Efficient Embedding of Strategic Attribute Grammars via Memoization

José Nuno Macedo
jose.n.macedo@inesctec.pt
HASLab & INESC TEC, University of Minho
Braga, Portugal

Emanuel Rodrigues
jose.e.rodrigues@inesctec.pt
HASLab & INESC TEC, University of Minho
Braga, Portugal

Marcos Viera
mviera@fing.edu.uy
Universidad de la República
Montevideo, Uruguay

João Saraiva
saraiva@di.uminho.pt
HASLab & INESC TEC, University of Minho
Braga, Portugal

Abstract

Strategic term re-writing and attribute grammars are two powerful programming techniques widely used in language engineering. The former relies on strategies to apply term re-write rules in defining large-scale language transformations, while the latter is suitable to express context-dependent language processing algorithms. These two techniques can be expressed and combined via a powerful navigation abstraction: generic zippers. This results in a concise zipper-based embedding offering the expressiveness of both techniques.

Such elegant embedding has a severe limitation since it recomputes attribute values. This paper presents a proper and efficient embedding of both techniques. First, attribute values are memoized in the zipper data structure, thus avoiding their re-computation. Moreover, strategic zipper based functions are adapted to access such memoized values. We have implemented our memoized embedding as the Ztrategic library and we benchmarked it against the state-of-the-art Strafunski and Kiama libraries. Our first results show that we are competitive against those two well established libraries.

CCS Concepts: • **Software and its engineering** → **Software development methods**; *Specification languages*; • **Theory of computation** → Program reasoning.

Keywords: Strategic Programming, Attribute Grammars, Zippers, Generic Traversals

ACM Reference Format:
José Nuno Macedo, Emanuel Rodrigues, Marcos Viera, and João Saraiva. 2023. Efficient Embedding of Strategic Attribute Grammars via Memoization. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3571786.3573019

PEPM '23, January 16–17, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0011-8/23/01…$15.00
https://doi.org/10.1145/3571786.3573019

1 Introduction

Strategic term re-writing [18] and Attribute Grammars (AG) [13] are two powerful language engineering techniques. The former provides an abstraction to define program/tree transformations: a set of re-write rules is applied while traversing the tree in some pre-defined recursion pattern, the strategy. The latter extends context-free grammars with attributes in order to specify static, context-dependent language algorithms.

There are many tools that support these techniques for the implementation of (domain specific) programming languages [3, 6, 8, 9, 11, 16, 17, 22, 23, 26, 30–32]. Unfortunately, most of these tools are large systems supporting one of the techniques, using their own AG or strategic specification language. As a consequence, they would require a considerable effort to extend and combine. There are, however, two exceptions: the Silver system [31] and the Kiama library [26] do support both techniques.

More recently, a combined embedding of the two techniques has been proposed in [19]. This embedding relies on a generic mechanism to navigate on both homogeneous and heterogeneous trees: generic zippers [1, 12]. Since both attribute grammars and strategies rely on the same generic tree traversal mechanism, each of the techniques can be expressed by generic zippers as shown in [20, 21], for AGs, and in [19], for strategic term re-writing. The embedding of the two techniques in the same simple setting has a key advantage: AGs and strategies embeddings can be easily combined, thus providing language engineers the best of the two worlds.

As previously shown in [10], the simple zipper-based embedding of AGs [20, 21] does not provide a proper embedding of the formalism: attribute values are re-calculated during the decoration of the tree. This not only goes against the semantics of AG formalism, where one attribute value is

computed at most once, but it also dramatically affects the attribute evaluator's performance. The combined embedding of strategies and AGs in that setting, as proposed in [19], has exactly the same performance issues.

In order to provide an efficient zipper-based embedding of strategic term re-writing and attribute grammars, that we call strategic AGs, we implement zipper-based strategies on top of the memoized zipper-based embedding of AGs [10]. Thus, strategies access memoized attribute values in the tree nodes, rather than having to re-compute such attribute values via the inefficient (non-memoized) embedding of AGs, as proposed in [19]. The purpose of this paper is three-fold:

- Firstly, we define zipper-based strategic combinators that can access memoized attribute values as supported by the efficient memoized embedding of zipper-based AGs [10]. Thus, we extend the Ztrategic library, developed in [19], with new combinators which work on trees where attribute values are memoized in the tree's nodes.
- Secondly, we improve the performance of the zipper-based strategic combinators. Influenced by the (attribute) grammar formalism, where terminal symbols are more suitable handled outside the formalism (usually specified via regular expressions and processed via efficient automata-based recognizers), we update zipper-based Ztrategic library combinators to not traverse such symbols. This does not limit the expressiveness of the strategic library, but does result in a considerable performance improvement of the implementations.
- Thirdly, we perform a detailed study on the performance of the non-memoized implementation proposed in [19] and our implementations. We consider four well-known language engineering tasks, namely, name analysis, program optimization, code smell elimination and pretty printing, which we elegantly expressed in the strategic and/or AG programming styles. Then, we compare the performance of our implementations against the state of the art Strafunski [17] system - the Haskell incarnation of strategic term re-writing - and Kiama [26] - the combined embedding of strategies and AGs in Scala.

Our preliminary results are surprising: the embedding of strategic term re-writing behaves similarly to Strafunski. However, the embedding of strategic AGs vastly outperforms Kiama's solutions.

This paper is organized as follows: Section 2 introduces strategic term re-writing, attribute grammars, a combined embedding of strategic AGs, and memoized AGs. Section 3 combines memoized AGs with strategies, and details a different implementation of *Ztrategic* that maximizes efficiency for the usage of memoized AGs; the concept of navigable symbols is also introduced in this library. Section 4 compares the performance of our work with the Strafunski and Kiama libraries, and elaborates on the obtained results. Section 5 details the relevant state of the art on strategic programming and AGs. Section 6 concludes our work and presents links to the relevant libraries and to a replication package.

$$add(e, const(0)) \longrightarrow e \quad (1)$$
$$add(const(0), e) \longrightarrow e \quad (2)$$
$$add(const(a), const(b)) \longrightarrow const(a+b) \quad (3)$$
$$sub(e1, e2) \longrightarrow add(e1, neg(e2)) \quad (4)$$
$$neg(neg(e)) \longrightarrow e \quad (5)$$
$$neg(const(a)) \longrightarrow const(-a) \quad (6)$$
$$var(id) \mid (id, just(e)) \in env \longrightarrow e \quad (7)$$

Figure 1. Optimization Rules

2 Zipping Strategies and Attribute Grammars

In this section, we describe the zipper-based Strategic Attribute Grammars embedding introduced in [19] that combines strategic programming and attribute grammars. Before we describe the embedding in detail, let us consider a motivating example that requires two widely used language engineering techniques: language analysis and language optimization. Consider the (sub)language of *Let* expressions as incorporated in most functional languages, including *Haskell*. Next, we show an example of a valid *Haskell* **let** expression

```
p = let a = b + 0
        c = 2
        b = let c = 3 in b + c
    in  a + 7 − c
```

and, we define the heterogeneous data type *Let* that we use to model let expressions in *Haskell* itself.

```
data Let  = Let List Exp
data List = NestedLet String Let List
          | Assign    String Exp List
          | EmptyList
data Exp  = Add   Exp Exp
          | Sub   Exp Exp
          | Neg   Exp
          | Var   String
          | Const Int
```

Consider now that we wish to implement a simple arithmetic optimizer for our language. Figure 1 presents such optimization rules directly taken from [15].

The first six optimization rules define context-free arithmetic rules. If we consider those six rules only, then strategic term re-writing is an extremely suitable formalism to express the desired optimization, since it provides a solution that just defines the work to be done in the constructors (tree nodes) of interest, and "ignores" all the others.

