case constr of
    RootLet_m → do   – Instantiation of the ata
                  algolIts ← algol68 Local
                  let ata_algol68 = RootAlg algolIts
                    – Algol semantics decorates the higher order tree
                    – Returns the synthesised attr. of the ata
                  return (Algol68.evaluator ata_algol68)
```

To show the complete Let-In embedded and memoized HOAG we include the fragment to calculate the value of a let expression (if no errors occur):

```
calculate_m :: Dir → Algol68_m Int
calculate_m = memo Attr_calculate$do
  constr ← constructor_m
  case constr of
    RootLet_m  → do errs ← errors_m Local
                    if (length errs ≠ 0)
                    then return 0
                    else calculate_m Child_1
    Let_m      → calculate_m Child_2
    In_m       → calculate_m Child_1
    Plus_m     → do n1 ← calculate_m Child_1
                    n2 ← calculate_m Child_2
                    return (n1 + n2)
    Variable_m → do ag ← get
                    getVarValue_m (lexeme_String ag) Local
    Constant_m → do ag ← get
                    return (lexeme_Int ag)
```

Function $getVarValue_m$ returns the value of a giving variable name whereas $lexeme_{Int} :: Algol68_m \rightarrow Int$ takes a zipper and returns a terminal symbol of type *Int*.

4.3. Performance of zipper-based embedded HOAGs

To assess the efficiency of the memoization approach in the context of HOAGs we consider three different versions of the Let-In zipper-based AG:

- $LetIn_{FO}$: a first-order attribute grammar where the name analysis and the calculation of the expression is expressed in the *LetIn* tree. No memoization is used.
- $LetIn_{HO}$: a HOAG, where the name analysis is expressed by the Algol 68 AG. No memoization is used.
- $LetIn_{MemoHO}$: a memoized HOAG, where the name analysis is expressed by the memoized Algol 68 AG, and the calculation of the expression is also memoized. In fact, this is the HOAG presented in the previous section.

To benchmark these three evaluators we consider two different **let-in** programs. The first program does not contain nested lets and consists of increasingly larger let-expressions: ranging from 200 to 1000 assignments (represented in Fig. 12 in x-axis). The second program contains increasingly larger and deeper nested lets: ranging from 6 to 17 nested levels (represented in Fig. 13 in x-axis). The execution time is represented in y-axis (in miliseconds).

Comparing the different higher-order versions, the performance results clearly show that the memoized version significantly improves the performance of the non-memoized implementations. In Fig. 12 both higher-order versions fit a quadratic curve, but the non-memoized version has a coefficient of 0.003 while the memoized has a coefficient of 0.0006. Fig. 13 shows an ever more important difference, since we go from a cubic to a quadratic curve.

The results of our benchmarks also show another interesting result: the non-memoized higher-order version of the Let-In AG is also more efficient than the first order AG! In fact, by expressing an algorithm in a more suitable tree, the attribute evaluation is done in a more efficient data structure (smaller, in this case). Although some attributes may need to be recomputed because of the non-memoized version of the AG, the fact of evaluating on smaller trees implies fewer attributes to be computed and fewer traversals to be performed.

Currently in the literature there is no study that compares the performance of first-order AGs with equivalent HOAGs. We think this is an interesting study to be performed, but we are also convinced that conventional techniques for HOAGs are not substantially affected by the tree structure, as our non-memoized zipper-based definitions are.

Figs. 14a, 14b and 14c show the memory profiler of the first-order, HO and full memoized HO Let-In AG, respectively. As expected the memoized version consumes more memory than the others. While the maximum residency in the first-order version (14a) is 2400 Kbytes, and in the higher-order case (14b) is 3200 Kbytes, the full memoized HO version (14c) reaches peaks of more than 16 Mbytes.

