Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Zipper-based embedding of strategic attribute grammars☆

José Nuno Macedo [a,*], Emanuel Rodrigues [a], Marcos Viera [b], João Saraiva [a]

[a] HASLab & INESC TEC, University of Minho, Braga, Portugal
[b] Universidad de la Republica, Montevideo, Uruguay

ARTICLE INFO

Keywords:
Strategic programming
Attribute grammars
Functional zippers

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. In addition, we increase the functionalities of strategic programming, enabling the definition of outwards traversals; i.e. outside the starting position.

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 hosted our memoized zipper-based embedding of strategic attribute grammars both in the Haskell and Python programming languages. Moreover, we benchmarked the libraries supporting both embedding against the state-of-the-art Haskell-based Strafunski and Scala-based Kiama libraries. The first results show that our Haskell Ztrategic library is very competitive against those two well established libraries.

1. Introduction

Strategic term re-writing (Luttik and Visser, 1997) and Attribute Grammars (AG) (Knuth, 1968) 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 (Gray et al., 1992; Reps and Teitelbaum, 1984; Kuiper and Saraiva, 1998; Mernik et al., 1995; Ekman and Hedin, 2007; Dijkstra and Swierstra, 2005; Van Wyk et al., 2008; van den Brand et al., 2001; Balland et al., 2007; Lämmel and Visser, 2002; Cordy, 2004; Sloane et al., 2010; Visser, 2001). 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 (Van Wyk et al., 2008) and the Kiama library (Sloane et al., 2010) do support both techniques.

More recently, a combined embedding of the two techniques has been proposed in Macedo et al. (2022). This embedding relies on a generic mechanism to navigate on both homogeneous and heterogeneous trees: generic zippers (Huet, 1997; Adams, 2010). 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 Martins et al. (2013) and Martins et al. (2016), for AGs, and in Macedo et al. (2022), 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 Fernandes et al. (2019), the simple zipper-based embedding of AGs (Martins et al., 2013, 2016) 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 Macedo et al. (2022), has exactly the same performance issues.

☆ Editor: Dr. Jacopo Soldani.
* Corresponding author.
E-mail addresses: jose.n.macedo@inesctec.pt (J.N. Macedo), jose.e.rodrigues@inesctec.pt (E. Rodrigues), mviera@fing.edu.uy (M. Viera), saraiva@di.uminho.pt (J. Saraiva).

https://doi.org/10.1016/j.jss.2024.111975
Received 9 March 2023; Received in revised form 30 October 2023; Accepted 19 January 2024
Available online 15 February 2024

In order to provide an efficient zipper-based embedding of strategic term re-writing and attribute grammars, that we call Strategic Attribute Grammars (SAGs), we implement zipper-based strategies on top of the memoized zipper-based embedding of AGs (Fernandes et al., 2019). 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 Macedo et al. (2022). The purpose of this paper is four-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 (Fernandes et al., 2019). Thus, we extend the Ztrategic library, developed in Macedo et al. (2022), with new combinators which work on trees where attribute values are memoized in the tree's nodes.
- Secondly, because the zipper data structure is the key ingredient for our embedding of strategic programming and attribute grammars, we host our embedding in the Python language via available libraries supporting algebraic data types and zippers. As a result, we provide an embedding of strategic AGs as a Python library.
- Thirdly, zippers are a generic and very flexible mechanism to navigate on trees. Thus, we use the power of zippers to express advanced navigation strategies which, for example, can express usual attribute propagation patterns offered in most AG-based systems. While classical AG systems have a fix, pre-defined notation for such patterns, via our strategic embedding we can express such patterns as first class citizens: new patterns can be defined via strategies. Moreover, 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 introduce the notion of non-navigable symbol which are not traversed by zippers. This does not limit the expressiveness of the strategic library, but does result in a considerable performance improvement of the implementations. While typical strategic traversal libraries navigate inwards from the starting position, we introduce outwards strategies that enable traversals outside the starting position.
- Fourthly, we perform a detailed study on the performance of the non-memoized implementation proposed in Macedo et al. (2022) 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 Haskell and Python implementations against the state of the art Strafunski (Lämmel and Visser, 2002) system - the Haskell incarnation of strategic term re-writing - and Kiama (Sloane et al., 2010) - the combined embedding of strategies and AGs in Scala.

Our preliminary results show that the Haskell embedding of strategic term re-writing behaves similarly to Strafunski. However, the embedding of SAGs vastly outperforms Kiama's solutions. Being a dynamic, interpreted language it is not surprising that our Python embedding presents the poorest performance.

This paper is an extended and revised version of the work presented at PEPM 2023 (Macedo et al., 2023). In this new paper we are extending the embeddings with new navigation strategies and attribute propagation patterns. We express Embedded Strategic Attribute Grammars in Python and we include the Python implementations in our detailed performance evaluation.

The rest of the paper is organized as follows: Section 2 introduces strategic term re-writing, attribute grammars, a combined embedding of SAGs, 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. In Section 4 we introduce some extensions to strategies and AGs that exploit the powerful navigation features of zippers. Section 5 presents the zipper-based embeddding of Stratetic AGs, where we discuss in detail the key differences from its Haskell counterpart. Section 6 compares the performance of our work with the Strafunski and Kiama libraries, and elaborates on the obtained results. Section 7 details the relevant state of the art on strategic programming and AGs. Section 8 concludes our work and below it we present links to the relevant libraries and to a replication package.