Strategic Term Re-writing: In fact, we can easily express this optimization in Ztrategic: the strategic term re-writing library of the combined embedding. We start by defining the *worker* function, that directly follows the six rules we are considering:

```
expr :: Exp → Maybe Exp
expr (Add e (Const 0))           = Just e
expr (Add (Const 0) t)           = Just t
expr (Add (Const a) (Const b)) = Just (Const (a + b))
expr (Sub a b)                   = Just (Add a (Neg b))
expr (Neg (Neg f))               = Just f
expr (Neg (Const n))             = Just (Const (−n))
expr _                           = Nothing
```

The worker function *expr* takes an *Exp* value, pattern matches on it, and in the cases of the rules returns the optimized expression. In all other cases, it returns *Nothing*. Notice that the optimizations are made locally, no recursion is involved. Having expressed all re-writing rules in function *expr*, now we need to use strategic combinators that navigate in the tree while applying the rules. In Figure 7 of the Appendix we show the complete API of Ztrategic, with all such possible combinators. In this case, to guarantee that all the possible optimizations are applied we use an *innermost* traversal scheme. Thus, our optimization is expressed as:

```
opt :: Zipper Let → Maybe (Zipper Let)
opt t = applyTP (innermost step) t
      where step = failTP `adhocTP` expr
```

Function *opt* defines a Type Preserving (*TP*) transformation; i.e. the input and result trees have the same type. Here, *step* is the transformation applied by the function *applyTP* to all nodes of the input tree *t* using the *innermost* strategy combinator. The re-write *step* performs the transformation specified in the *expr* worker function for all the cases considered by the optimizations and fails silently (*failTP*) in other case. Notice in the signature the use of **Zipper** to navigate through the structure (of type *Let*) to be transformed.

Let us now consider the context dependent rule 7 in our optimization. This rule requires the computation of the environment where a name is used. This environment has to be computed according to the non-trivial scope rules of the *Let* language. The semantics of *Let* does not force a declare-before-use discipline, meaning that a variable can be declared after its first use. Consequently, a conventional implementation of the scope rules naturally leads to an algorithm, that traverses each block twice: once for accumulating the declarations of names and constructing an environment and a second time to process the uses of names (using the computed environment) in order to check for the use of non-declared identifiers.

In fact, both the scope rules and context dependent re-writing are not easily expressed within strategic term re-writing.

Attribute Grammars: The formal specification of scope rules is in the genesis of the Attribute Grammar formalism [14]. AGs are particularly suitable to specify language engineering tasks, where context information needs to be first collected before it can be used.

We start by specifying the scope rules of *Let* via an AG. Due to space limitations, we adopt a visual AG notation that is often used by AG writers to sketch a first draft of their grammars. Thus, the scope rules of *Let* are visually expressed in Figure 2. We define an extra type Root, to identify the root of the tree:

```
data Root = Root Let
```

The diagrams in the figure are read as follows. For each constructor/production (labeled by its name) we have the type of the production above and below those of its children. To the left of each symbol we have the so-called *inherited attributes*: values that are computed top-down in the grammar. To the right of each symbol we have the so-called *synthesized attributes*: values that are computed bottom-up. The arrows between attributes specify the information flow to compute an attribute. Thus, the AG expressed in Figure 2 is the following. The inherited attribute *dcli* is used as an accumulator to collect all *names* defined in a *Let*: it starts as an empty list in the *Root* production, and when a new name is defined (productions *Assign* and *NestedLet*) it is added to the accumulator. The total list of defined names is synthesized in attribute *dclo*, which at the *Root* node is passed down as the environment (inherited attribute *env*). Moreover, a nested let inherits (attribute *dcli*) the environment of its outer let. The type of the three attributes is a list of pairs, associating the name to its *Let* expression definition.

The ZipperAG [20] library of the combined embedding defines a set of simple AG-like combinators; namely the combinator "*child*", written as the infix function .$, to access the child of a tree node given its index, and the combinator *parent* to move the focus to the parent of a tree node. With these two zipper-based AG combinators, we are able to express in a AG programming style the scope rules of *Let*. For example, let us consider the synthesized attribute *dclo*. In the diagrams of our visual AG the *NestedLet* and *Assign* productions we see that *dclo* is defined as the *dclo* of the third child. Moreover, in production *EmptyList* attribute *dclo* is a copy of *dcli*. This is exactly how such equations are written in the zipper-based AG, as we can see in the next function[1]:

```
dclo :: AGTree [(String, Zipper Root)]
dclo t = case (constructor t) of
           Let_Let       → dclo (t.$1)
           NestedLet_List → dclo (t.$3)
           Assign_List   → dclo (t.$3)
           EmptyList_List → dcli t
```

[1]The function constructor and the constructors used in the case alternatives is boilerplate code needed by the AG embedding. This code is defined once per tree structure (*i.e.*, AG), and can be generated by template *Haskell* [25]

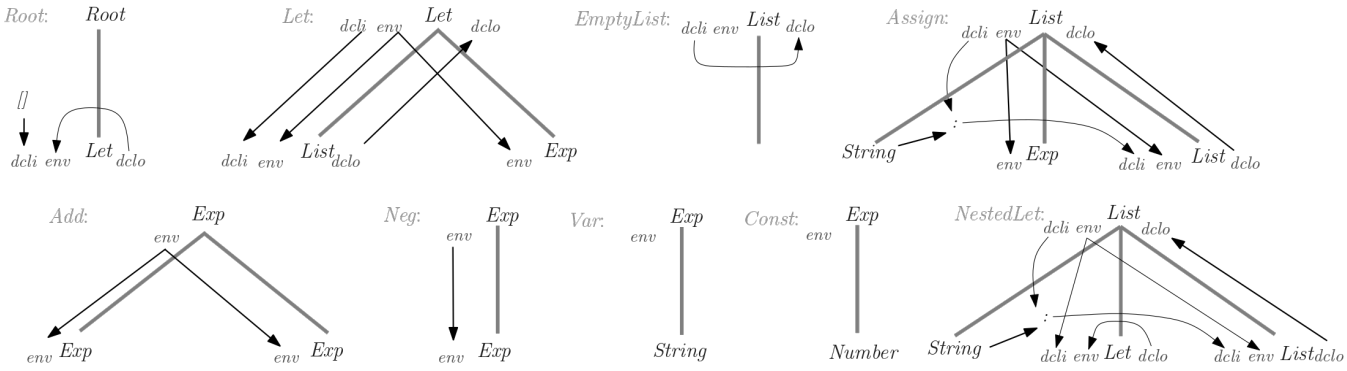

Figure 2. Attribute Grammar Specifying the Scope Rules of *Let*

Consider now the case of defining the inherited attribute *env* that we will need to express optimization (7). In most diagrams an occurrence of attribute *env* is defined as a copy of the parent. There are two exceptions: in productions *Root* and *NestedLet*. In both cases, *env* gets its value from the synthesized attribute *dclo* of the same non-terminal/type. Thus, the *Haskell env* function looks as follows:

```
env :: AGTree [(String, Zipper Root)]
env t = case (constructor t) of
          Root_P   → dclo t
          Let_Let  → dclo t
          _        → env (parent t)
```

We omit here the definition of attribute *dcli*, where the declared names are being accumulated.

Combining Strategies and Attribute Grammars: AG evaluators decorate the underlying trees with attribute values. Thus, an instance of attribute *env* is associated to every *Var* node, defining its environment. Recall that *env* of *var*(*id*) is the missing ingredient to implement rule (7).

Since we work with a combined embedding, we define a strategic re-writing worker function that implements rule 7:

```
expC :: Exp → Zipper Root → Maybe Exp
expC (Var i) z = expand (i, lev z) (env z)
expC _       z = Nothing
```

The variable *i* is expanded according to its environment, as defined by rule 7. Because the *Let* language has nesting we use an attribute named *lev* to distinguish definitions with the same name at different nested levels. Thus, the *expand* function looks up the defined variable *i* in the level *lev* or a lower level, in its environment *env*. In case it is found, the expanded definition of variable *i* is returned, otherwise the optimization is not performed. The definitions of the function *expand* and the attribute *lev* are omitted for brevity.

Now we combine this rule with the previously defined *expr*, implementing rules 1 to 6, and apply them to all nodes.

```
opt' :: Zipper Root → Maybe (Zipper Root)
opt' r = applyTP (innermost step) r
       where step = failTP `adhocTPZ` expC `adhocTP` expr
```

Our motivating example shows the abstraction and expressiveness provided by combining strategies and attribute grammars in the same zipper-based setting [19]. However, this embedding of AGs has a severe limitation since when decorating the tree, it re-computes the same attribute instances. The reader may have noticed that every time the worker function *expC* is called, then the call to *env z* does lead to the (re)decoration of the full tree. Thus, the number of calls to rule (7) results in the same number of full tree (re)decorations. As expected, this drastically affects the performance of the AG embedding[10] and, consequently, of the combined one, as well.