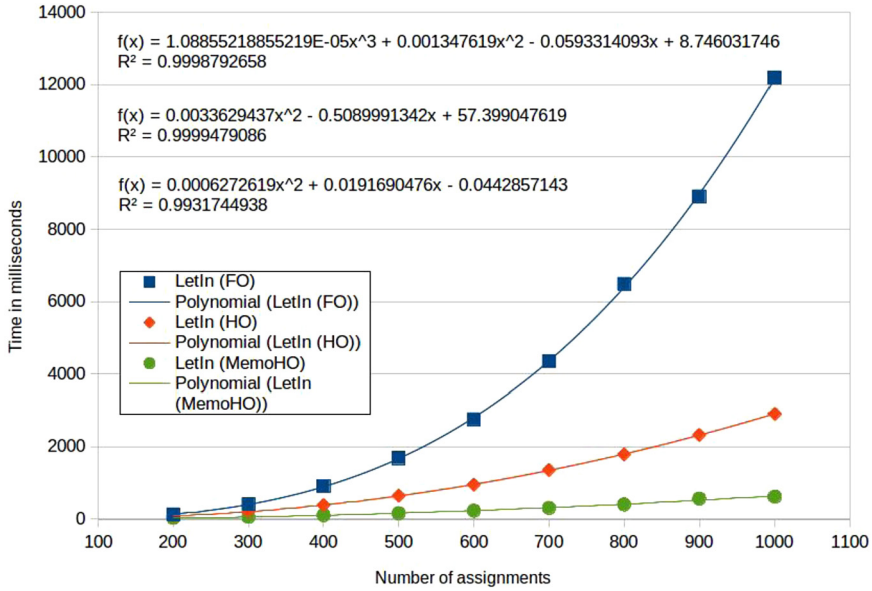


Fig. 12. Performance of the Let-In embedded attribute grammars **without** nested lets.

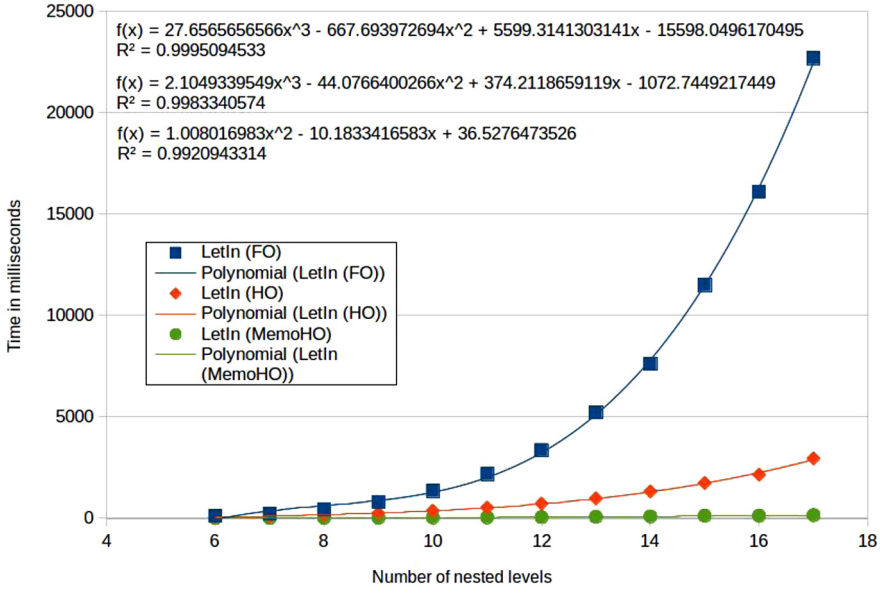


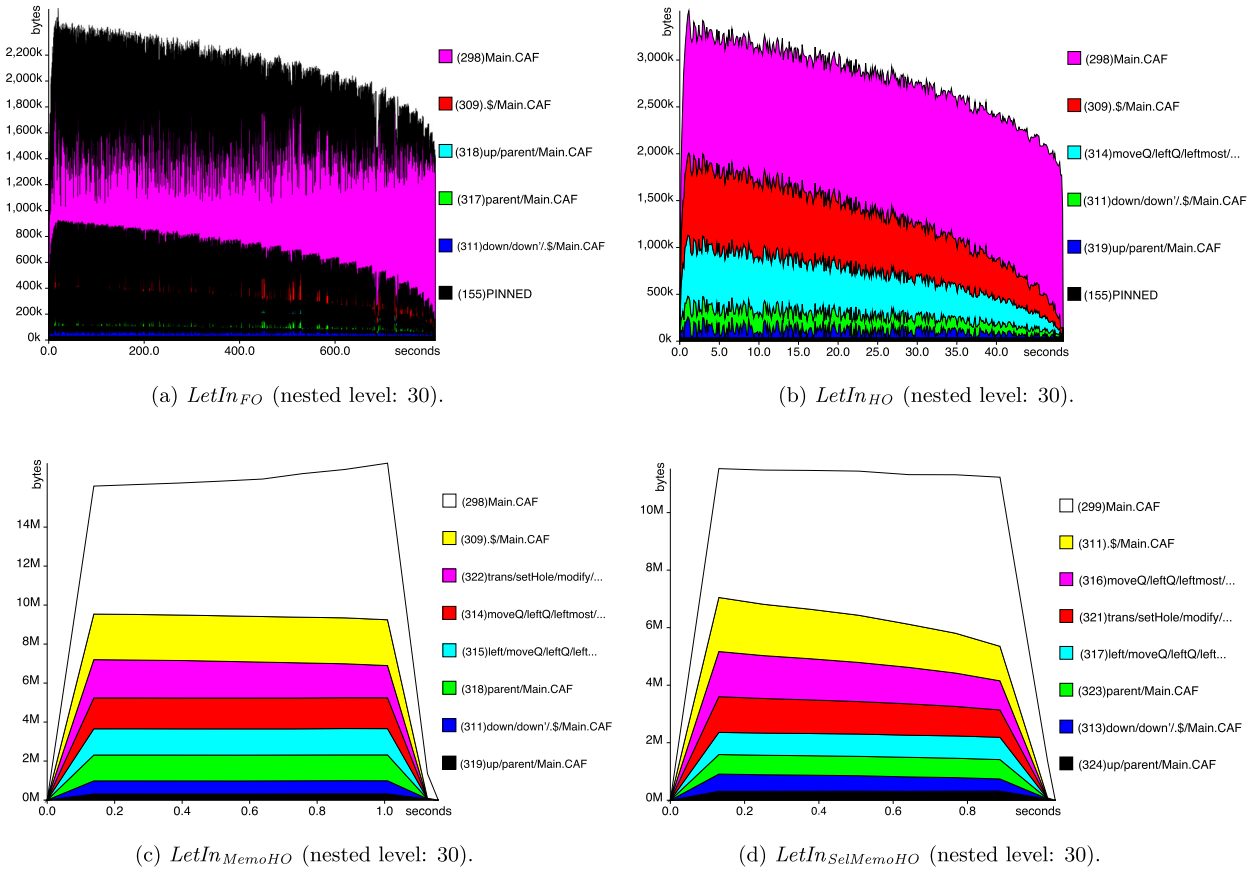
Fig. 13. Performance of the Let-In embedded attribute grammars **with** nested lets.

4.4. Selective memoization

One key advantage of using an embedded approach combined with a component-based programming style in AGs, is that we can easily plug-in different evaluators into a specification. We have shown how, in the Repmin example, the use of selective memoization can improve the performance of the evaluators. As another example, we can easily have a fourth evaluator for the Let-In AG, if in the $LetIn_{HO}$ we use the memoized evaluator of Algol 68 AG (that is, the memoized implementation we used in section 3.4.2). The resulting implementation is an hybrid evaluator where some attribute values are not memoized (the $algol_{68}$ and *calculate*) while others are (all attributes that decorate an *Algol68* tree).

Fig. 15 shows the performance results of this hybrid evaluator and the full memoized $LetIn_{MemoHO}$ when processing the same nested **let-in** programs.

These results show that by not memoizing the purely synthesized attributes $algol_{68}$ and *calculate*, we have a slightly more efficient implementation (both fit cubic curves, one with coefficient 0.52 and the other with 0.71). Indeed, the memoization of such attributes in $LetIn_{MemoHO}$ does increase memory consumption (Figs. 14c and 14d), while no reuse of previously



(a) $LetIn_{FO}$ (nested level: 30).

(b) $LetIn_{HO}$ (nested level: 30).

(c) $LetIn_{MemoHO}$ (nested level: 30).

(d) $LetIn_{SelMemoHO}$ (nested level: 30).

Fig. 14. Heap Profile on LetIn (values in Mbytes).

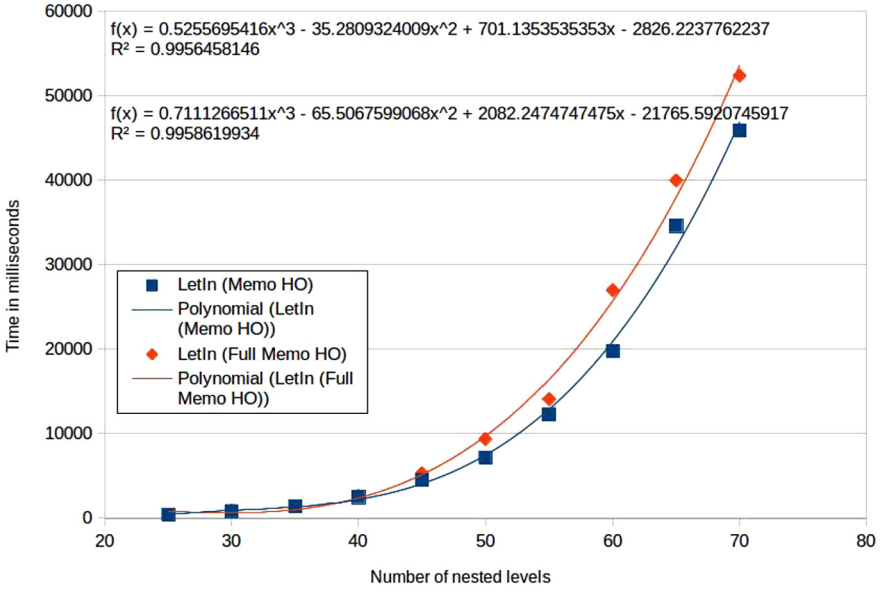


Fig. 15. Selective Memo HO versus Full Memoization HO.

computed values is achieved. This is exactly the same situation as in the HTML table formatter, where the evaluator memoizes all attributes including the synthesized one that builds the (large) ASCII string, which also results in a large memory consumption with no computation being reused.

In fact, memoization of all attribute values gives in principle optimal performance in the sense that each attribute value is computed only once. However, as our results suggest performance can be further improved by a selective memoization strategy. As future work, we plan to study techniques for the selective memoization of attribute values that can be incorporated to our memoized zipper-based HOAGs.

5. Related work

The use of zippers to model AGs has been proposed by several researchers. Uustalu and Vene have shown how to embed attribute computations using comonadic structures, where each tree node is paired with its attribute values [42]. This approach is remarkable for its use of a zipper as in our work. However, it appears that their zipper is not generic and must be instantiated for each tree structure. Laziness is used to avoid static scheduling, and no technique is defined to avoid attribute re-computations.

Badouel *et al.* define attribute evaluation as zipper transformers [2]. While their encoding is simpler than that of Uustalu and Vene, they also use laziness as a key aspect and the zipper representation is similarly not generic. Other work by Badouel *et al.* [4] also requires laziness and forces the programmer to be aware of a cyclic representation of zippers.

Yakushev *et al.* describe a fixed point method for defining operations on mutually recursive data types, with which they build a generic zipper [46]. Their approach is to translate data structures into a generic representation on which traversals and updates can be performed, and then to translate back. Even though their zipper is generic, the implementation is more complex than ours and includes the extra overhead of translation. It also uses advanced features of Haskell such as type families and rank-2 types.

In [26] we presented the zipper-based embedding of first-order AGs. Then, we extended this work to consider most modern attribute grammar extensions, including HOAGs [27]. In [13] we used memoization to develop efficient implementations for first-order AGs. None of these works consider the memoization of HOAGs.

Attribute grammars and circular, lazy functional programs are closely related, as shown by Johnsson [17]. As a consequence, lazy languages are a natural setting to implement AGs [25]. Circular programs were first presented by Bird as a technique to eliminate multiple traversals of data [6]. Lazy evaluation can be used to avoid recomputations, and can be combined with memoization [16]. However, the memoization of circular programs is still an open problem.