2.1 Term Re-Writing via Higher Order AGs

Classical AGs have a severe drawback: every computation has to be expressed in terms of the underlying AST. In fact, Higher-order AGs (HAG) [33] were introduced with the main goal of solving this limitation. In HAGs when a computation cannot be easily expressed in terms of the original AST, a better suited data structure can be computed before. Thus, HAG do support term re-writing as shown in [24]. The zipper based embedding of AGs supports this extension [21]. Next, we show the *Let* optimization in a pure HAG setting.

```
optRoot :: Zipper Root → Root
optRoot ag = case (constructor ag) of
  Root_P    → Root $ optLet (ag.$1)

optLet :: Zipper Root → Let
optLet ag = case (constructor ag) of
  Let_Let   → Let (optList (ag.$1)) (optExp (ag.$2))

optList :: Zipper Root → List
optList ag = case (constructor ag) of
  EmptyList_List → EmptyList
  Assign_List    → Assign (lexeme_Name ag)
                          (optExp (ag.$2)) (optList (ag.$3))
  NestedLet_List → NestedLet (lexeme_Name ag)
                             (optLet (ag.$2)) (optList (ag.$3))

optExp :: Zipper Root → Exp
optExp ag = case (constructor ag) of
  Add_Exp   → case (lexeme_Add1 ag, lexeme_Add2 ag) of
```

```
                (e, Const 0)  → e
                (Const 0, t)  → t
                (Const a, Const b) → Const (a + b)
                _             → Add (optExp (ag.$1)) (optExp (ag.$2))
Sub_Exp     → Add (lexeme_Sub1 ag) (Neg (lexeme_Sub2 ag))
Const_Exp   → Const (lexeme_Const ag)
Neg_Exp     → case (lexeme_Neg ag) of
                (Neg (Neg f))   → f
                (Neg (Const n)) → Const (−n)
                _               → Neg (optExp (ag.$1))
Var_Exp     → case expand (lexeme_Var ag, lev ag) (env ag) of
                Just e    → e
                Nothing → Var (lexeme_Var ag)
```

This fragment expresses a single transformation/re-writing of the original AST into a new (higher-order) tree. In order to guarantee that all the possible optimizations are applied, we define a circular attribute higher-order attribute, named *attributable attribute ata*, that is evaluated until a fix point is reached [28].

```
circ :: Root → Root
circ = fix (λf ata → if ata ≡ (optRoot $ mkAG ata)
                     then ata else f (optRoot $ mkAG ata))
```

The higher-order solution corresponds to the computation of this higher-order attribute and we write *optHAG* = *circ*.

As clearly shown in *optHAG*, term re-writing via circular HAG does not offer the expressiveness offered by strategic programming. Firstly, all non-terminals/types and their productions/constructors are included in the HAG solutions, even when there is no useful work to be performed there. Secondly, the recursion scheme is fixed and coded directly in attribute equations. Thus, it can not be reused. In fact, the definition of traversals and recursion schemes is against the declarative nature of standard AGs. In order to avoid trivial and polluting equations, AG systems offer a set of copy-rule abstractions allowing the automatic generation of such attribute equations. Thus, we may consider some form of attribute equation generation that always generates the equations that call the constructor without doing useful work. The automatic generation of copy rules, however, may induce hidden (real) circular attribute dependencies, that are hard to identify and debug.

2.2 Memoized Attribute Grammars

In order to avoid attribute re-computation and, consequently, to improve the performance of the AG embedding, memoization was incorporated into the zipper-based AGs [10]. To memoize the computed attributes for a given data structure, a new similar data structure is defined where a memoization table (here referred to as *m*) is associated with each node. All dependent data structures are merged into a single one, which allows for easier handling of the memoization tables:

```
data Let m = Root       (Let m)                 m
           | Let        (Let m) (Let m)         m
           | NestedLet String (Let m) (Let m)   m
           | Assign     String (Let m) (Let m)  m
           | EmptyList                          m
           | Add        (Let m) (Let m)         m
           | Sub        (Let m) (Let m)         m
           | Neg        (Let m)                 m
           | Var        String                  m
           | Const      Int                     m
```

Thus, the type of the memoization table for any given node can be, for example, a tuple in which each value might contain a memoized attribute. In our *Let* example, this tuple and an empty memoization table are defined as follows:

```
type MemoTable = (Maybe Env   -- dcli
                 ,Maybe Env   -- dclo
                 ,Maybe Env)  -- env
emptyMemo = (Nothing, Nothing, Nothing)
```

Next, we have to define *Let* as an instance of the *Memoizable* data class, with a memoization table of type *MemoTable*.

```
instance Memoizable Let MemoTable where
  updMemoTable :: (m → m) → Let m → Let m
  updMemoTable = updMemoTable′
  getMemoTable :: Let m → m
  getMemoTable = getMemoTable′
```

We omit the definition of the functions *getMemoTable*′ and *updMemoTable*′ as they are simple *Haskell* functions that get and replace the memoization table of a node, respectively.

Each of the attributes to be memoized is defined as a data type. This will be useful in determining which attribute is to be memoized when computing attributes.

```
data Att_dcli = Att_dcli
data Att_dclo = Att_dclo
data Att_env = Att_env
```

Finally, we define how each of the attributes is stored in the memoization table. For this, we use the *Memo* data class, specifying, for example, how the attribute *dcli* interacts with our *MemoTable*, storing a value of type *Env*:

```
instance Memo Att_dcli MemoTable Env where
  mlookup _ (a, _, _)  = a
  massign _ v (a, b, c) = (Just v, b, c)
```

Here, the function *mlookup* defines how to obtain a *dcli* value from the memoization table and *massign* defines how to update it. We define similar instances for the other attributes.

The definition of the instances of *Let* and its attributes as instances of *Memo* and *Memoizable* allow for the usage of the **memo** function, which hides all the memoization work enabling writing memoized attributes in a similar fashion to the non-memoized examples shown before. Next, we redefined the *dclo* attribute using memoization:

```
dclo :: (Memo Att_dclo MemoTable Env)
     ⇒ AGTree_m Let MemoTable Env
dclo = memo Att_dclo $
          λag → case (constructor ag) of
            Root_P   → dclo .@. (ag.$1)
            Let_Let  → dclo .@. (ag.$1)
```

```
NestedLet_List  → dclo .@. (ag.$3)
Assign_List     → dclo .@. (ag.$3)
EmptyList_List  → dcli ag
```

This attribute will be computed similarly to the non-memoized version when there is no value computed for it previously, and the result will be automatically stored in the memoization table. If the attribute was computed previously, then the previous value is re-used.

The *env* attribute defined in this fashion will be stored automatically when computed, and further uses of this attribute will just re-use the previously computed value. Any attributes that are used to compute *env* are also memoized.

```
env :: (Memo Att_env MemoTable Env)
    ⇒ AGTree_m Let MemoTable Env
env = memo Att_env $
            λag → case (constructor ag) of
              Root_P → dclo ag
              Let_Let → dclo ag
              _       → env `atParent` ag
```

The combinators (.@.) and *atParent* perform an attribute computation at a given child and at the parent, respectively, returning the result of the computation and new tree with the memotables possibly updated. Thus, attribute computations are represented by a *State*-monad with type:

```
type AGTree_m dtype m a = Zipper (dtype m)
                        → (a, Zipper (dtype m))
```

3 Combining Attribute Memoization with Zippers

As shown in [10] the attribute evaluation of memoized zipper-based AGs is much faster than the non-memoized one. As we will show in Section 4, the combined embedding proposed in [19] suffers from the same performance issues. Before we discuss such performance results, we introduce a new set of strategic combinators that do work with memoized attributes, yielding an efficient embedding of both techniques.

3.1 Memoized Strategies

The memoized attributes showcased previously are extremely powerful performance-wise and they can be used directly with the existing Ztrategic library. In fact, the memoized attributes produce two outputs: the actual attribute value, as well as the resulting data structure, with all computed attributes stored in the respective memoization tables. By ignoring the updated data structure and using only the attribute value, these memoized attributes can be plugged directly into Ztrategic. We define *exprM*, similar to previously defined *expr* but operating on the memoized data type *Let MemoTable*. When the node is transformed and there is no memoization table to place in the new node, we use *emptyMemo* to define an empty table.

```
exprM :: Let MemoTable → Maybe (Let MemoTable)
exprM (Add e (Const 0 _) m)              = Just $ e
exprM (Add (Const 0 _) t _)              = Just $ t
exprM (Add (Const a _) (Const b _) m) = Just $ Const (a + b) m
exprM (Sub a b m) = Just $ Add a (Neg b emptyMemo)
                           emptyMemo
exprM (Neg (Neg f _) _)                  = Just $ f
exprM (Neg (Const n m) _)                = Just $ Const (−n) m
exprM _                                  = Nothing
```

We define *exprX*, similar to previously defined *expC* but using memoized attributes. Recall that this function applies optimization rule 7, replacing a variable name by its definition whenever is possible. We use again the auxiliary function *expand* to perform this task. Because the result is an *Exp* which is not compatible with the memoized *Let* datatype, we use the function *buildMemoTreeExp* to convert it.

```
exprX :: Let MemoTable → Zipper (Let MemoTable)
      → Maybe (Let MemoTable)
exprX (Var i _) z = let (e, _) = env z
                        (l, _) = lev z
                    in  fmap (buildMemoTreeExp emptyMemo) $
                             expand (i, l) e
exprX _ z = Nothing
```

Notice that we are ignoring the second component of the results of the evaluation of *env* and *lev* which is the zipper with the updated memotables. Having defined the type specific worker functions *exprX* and *exprM*, we can now build a strategy to apply them to all nodes, through the *innermost* strategy. Because we have two different data types here, namely the non-memoized *Let* nodes, here denoted by *Root*, and the memoized *Let* nodes denoted by *Let MemoTable*, we use auxiliary functions *buildMemoTree* and *letToRoot* to convert the types before and after application of the strategy.

```
opt :: Root → Root
opt t = letToRoot (fromZipper t')
  where z :: Zipper (Let MemoTable)
        z = toZipper (buildMemoTree emptyMemo t)
        Just t' = applyTP (innermost step) z
        step = failTP `adhocTPZ` exprX `adhocTP` exprM
```

We have just presented our first zipper-based strategic AG definition using memoized attributes. However, this optimization strategy is much slower than the previous definition which does not use memoized attributes. To solve this, we need to dig deeper into how internal data structure navigation in strategies is defined, through the zipper mechanism.

The navigation in a zipper is provided by the generic zippers library [1]. This library includes the **toZipper** :: *Data a* ⇒ *a* → **Zipper** *a* function that produces a zipper out of any data type, requiring only that the data types have an instance of the Data and Typeable type classes. It includes also functions **right**, **left**, **down** and **up** to move the focus of the zipper towards the corresponding directions. They all have type **Zipper** *a* → *Maybe* (**Zipper** *a*), meaning that such functions take a zipper and return a new zipper when the navigation does not fail. There are also functions to get and set the node the zipper is focusing on,

namely **getHole** :: *Typeable b* ⇒ **Zipper** *a* → *Maybe b* and **setHole** :: *Typeable a* ⇒ *a* → **Zipper** *b* → **Zipper** *b*.

While navigating in a zipper where memotables are nodes of the tree, we need to guarantee that memotables are not considered when traversing all the nodes of a memoized zipper. In fact, part of the performance loss of our new definition of **let** derives from unneeded strategic traversals inside each node's memoization tables. Thus, we start by defining new versions of the generic zippers functions that do avoid navigating in some nodes of the tree, namely the memotable. Next, we define function *right′* that has this behaviour.

```
right′ :: StrategicData a ⇒ Zipper a → Maybe (Zipper a)
right′ z = case right z of
  Just r  → if isNavigable r then Just r else right′ r
  Nothing → Nothing
```

This function checks if the node we are traversing towards is navigable as defined by function *isNavigable* (defined in class *StrategicData* that we will explain in Section 3.3). If it is, *right′* behaves as the zipper function **right** would. If it is not, we skip it by navigating to the right again. There is a function named *left′* with an equivalent behaviour. In Section 3.3 we extend this notion of navigable nodes to other (AG) symbols, and we will show how to define a memotable to not be navigable.

Having defined the memoization tables as nodes that are not navigable, the performance of the new *opt* implementation using memoized attributes is better, but still worse than the non-memoized version. The strategies previously shown assume that the underlying data structure does not change. For a tree with several nodes, transforming a single node should result in changing that same node without impacting the rest of the tree. However, when combining strategies with memoized AGs, this is not the case anymore. When traversing a node, we might compute an attribute which will traverse the whole tree while computing and/or memoizing attributes in all of its nodes, effectively changing them.

Because of this, a memoization-friendly version of the Ztrategic library was developed. The focus of this version of the library will be in allowing the propagation of memoization throughout the data structure while traversing it. For each visited node, instead of returning just the result of traversing that node, we also return the updated zipper of the data structure. While keeping this in mind, the semantics are kept as similar to the original implementation as possible.

Before we present the combinators that navigate in memoized zippers, let us start by presenting some basic functions that are their building blocks. First, we introduce a function to elevate a user-defined function to the zipper level. Let us recall the original definition of this function, in *Ztrategic*:

```
zTryApplyMZ :: (Typeable a, Typeable b)
            ⇒ (a → Zipper c → Maybe b) → TP c
```

Our new implementation of this function follows a similar type signature:

```
zTryApplyMZ :: (Typeable a)
            ⇒ (a → Zipper c → Maybe (Zipper c)) → TP c
```

We omit the definition of *zTryApplyMZ* for brevity: it requires a function that takes the node *a* to be transformed, as well as **Zipper** *c* which points to the same node, and returns a *Maybe* (**Zipper** *c*), meaning either an updated zipper, or a *Nothing* value representing no changes. Note that the required function should output an updated **Zipper** *c*, instead of a plain value *b* as was required in the original definition, i.e. the function can be a memoized attribute, that updates memotables in the zipper. The *zTryApplyMZ* function returns a *TP c*, in which TP is a type for specifying Type-Preserving transformations on zippers, and *c* is the type of the zipper. It is defined as follows:

```
type TP a = Zipper a → Maybe (Zipper a)
type TU m d = (forall a . Zipper a → (m d, Zipper a))
```

For example, if we are applying transformations on a zipper built upon the *Let* data type, then those transformations are of type *TP Let*. Similarly to Strafunski and Ztrategic, we also introduce the type *TU m d* for Type-Unifying operations, which aim to gather data of type *d* into the data structure *m*.

Unlike in Ztrategic, these transformations also return a **Zipper** *a* value, which is the updated version of the input zipper. Therefore, both Type-Preserving and Type-Unifying strategies will update the data structure being traversed, as required by the memoized AGs.

Next, we define a combinator to compose two transformations, building a more complex zipper transformation that tries to apply each of the initial transformations in sequence, skipping transformations that fail.

```
adhocTPZ :: Typeable a ⇒ TP (d m) →
  (a → Zipper (d m) → Maybe (Zipper (d m))) → TP (d m)
adhocTPZ f g = maybeKeep f (zTryApplyMZ g)
```

Very much like the *adhocTP* combinator described in [19], the *adhocTPZ* function receives transformations f and g as parameters, as well as zipper *z*. The previously shown *zTryApplyMZ* function changes *g* into a *TP* transformation, which is then combined with *f* by function *maybeKeep*. Function *maybeKeep* tries to apply the second function it receives (here it being the transformed *g* function), and if it fails, *f* is applied instead.

Let us return to the *Let* optimization described in previous section. Let us also consider a function *exprZM*, similar to *expr* but receiving also a zipper as argument and returning an updated zipper (we will be defining this function later). Then, we can use *adhocTPZ* to combine the *exprZM* function with the default failing strategy *failTP*:

```
step = failTP `adhocTPZ` exprZM
```

The rest of the Ztrategic library is re-written to accommodate for the different definitions of the types of transformations, including the functions *failTP* and *idTP*.

Using the *right′* zipper navigation function defined before, we can now define for example a combinator that navigates in all navigable nodes of a tree.

```
allTPright :: StrategicData (d m) ⇒ TP (d m) → TP (d m)
allTPright f z = case right′ z of
  Nothing → return (z, z)
  Just r  → fmap (fromJust . left′) (f r)
```

This function is a combinator that, given a type-preserving transformation f for zipper z, tries to travel to the node located to the right using zipper function *right′*, and if it succeeds, it applies f and returns with function *left′*. If it fails, the original zipper is returned. Because the result of f is an optional *Maybe* value, we use *fmap* to apply navigation back to the left inside it. There is also a similar *allTPdown* combinator that navigates downwards on the zipper.