Attribute grammars are traditionally implemented as tree walk evaluators that traverse the underlying abstract syntax tree while computing attribute values [1]. The result of this process is a *decorated tree*, meaning that attribute values are stored in tree nodes, and thus, decorating it. In fact, imperative AG implementations may store attribute values in the tree nodes (as side effects). Attribute evaluators may also be defined as a set of recursive functions, where inherited attribute correspond to arguments and synthesized attributes correspond to results of such functions [19,35]. In both AG implementations, however, a memoization mechanism is necessary to pass attribute values computed in one tree traversal and used in a following one: imperative solutions use the tree nodes to memoize such values [20], while functional evaluators use additional (gluing) data structures [33]. Advanced AG static analysis techniques are used to analyze attribute dependencies (induced by the AG equations) to detect which attribute values do need to be memoized [20], so that not all attribute values are stored in the tree nodes.

On the contrary, incremental attribute evaluators memoize all attribute values in order to obtain an optimal re-evaluation time. After a change in the underlying tree [31], the dependency graph induced by the attribute equations is used to re-evaluate only the attributes affected by such a tree change. After the seminal work of Reps and Teitelbaum [31,32] on (incremental) syntax-oriented editors, several AG systems were developed to produce evaluators which memoize all attribute values, either in the tree's nodes of an imperative evaluator [32,12], or by memoizing function calls in a functional evaluator [30,24,37]. Our embedding of AG mimics the imperative evaluators in the sense that all attribute values are cached in tree nodes.

The incremental evaluation of HOAGs has being extensively studied by Swiersta's group [30,37,24,36] where AGs are compiled to purely functional attribute evaluators, and function memoization is used to achieve incremental evaluation. More recently, a new technique to compile HOAGs into incremental evaluators was developed: the evaluator maintains additional bookkeeping about which attribute values have changed [7,8]. However, the overhead that comes with the bookkeeping greatly impacts performance.

To improve the performance of attribute evaluators, Söderberg and Hedin propose automate selective memoization in the context of reference AGs [39]. As we discussed previously, selective memoization can improve the performance of full memoized zipper-based evaluators. Thus, we plan to incorporate automated selective memoization in our embedding.

6. Conclusion

Functional zippers provide a concise and natural setting to express AGs in a functional setting. In fact, AGs and their higher-order extension can be elegantly embedded in a functional language using a simple and light zipper-based naviga-

tion engine. The straightforward zipper-based embedding of AGs, however, has a severe drawback: it does not ensure that attributes are computed only once at tree nodes.

In this paper we showed how memoization is introduced in zipper-based embeddings of first order and higher-order AGs, so that every attribute value in the abstract tree is computed at most once. The memoized zipper-based embedding of (higher-order) AGs maintains the elegance of the original embedding, and exhibits much better performance, often by various different orders of magnitude.

As a possible direction of future research, we would like to test our approach with other embeddings of AGs, such as the ones of [42,3,4]. This comparison should be performed whenever possible (for example, it might be hard to perform with specific AG systems such as [11,24,43]), 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.

References

[1] Alblas Henk, Attribute evaluation methods, in: H. Alblas, B. Melichar (Eds.), International Summer School on Attribute Grammars, Applications and Systems, in: LNCS, vol. 545, Springer-Verlag, 1991, pp. 48–113.
[2] E. Badouel, R. Tchougong, C. Nkuimi-Jugnia, B. Fotsing, Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition, Theor. Comput. Sci. 480 (2013) 1–25.
[3] Eric Badouel, Bernard Fotsing, Rodrigue Tchougong, Yet Another Implementation of Attribute Evaluation, Research Report RR-6315, INRIA, 2007.
[4] Eric Badouel, Bernard Fotsing, Rodrigue Tchougong, Attribute grammars as recursion schemes over cyclic representations of zippers, Electron. Notes Theor. Comput. Sci. 229 (5) (2011) 39–56.
[5] Florent Balestrieri, The Productivity of Polymorphic Stream Equations and the Composition of Circular Traversals, PhD Thesis, University of Nottingham, 2015.
[6] Richard S. Bird, Using circular programs to eliminate multiple traversals of data, Acta Inform. 21 (1984) 239–250.
[7] Jeroen Bransen, Atze Dijkstra, S. Doaitse Swierstra, Incremental evaluation of higher order attributes, in: Kenichi Asai, Kostis Sagonas (Eds.), Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15–17, 2015, ACM, 2015, pp. 39–48.