The definition of high-level strategies, such as *full_tdTP* (full, top-down, type-preserving), is similar to *Ztrategic*. However, the input data must be an instance of *StrategicData* so that they do consider the introduced navigable mechanisms. We refer to the definition of *innermost* in [19], and we show our updated definition:

```
innermost :: StrategicData (d m) ⇒ TP (d m) → TP (d m)
innermost s = repeatTP (once_buTP s)
```

The full API of our extended versions of Ztrategic is available in Figure 7 in appendix.

Let us return to our Let running example. Function *exprX* applied rule 7 through the usage of memoized attributes, but with the updated data structures they produce being ignored. Let us now define function *exprMZ*, similar to function *exprX* but reusing the updated zippers that the memoized attributes produce, which we label z' and z''. We change this updated zipper by using the zipper function **setHole** to set its current value to *expr*, which is the value we would return directly in previous definition *exprX*.

```
exprZM :: Let MemoTable → Zipper (Let MemoTable)
       → Maybe (Zipper (Let MemoTable))
exprZM (Var i _) z
  = let (e, z′)  = env z
        (l, z′′) = lev z′
        expr :: Maybe (Let MemoTable)
        expr = fmap (buildMemoTreeExp emptyMemo) $
                    expand (i, l) e
    in fmap (λk → setHole k z′′) expr
exprZM _       z = Nothing
```

We once again define a strategy for optimization of *Let* expressions, using memoized attributes and the new *Ztrategic* library module for propagation of memoization tables. We use the *exprM* function defined previously, as it does not depend on attributes and thus does not need any changes.

```
opt :: Root → Root
opt t = letToRoot (fromZipper $ fromJust
                                (applyTP (innermost step) z))
  where z :: Zipper (Let MemoTable)
        z = toZipper (buildMemoTree emptyMemo t)
        step = failTP `adhocTPZ` exprZM `adhocTP` exprM
```

This version of the *Let* expression optimization strategy is much more efficient than the non-memoized version presented before. Although this memoization mechanism introduces some intrusive code in our definitions, the improvement in terms of runtime is worth this effort. We compare the performance of non-memoized and memoized approaches in detail in Section 4.

3.2 Memoization Table Correctness

The running example in this paper does not address an underlying concern with attribute memoization, specifically, the invalidation of outdated attribute computations. Since Type-Preserving strategies change the underlying data structure, actions such as updates, insertions and removal of nodes can make the values of certain attributes invalid. If an attribute value is memoized first and then the data structure is changed, said value is kept memoized, and thus following attribute computations on that node yield incorrect results.

We address this problem by providing two alternative strategy application patterns for data structures with memoization. We use *applyTP_unclean* when we do not care about cleaning the memoization tables after application of a strategy, which would be the case for the running example in this paper. If such cleaning of memoization tables is required, the combinator *applyTP* will perform such cleaning after application of a strategy.

We showcase this problem by defining a simple attribute, named *adds*, which counts the number of *Add* nodes in a *Let* tree. The number of *Add* nodes in a *Let* tree decreases when optimization *opt* is applied due to several optimization rules being targeted at these nodes. As such, we define:

```
optValue_unclean l = adds (opt_unclean (adds l))
optValue_clean   l = adds (opt_clean   (adds l))
```

Both of these values contain an initial usage of attribute *adds* which 1) counts the number of *Add* nodes in a given *Let* (we ignore this value), and 2) memoizes all computed attributes in said *Let*. With the values of *adds* already memoized in the data structure, one variant of *opt* is then used to optimize the data structure. Finally, attribute *adds* is used again to compute the number of *Add* nodes, and this usage will attempt to look up the memoized values in the data structure. For *optValue_unclean*, the incorrect, pre-optimization value is obtained, while *optValue_clean* will produce the correct result.

We can guarantee node-to-node correctness of attributes through the usage of the non-memoized version of *Ztrategic*, such that the usage of memoized attributes does not change the underlying data structure's memoization tables, computing only the attribute value. We do this when defining *exprX*, which we have concluded to be inefficient.

Thus we have the trade-off of gaining performance at the cost of potential correctness of the results when using memoized attributes with strategies. It is the responsibility of the programmer to be careful on if any memoized attributes change with the transformations being applied to the data structure, and if they do, to use the proper mechanisms to work around it. For type unifying strategies, such concerns are unneeded as the data structure is being consumed.

3.3 Navigable Symbols

Since strategies (typically) traverse all nodes of a data structure, every memoization table stored in every node would be treated as additional data and be unnecessarily traversed. As shown in the definition of zipper function *right'*, we use the predicate *isNavigable* to avoid traversing the memotable nodes. When defining a strategic AG, however, there are other symbols that may not be traversed, since they are usually handled outside the grammar formalism. Indeed, the terminal symbols of a grammar are usually handled outside the formalism. They are often specified via regular expressions, and not by grammars. Moreover, they are efficiently processed via efficient automata-based recognizers. Thus, we consider terminal symbols as non navigable symbols/values in our trees, as opposed to Strafunski and Ztrategic.

To allow for this behaviour, any data type to be traversed using this library must define an instance of *StrategicData*. As an example, we can define this instance easily with the default behaviour of not skipping any nodes, for the Let datatype:

```
instance StrategicData Let
```

We could optimize this by instead of defining our own behaviour for our data, for example by skipping any node that is an *Int* or a *String*, which we expect to never want to traverse directly. That is not to say that we cannot change any *Int* or *String* in our traversals; we would expect to change *Strings* when traversing a *Var* constructor, but not by directly visiting the *String* node (which in itself is a list of *Char* that would also be traversed individually). We also define *MemoTable* as not navigable. We can define it like so:

```
instance StrategicData (Let MemoTable) where
  isNavigable z = ¬ (isJust (getHole z :: Maybe MemoTable)
                  ∨ isJust (getHole z :: Maybe String )
                  ∨ isJust (getHole z :: Maybe Int    ))
```

The *isNavigable* function should return false whenever *z* points towards a terminal symbol/node or to the memotable node. In this case, besides the memotable we consider only two terminal symbols, *String* and *Int*, but we could define more complex logic in this function, such as skipping only negative integers.

We reflect of the performance impact of this change in section 4. We also require this change to define memoization in a strategic setting.

4 Performance

In this section, we compare the performance of the Ztrategic library with state-of-the-art libraries Strafunski and Kiama. In terms of expressiveness, Ztrategic is capable of representing AGs, strategies, and the combination of AGs and strategies in a unified setting. Thus, we focus our analysis in the runtime and memory consumption of different implementations of a Haskell code smell eliminator, a *repmin* program, a *Let* optimizer (as described in this paper), and an advanced multiple layout pretty printing algorithm.

All implementations of these strategic AGs, together with the necessary resources (tests, scripts, etc) to replicate our study, are available in our replication package as detailed in Section 6. All tests were run 10 times and averaged in a ThinkPad 13 (2nd Gen, Intel i7-7500U (4) 3.500GHz) laptop with 8 Gb RAM and EndeavourOS Linux x86 64 bits.

4.1 Strategic Haskell Smell Elimination

Source code smells make code harder to comprehend. A smell is not an error, but it indicates a bad programming practice. Smells occur in any language and Haskell is no exception. For example, inexperienced Haskell programmers often write $l \equiv [\,]$ to check whether a list is empty, instead of using the predefined *null* function. We implemented this full Haskell language refactoring tools as a pure strategic program. It detects and eliminates all Haskell smells as reported in [7].

In order to compare Ztrategic with the Haskell state-of-the-art Strafunski counterpart we run both strategic solutions with a large *smelly* input. We consider 150 Haskell projects developed by first-year students as presented in [2]. In these projects there are 1139 Haskell files totaling 82124 lines of code, of which exactly 1000 files were syntactically correct [2]. Both Ztrategic and Strafunski smell eliminators detected and eliminated 850 code smells in those files. Figure 3 shows the runtime (left) and memory consumption (right) of running both libraries.

There are three entries in these figures: a normal Ztrategic implementation, a Ztrategic implementation in which we skip unnecessary nodes (corresponding to *terminal symbols*) in the traversal, and a similar implementation in Strafunski.

Strafunski outperforms Ztrategic, which is to be expected as Ztrategic library has an additional overhead of creating and handling a zipper over the traversed data. However, when skipping terminals, Ztrategic has almost the same performance of the well established and fully optimized Strafunski system.

4.2 Repmin as a Strategic Program

The *repmin* problem is a well-known problem widely used to show the power of circular, lazy evaluation as shown by Bird[4]. The goal of this program is to transform a binary

[2]The student projects used in this benchmark are available at this work's repository.

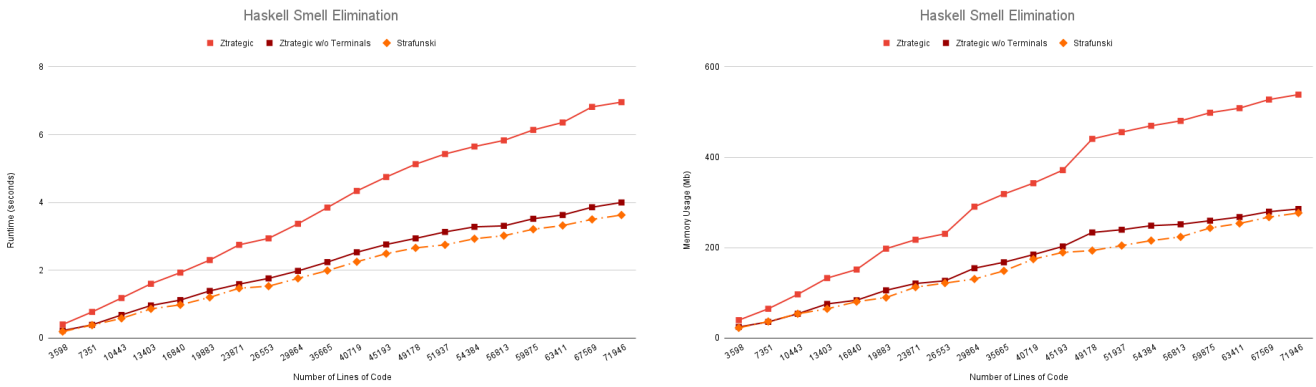

Figure 3. Haskell Smells elimination: Ztrategic versus Strafunski

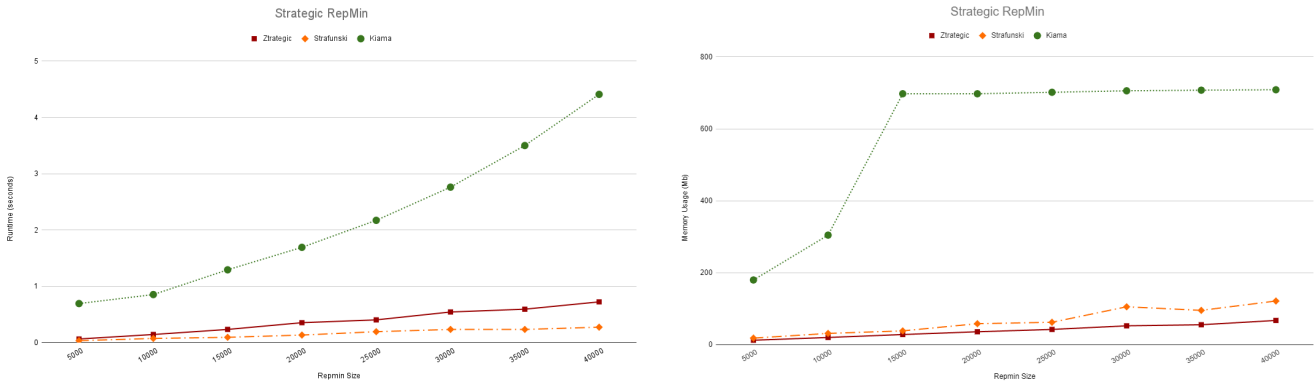

Figure 4. Strategic Repmin: Ztrategic versus Strafunski versus Kiama

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. The *repmin* problem can be easily implemented by two strategic functions: First, a Type Unifying strategy traverses the tree and computes its minimum value. Then, a Type Preserving strategy traverses again the tree and constructs the new tree, using the previously computed minimum. In Figure 4 we show the results of implementing these solutions using strategies in Ztrategic, Strafunski and Kiama. Here, *Repmin size* refers to the number of nodes the input binary tree contains. Again, Ztrategic behaves very similar to Strafunski and both outperform Kiama's implementation in speed and memory consumption.

4.3 Repmin as an Attribute Grammar

We also compare the performance of *repmin* when fully expressed as an AG. Actually, the Kiama implementation of *repmin* is part of the Kiama library. It is very similar to the Ztrategic version of *repmin* in [10], which we use here.

Figure 5 shows the results of comparing our memoized implementation of *repmin* in Ztrategic using AGs, with Kiama. To be able to compare the performance the strategic and AG solutions, we run them with exactly the same inputs.

In Figure 5 (left) we can see that the non-memoized version of Ztrategic increases its execution time exponentially and is much slower than the other versions. [3] However, when we zoom in to the behavior of the other implementations in Figure 5 right, we can notice that memoized Ztrategic outperforms Kiama. In fact, for a tree with 40000 nodes, the memoized Ztrategic AG runs in 0.3 seconds, while Kiama needs 0.78 seconds to perform the same task. When we compare these results with the strategic solution (shown in Figure 4), where Kiama is outperformed by Ztrategic and Strafunski, we see that, for 40000 nodes, the AG version of Kiama's implementation is 5.65 times faster than the strategic implementation for the same library. Comparatively, Ztrategic's non-memoized AG is 292 times slower than the strategic approach, while the non-memoized AG is 2.4 times faster than the strategic approach. The overall faster implementation is the strategic Strafunski's with a runtime of 0.27 seconds,

[3] All these runtime numbers, and the numbers used to produced all Figures, are available in a spreadsheet included in our replication package.

Table 1. Runtime of pretty printing *Let* Expressions for increasingly larger **let** inputs.

	let 1	let 2	let 3	let 4	let 5	let 6
AG	0	0,02	0,1	0,91	7,95	68,49
MemoAG	0,01	0,01	0,02	0,03	0,06	0,1
Kiama	0,72	5,79	292,69	–	–	–

thus 1.11 times faster than the memoized AG. The memoized AG is extremely competitive considering that is has added overload of handling zippers and a memoization table.

4.4 Let Optimization

We have implemented the **let** strategic AG algorithm in Kiama, as it also provides strategies and AGs.

Figure 6 shows the performance of optimizing several Let inputs, in terms of runtime and memory usage. Along the X axis, *Let size* refers to the number of nested **let** blocks contained in the input data to be optimized. The Ztrategic implementation once again vastly outperforms Kiama in both runtime and memory consumption, for any input size. Because the Kiama implementation shows a poor performance, compared to our memoized strategic AG, we also include Kiama's baseline execution: it just generates the input AST and prints it without performing any optimization. This task already takes more time than the optimization in the memoized Ztrategic. Kiama uses an advanced mechanism to combine strategies and AGs where ASTs are defined by reachability relations [27]. The mechanism to transform a tree into relations already induces a significant overhead in the Kiama baseline execution. This also drastically influences the memory usage of Kiama's solutions as we can see in the Kiama's solution for the *repmin* problem in Figure 4.

4.5 Multiple Layout Pretty Printing

We have expressed the large and complex optimal pretty printing AG, presented in [29], both in the Ztrategic and Kiama libraries. This AG specifies a multiple layout pretty printer that adapts the layout according to the available width of the page. Indeed, it defines a complex four traversal algorithm and it is one of the most complex AG available.

We used the Ztrategic and Kiama versions of this algorithm to pretty print **let** expressions from our running example. Table 1 presents the runtime in seconds of executing the same pretty printing with the non-memoized and memoized Ztrategic and Kiama AG embeddings. Here, the number of a *Let* expression refers to its nesting level, such that *Let n* will have *n* nested let declarations as well as 10 variable declarations for each nested declaration.

As expected the memoized Ztrategic solution is much faster than the non-memoized counterpart. The Kiama solution shows the poorest performance and is only able to pretty print the smaller three **let** expressions. This large AG defines many attributes and the tree/attribute relations mechanism used by Kiama to fully support strategic AGs do induce a considerable overhead.