[8] Jeroen Bransen, Atze Dijkstra, S. Doaitse Swierstra, Incremental evaluation of higher-order attributes, in: Selected and Extended Papers from Partial Evaluation and Program Manipulation 2015, PEPM'15, Sci. Comput. Program. 137 (2017) 98–124.
[9] Oege de Moor, Kevin Backhouse, Doaitse Swierstra, First-class attribute grammars, in: 3rd Workshop on Attribute Grammars and Their Applications, Ponte de Lima, Portugal, 2000, pp. 1–20.
[10] Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra, The architecture of the Utrecht Haskell compiler, in: Haskell Symposium, 2009, pp. 93–104.
[11] Atze Dijkstra, 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.
[12] Torbjörn Ekman, Görel Hedin, The JastAdd extensible Java compiler, SIGPLAN Not. 42 (10) (October 2007) 1–18.
[13] João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, Marcos Viera, Memoized zipper-based attribute grammars, in: Fernando Castor, Yu David Liu (Eds.), Programming Languages – Proceedings of 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22–23, 2016, in: Lecture Notes in Computer Science, vol. 9889, Springer, 2016, pp. 46–61.
[14] João Paulo Fernandes, João Saraiva, Tools and libraries to model and manipulate circular programs, in: Symposium on Partial Evaluation and Program Manipulation, ACM, 2007, pp. 102–111.
[15] Gérard Huet, The zipper, J. Funct. Program. 7 (5) (1997) 549–554.
[16] John Hughes, Lazy memo-functions, in: Jean-Pierre Jouannaud (Ed.), Functional Programming Languages and Computer Architecture, in: LNCS, vol. 201, Springer-Verlag, September 1985, pp. 129–146.
[17] Thomas Johnsson, Attribute grammars as a functional programming paradigm, in: Functional Programming Languages and Computer Architecture, 1987, pp. 154–173.
[18] Martin Jourdan, An optimal-time recursive evaluator for attribute grammars, in: M. Paul, B. Robinet (Eds.), International Symposium on Programming, Springer, Berlin, Heidelberg, 1984, pp. 167–178.
[19] Martin Jourdan, Strongly non-circular attribute grammars and their recursive evaluation, SIGPLAN Not. 19 (6) (June 1984) 81–93.
[20] Uwe Kastens, Implementation of visit-oriented attribute evaluators, in: H. Alblas, B. Melichar (Eds.), International Summer School on Attribute Grammars, Applications and Systems, in: LNCS, vol. 545, Springer-Verlag, 1991, pp. 114–139.
[21] Uwe Kastens, Peter Pfahler, Matthias T. Jung, The Eli system, in: Int. Conference on Compiler Construction, Springer-Verlag, 1998, pp. 294–297.
[22] Oleg Kiselyov, Ralf Lämmel, Keean Schupke, Strongly typed heterogeneous collections, in: Workshop on Haskell, ACM, 2004, pp. 96–107.
[23] Donald Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (June 1968); Correction, Math. Syst. Theory 5 (1) (March 1971).
[24] Matthijs Kuiper, João Saraiva, Lrc – a generator for incremental language-oriented tools, in: International Conference on Compiler Construction, Springer-Verlag, 1998, pp. 298–301.
[25] Matthijs Kuiper, Doaitse Swierstra, Using attribute grammars to derive efficient functional programs, in: Computing Science in the Netherlands, CSN'87, November 1987.
[26] Pedro Martins, João Paulo Fernandes, João Saraiva, Zipper-based attribute grammars and their extensions, in: André Rauber Du Bois, Phil Trinder (Eds.), Programming Languages – Proceedings of the 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3–4, 2013, in: Lecture Notes in Computer Science, vol. 8129, Springer, 2013, pp. 135–149.
[27] Pedro Martins, João Paulo Fernandes, João Saraiva, Eric Van Wyk, Anthony Sloane, Embedding attribute grammars and their extensions using functional zippers, Science of Computer Programming 132 (2016) 2–28.
[28] Arie Middelkoop, Atze Dijkstra, S. Doaitse Swierstra, Iterative type inference with attribute grammars, in: International Conference on Generative Programming, ACM, 2010, pp. 43–52.