5 Related Work

This paper is inspired by the work of Sloane who developed Kiama [26]: an embedding of strategic term re-writing and AGs in the Scala programming language. It is an embedding of strategic AGs, and relies on Scala mechanisms to navigate in the trees and memoizes attribute values on a global memoization table to avoid attribute recalculation. Our library relies on zippers and uses local memo tables to avoid attribute recalculation. Kiama uses reachability relations and the notion of attribute families to fully support strategic AGs. Attributes that depend on their context are defined parametrized by the tree they belong to. This allows sub-trees to be shared while maintaining the correctness of context-dependent attributes, since the same attribute has a different entry in the memoization table for the original tree and the one resulting from a re-write. On the other hand, context-independent attributes can use previously memorized values on modified trees. This approach is similar to our use of *applyTP* and *applyTP_unclean*, albeit finer-grained, as only context-dependent attributes are cleaned.

The extensible AG system Silver [31] has also been extended to support strategic term re-writing [15]. Strategic re-writing rules can use the attributes of a tree to reference contextual information during re-writing, much like we present in our work. While we use a functional embedding, Silver compiles its own Strategic AG specification into low code. The paper includes several practical application examples, namely the evaluation of λ-calculus, a regular expression matching via Brzozowski derivatives, and the normalization of for-loops. All these examples can be directly expressed in our setting. They also present an application to optimize translation of strategies. Because our techniques rely on shallow embeddings, where no data type is used to express strategies nor AGs, we are not able to express strategy optimizations, without relying on meta-programming techniques [25]. Nevertheless, our embeddings result in very simple and small libraries that are easier to extend and maintain, specially when compared to the complexity of extending and maintaining a full language system such as Silver.

RACR [5] is an embedding of Reference Attribute Grammars in Scheme, that allows graph re-writing and incremental attribute evaluation. A *dynamic attribute dependency graph* is constructed during evaluation in order to determine which attributes are affected by a re-writing and therefore should be re-evaluated. To keep our embedding simple we do not perform that kind of analysis, although incorporating it is an interesting line of possible future work.

JastAdd is a reference attribute grammar based system [9]. It supports most of AG extensions, namely reference and circular AGs [28]. It also supports tree re-writing, with re-write rules that can reference attributes. JastAdd, however, provides no support for strategic programming, that is, there is no mechanism to control how the re-write rules are applied.

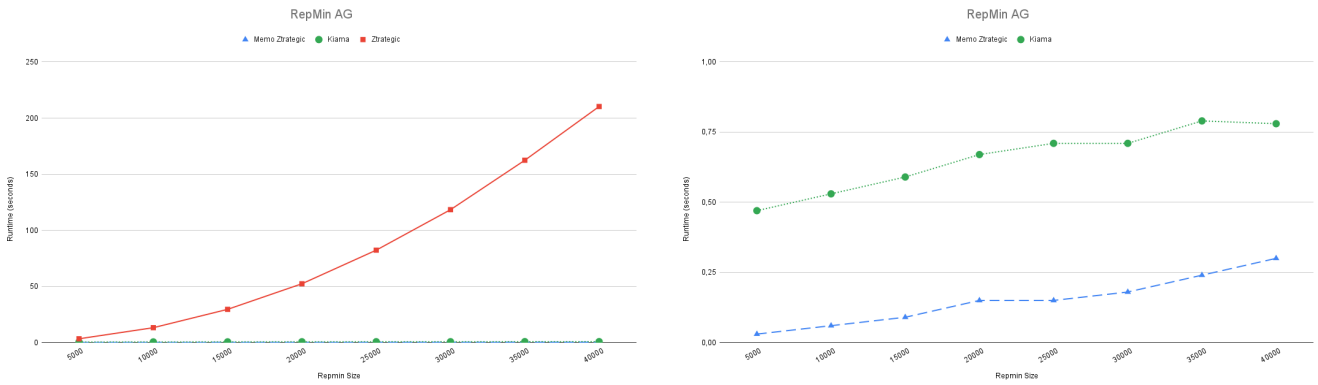

Figure 5. AG Repmin. Left: Ztrategic versus memoized Ztrategic and Kiama. Right: memoized Ztrategic and Kiama.

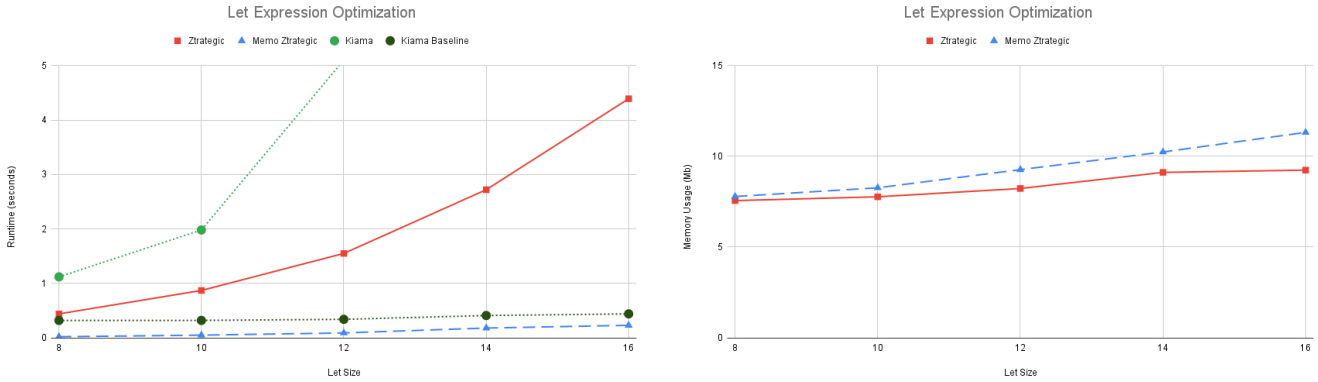

Figure 6. Let Optimization: Ztrategic versus Kiama

The zipper-based AG embedding we integrate in *Ztrategic* supports all modern AG extensions, including reference and circular AGs [10, 21]. Because strategies and AGs are first class citizens we can smoothly combine any of such extensions with strategic term re-writing.

In the context of strategic term re-writing, the *Ztrategic* library is inspired by Strafunski [17]. In fact, *Ztrategic* already provides almost all Strafunski functionality. There is, however, a key difference between these libraries: while Strafunski accesses the data structure directly, *Ztrategic* operates on zippers. As a consequence, we can easily access attributes from strategic functions and strategic functions from attribute equations.

6 Conclusion

This paper presented an embedding of strategic attribute grammars, which combine strategic term re-writing and the attribute grammar formalism. Both embeddings rely on memoized zippers: attribute values are memoized in the zipper data structure, thus avoiding their re-computation. Strategic zipper based functions access such memoized values.

We compared the performance of our embedding with state-of-the-art libraries Strafunski and Kiama. We have implemented in these three libraries several language engineering tasks, namely, a let optimizer, a code refactor, and an advanced pretty printing algorithm. Our results show that the embedding strategic term re-writing behaves very similar to Strafunski. Our results also show that our embedding of strategic AGs vastly outperforms Kiama's solutions.

Acknowledgements

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. The first author is also sponsored by FCT grant `2021.08184.BD`. We would like to thank Tony Sloane for his validation of the Kiama implementations used in Section 4.

Replication Packages

All the necessary resources to replicate this study, as well as the full set of results, are publicly available as embedded hyperlinks for the base ZippersAG, Strafunski, Ztrategic and Kiama libraries as well as Examples used in this paper.

A Ztrategic API

Strategy types

type *TP a* = **Zipper** *a* → *Maybe* (**Zipper** *a*)
type *TU m d* = (*forall a* . **Zipper** *a* → (*m d*, **Zipper** *a*))

Strategy Application

applyTP :: *TP a* → **Zipper** *a* → *Maybe* (**Zipper** *a*)
applyTP_unclean :: *TP a* → **Zipper** *a* → *Maybe* (**Zipper** *a*)
applyTU :: *TU m d* → **Zipper** *a* → (*m d*, **Zipper** *a*)
applyTU_unclean :: *TU m d* → **Zipper** *a* → (*m d*, **Zipper** *a*)

Primitive strategies

idTP :: *TP a*
constTU :: *d* → *TU m d*
failTP :: *TP a*
failTU :: *TU m d*
tryTP :: *TP a* → *TP a*
repeatTP :: *TP a* → *TP a*

Strategy Construction

monoTP :: (*a* → *Maybe b*) → *TP e*
monoTU :: (*a* → *m d*) → *TU m d*
monoTPZ :: (*a* → **Zipper** *e* → *Maybe* (**Zipper** *e*)) → *TP e*
monoTUZ :: (*a* → **Zipper** *e* → (*m d*, **Zipper** *e*)) → *TU m d*
adhocTP :: *TP e* → (*a* → *Maybe b*) → *TP e*
adhocTU :: *TU m d* → (*a* → *m d*) → *TU m d*
adhocTPZ :: *TP e* → (*a* → **Zipper** *e* → *Maybe* (**Zipper** *e*)) → *TP e*
adhocTUZ :: *TU m d* →
(*a* → **Zipper** *c* → (*m d*, **Zipper** *c*)) → *TU m d*

Composition / Choice

seqTP :: *TP a* → *TP a* → *TP a*
choiceTP :: *TP a* → *TP a* → *TP a*
seqTU :: *TU m d* → *TU m d* → *TU m d*
choiceTU :: *TU m d* → *TU m d* → *TU m d*

AG Combinators

(.$) :: **Zipper** *a* → *Int* → **Zipper** *a*
parent :: **Zipper** *a* → **Zipper** *a*
(.|) :: **Zipper** *a* → *Int* → *Bool*

Traversal Combinators

allTPright :: *TP a* → *TP a*
oneTPright :: *TP a* → *TP a*
allTUright :: *TU m d* → *TU m d*
allTPdown :: *TP a* → *TP a*
oneTPdown :: *TP a* → *TP a*
allTUdown :: *TU m d* → *TU m d*

Traversal Strategies

full_tdTP :: *TP a* → *TP a*
full_buTP :: *TP a* → *TP a*
once_tdTP :: *TP a* → *TP a*
once_buTP :: *TP a* → *TP a*
stop_tdTP :: *TP a* → *TP a*
stop_buTP :: *TP a* → *TP a*
innermost :: *TP a* → *TP a*
outermost :: *TP a* → *TP a*
full_tdTU :: *TU m d* → *TU m d*
full_buTU :: *TU m d* → *TU m d*
once_tdTU :: *TU m d* → *TU m d*
once_buTU :: *TU m d* → *TU m d*
stop_tdTU :: *TU m d* → *TU m d*
stop_buTU :: *TU m d* → *TU m d*
foldr1TU :: *TU m d* → **Zipper** *e* → (*d* → *d* → *d*) → *d*
foldl1TU :: *TU m d* → **Zipper** *e* → (*d* → *d* → *d*) → *d*
foldrTU :: *TU m d* → **Zipper** *e* → (*d* → *c* → *c*) → *c* → *c*
foldlTU :: *TU m d* → **Zipper** *e* → (*c* → *d* → *c*) → *c* → *c*

Memoized AG Combinators

(.@.) :: (**Zipper** *a* → (*r*, **Zipper** *a*))
→ **Zipper** *a* → (*r*, **Zipper** *a*)
atParent :: (**Zipper** *a* → (*r*, **Zipper** *a*))
→ **Zipper** *a* → (*r*, **Zipper** *a*)
atRight :: (**Zipper** *a* → (*r*, **Zipper** *a*))
→ **Zipper** *a* → (*r*, **Zipper** *a*)
atLeft :: (**Zipper** *a* → (*r*, **Zipper** *a*))
→ **Zipper** *a* → (*r*, **Zipper** *a*)
memo :: *attr* → *AGTree_m d m a* → *AGTree_m d m a*

Figure 7. Full Memoized Ztrategic API and AG Combinators

References

[1] Michael D. Adams. 2010. Scrap Your Zippers: A Generic Zipper for Heterogeneous Types. In *WGP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1863495.1863499

[2] José Bacelar Almeida, Alcino Cunha, Nuno Macedo, Hugo Pacheco, and José Proença. 2018. Teaching How to Program Using Automated Assessment and Functional Glossy Games (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 82 (July 2018), 17 pages. https://doi.org/10.1145/3236777

[3] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking Rewriting on Java. In *Term Rewriting and Applications*, Franz Baader (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–47.

[4] R. S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21 (January 1984), 239–250. https://doi.org/10.1007/BF00264249

[5] Christoff Bürger. 2015. Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (Pittsburgh, PA, USA) *(SLE 2015)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/2814251.2814257

[6] James R. Cordy. 2004. TXL - A Language for Programming Language Tools and Applications. *Electronic Notes in Theoretical Computer Science* 110 (2004), 3–31. https://doi.org/10.1016/j.entcs.2004.11.006 Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004).

[7] Jonathan Cowie. 2005. *Detecting Bad Smells in Haskell.* Technical Report. University of Kent, UK.

[8] Atze Dijkstra and S. Doaitse Swierstra. 2005. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming*, Varmo Vene and Tarmo Uustalu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–72.

[9] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[10] João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, and Marcos Viera. 2019. Memoized zipper-based attribute grammars and their higher order extension. *Sci. Comput. Program.* 173 (2019), 71–94. https://doi.org/10.1016/j.scico.2018.10.006

[11] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. 1992. Eli: A Complete, Flexible Compiler Construction System. *Commun. ACM* 35, 2 (Feb. 1992), 121–130. https://doi.org/10.1145/129630.129637

[12] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (Sept. 1997), 549–554.

[13] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[14] Donald E. Knuth. 1990. The genesis of attribute grammars. In *Attribute Grammars and their Applications*, P. Deransart and M. Jourdan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/3-540-53101-7_1

[15] Lucas Kramer and Eric Van Wyk. 2020. Strategic Tree Rewriting in Attribute Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 210–229. https://doi.org/10.1145/3426425.3426943

[16] Matthijs Kuiper and João Saraiva. 1998. Lrc - A Generator for Incremental Language-Oriented Tools. In *7th International Conference on Compiler Construction, CC/ETAPS'98 (LNCS, Vol. 1383)*, Kay Koskimies (Ed.). Springer-Verlag, 298–301.

[17] Ralf Lämmel and Joost Visser. 2002. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Languages*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–154.

[18] Sebastiaan P. Luttik and Eelco Visser. 1997. Specification of Rewriting Strategies. In *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications* (Amsterdam, The Netherlands) *(Algebraic'97)*. BCS Learning & Development Ltd., Swindon, GBR, 9.

[19] José Nuno Macedo, Marcos Viera, and João Saraiva. 2022. Zipping Strategies and Attribute Grammars. In *Functional and Logic Programming - 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10-12, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13215)*, Michael Hanus and Atsushi Igarashi (Eds.). Springer, 112–132. https://doi.org/10.1007/978-3-030-99461-7_7

[20] Pedro Martins, João Paulo Fernandes, and João Saraiva. 2013. Zipper-Based Attribute Grammars and Their Extensions. In *Programming Languages*, André Rauber Du Bois and Phil Trinder (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–149.

[21] Pedro Martins, João Paulo Fernandes, João Saraiva, Eric Van Wyk, and Anthony Sloane. 2016. Embedding Attribute Grammars and Their Extensions Using Functional Zippers. *Sci. Comput. Program.* 132, P1 (Dec. 2016), 2–28. https://doi.org/10.1016/j.scico.2016.03.005

[22] Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. 1995. LISA: A Tool for Automatic Language Implementation. *SIGPLAN Not.* 30, 4 (April 1995), 71–79. https://doi.org/10.1145/202176.202185

[23] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. *SIGPLAN Not.* 19, 5 (April 1984), 42–48. https://doi.org/10.1145/390011.808247

[24] João Saraiva. 2002. Component-Based Programming for Higher-Order Attribute Grammars. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*. 268–282. https://doi.org/10.1007/3-540-45821-2_17

[25] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[26] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. 2010. A Pure Object-Oriented Embedding of Attribute Grammars. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 205–219. https://doi.org/10.1016/j.entcs.2010.08.043

[27] Anthony M. Sloane, Matthew Roberts, and Leonard G. C. Hamey. 2014. Respect Your Parents: How Attribution and Rewriting Can Get Along. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 191–210.

[28] Emma Söderberg and Görel Hedin. 2013. Circular Higher-Order Reference Attribute Grammars. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 302–321.

[29] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. 1999. Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–206.

[30] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, Berlin, Heidelberg, 365–370.

[31] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2008. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science* 203, 2 (2008), 103–116. https://doi.org/10.1016/j.entcs.2008.03.047

[32] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*. Springer-Verlag, Berlin, Heidelberg, 357–362.

[33] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher Order Attribute Grammars. *SIGPLAN Not.* 24, 7 (jun 1989), 131–145. https://doi.org/10.1145/74818.74830

Received 2022-10-18; accepted 2022-11-15