[29] Ulf Norell, Alex Gerdes, Attribute grammars in Erlang, in: Workshop on Erlang, ACM, 2015, pp. 1–12, 2015.
[30] Maarten Pennings, Doaitse Swierstra, Harald Vogt, Using cached functions and constructors for incremental attribute evaluation, in: M. Bruynooghe, M. Wirsing (Eds.), Programming Language Implementation and Logic Programming, in: LNCS, vol. 631, Springer-Verlag, 1992, pp. 130–144.
[31] Thomas Reps, Optimal-time incremental semantic analysis for syntax-directed editors, in: Proceedings of the 9th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL '82, ACM, New York, NY, USA, 1982, pp. 169–176.
[32] Thomas Reps, Tim Teitelbaum, The synthesizer generator, SIGPLAN Not. 19 (5) (April 1984) 42–48.
[33] João Saraiva, Purely Functional Implementation of Attribute Grammars, PhD Thesis, Utrecht University, The Netherlands, December 1999.
[34] João Saraiva, Component-based programming for higher-order attribute grammars, in: Don S. Batory, Charles Consel, Walid Taha (Eds.), Generative Programming and Component Engineering, Proceedings of ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6–8, 2002, in: Lecture Notes in Computer Science, vol. 2487, Springer, 2002, pp. 268–282.

[35] João Saraiva, Doaitse Swierstra, Data structure free compilation, in: Stefan Jähnichen (Ed.), 8th International Conference on Compiler Construction, CC/ETAPS'99, in: LNCS, vol. 1575, Springer-Verlag, March 1999, pp. 1–16.
[36] João Saraiva, S. Doaitse Swierstra, Generating spreadsheet-like tools from strong attribute grammars, in: Frank Pfenning, Yannis Smaragdakis (Eds.), Generative Programming and Component Engineering, Proceedings of the Second International Conference, GPCE 2003, Erfurt, Germany, September 22–25, 2003, in: Lecture Notes in Computer Science, vol. 2830, 2003, pp. 307–323.
[37] João Saraiva, S. Doaitse Swierstra, Matthijs F. Kuiper, Functional incremental attribute evaluation, in: David A. Watt (Ed.), Compiler Construction, Proceedings of the 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25–April 2, 2000, in: Lecture Notes in Computer Science, vol. 1781, Springer, 2000, pp. 279–294.
[38] Anthony M. Sloane, Lennart C.L. Kats, Eelco Visser, A pure object-oriented embedding of attribute grammars, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 205–219.
[39] Emma Söderberg, Görel Hedin, Automated selective caching for reference attribute grammars, in: Proceedings of the Third International Conference on Software Language Engineering, SLE'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 2–21.
[40] Doaitse Swierstra, Pablo Azero, João Saraiva, Designing and implementing combinator languages, in: Third Summer School on Advanced Functional Programming, in: LNCS Tutorial, vol. 1608, Springer-Verlag, 1999, pp. 150–206.
[41] Doaitse Swierstra, Harald Vogt, Higher order attribute grammars, in: H. Alblas, B. Melichar (Eds.), International Summer School on Attribute Grammars, Applications and Systems, in: LNCS, vol. 545, Springer-Verlag, 1991, pp. 48–113.
[42] Tarmo Uustalu, Varmo Vene, Comonadic functional attribute evaluation, in: Trends in Functional Programming, in: Intellect Books, vol. 10, 2005, pp. 145–162.
[43] Eric Van Wyk, Derek Bodin, Jimin Gao, Lijesh Krishnan, Silver: an extensible attribute grammar system, Electron. Notes Theor. Comput. Sci. 203 (2) (2008) 103–116.
[44] Marcos Viera, Doaitse Swierstra, Wouter Swierstra, Attribute grammars fly first-class: how to do aspect oriented programming in Haskell, in: International Conference on Functional Programming, ACM, 2009, pp. 245–256.
[45] Harald Vogt, S. Doaitse Swierstra, Matthijs Kuiper, Higher order attribute grammars, SIGPLAN Not. 24 (7) (June 1989) 131–145.
[46] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, Johan Jeuring, Generic programming with fixed points for mutually recursive datatypes, in: Proc. of the 14th ACM SIGPLAN International Conference on Functional Programming, 2009, pp. 233–244.