2. Zipping strategies and attribute grammars

In this section, we describe the zipper-based Strategic Attribute Grammars embedding introduced in Macedo et al. (2022) 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. We take this definition from previous work with strategies and attribute grammars in Macedo et al. (2022).

```
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. Fig. 1 presents such optimization rules directly taken from Kramer and Van Wyk (2020).

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

$$\begin{aligned}
add(e, const(0)) &\rightarrow e && (1)\\
add(const(0), e) &\rightarrow e && (2)\\
add(const(a), const(b)) &\rightarrow const(a+b) && (3)\\
sub(e1, e2) &\rightarrow add(e1, neg(e2)) && (4)\\
neg(neg(e)) &\rightarrow e && (5)\\
neg(const(a)) &\rightarrow const(-a) && (6)\\
var(id) \mid (id, just(e)) \in env &\rightarrow e && (7)
\end{aligned}$$

Fig. 1. Optimization rules.

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 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 cases. Even though the data type we consider is heterogeneous, we do not need special considerations for this as the default $failTP$ case will handle all remaining data types. 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 (Knuth, 1990). 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. 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 Fig. 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 Fig. 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 (Martins et al., 2013) 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 Env
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
```

This attribute returns a value of type Env, which is a list of names with their associated nesting level and definition. Thus, type Env is defined as the following type synonym:

```
type Env = [(String, Int, Maybe Exp)]
```

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:

[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 $Haskell$ (Sheard and Jones, 2002).

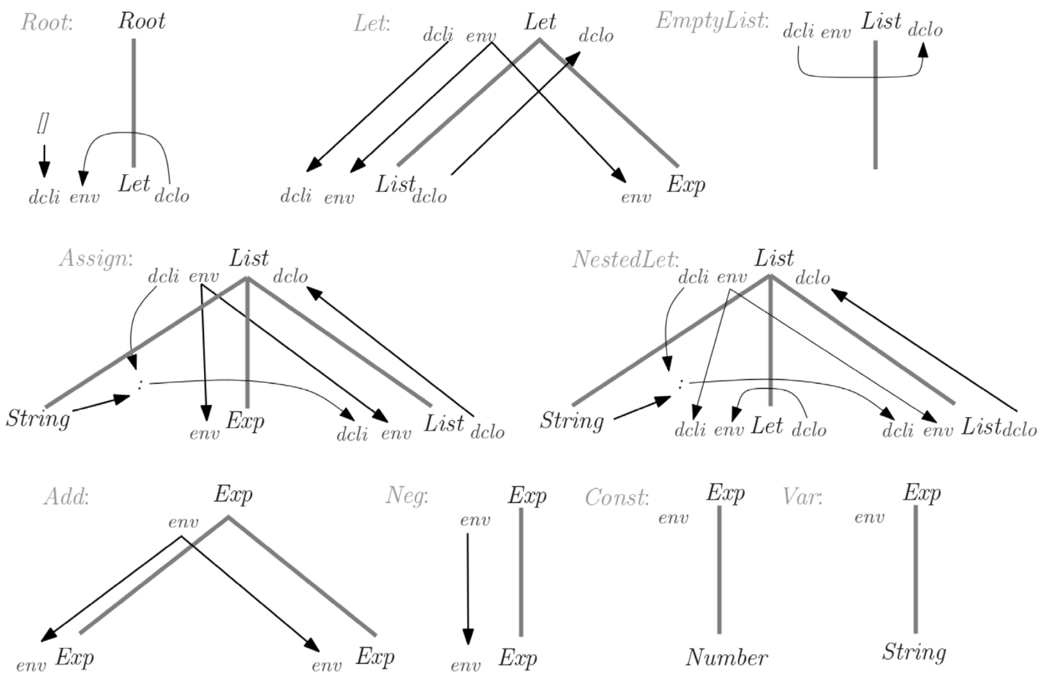


Fig. 2. Attribute grammar specifying the scope rules of *Let*.

```
env :: AGTree Env
env t = case (constructor t) of
          Root_P  → dclo t
          Let_Let → dclo t
          _       → env (parent t)
```

Let us define now the equations of inherited attribute $dcli$. As shown in Fig. 2, the initial list of declarations $dcli$ at the $Root$ of the tree is the empty list, since the outermost block is context-free. Attribute $dcli$ of Let occurs again in $NestedLet$. In this case, $dcli$ is defined as the inherited attribute of the parent. The $List$ nonterminal also has $dcli$ as inherited attribute. As we can see in Fig. 2, $List$ occurs in productions Let, $Assign$ and $NestedLet$. Each of these occurrences is defined by different equations. For example, in $Assign$ the attribute $dcli$ of $List$ is defined by adding the defined name ($String$) to the $dcli$ of the parent (Fig. 2 omits the use of lev and Exp). Thus, the zipper-based function $dcli$ directly follows our visual notation and it is written as follows:

```
dcli :: Zipper Root → Env
dcli ag = case (constructor ag) of
  Let_Let → case (constructor (parent ag)) of
            Root_P         → []
            NestedLet_List → env (parent ag)
  _       → case (constructor (parent ag)) of
            Let_Let        → dcli (parent ag)
            Assign_List    → (Lexeme_Name (parent ag), lev (parent ag)
                             , Lexeme_Exp (parent ag))
                              : (dcli (parent ag))
            NestedLet_List → (Lexeme_Name (parent ag), lev (parent ag),
                             Nothing)
                              : (dcli (parent ag))
```

where $Lexeme_{Name}$ and $Lexeme_{Exp}$ implement the so-called AG syntactic Refs. Reps and Teitelbaum (1984): the use of non-terminals in the AG equations. We omit here their definition because they directly follow from the language data types, and, they can be generated via Template Haskell (Sheard and Jones, 2002). In this example we collect an environment with the expressions defined by each variable. Thus, in the optimization rule we will have to consider only the variables bound to constant expressions. We could perform a more aggressive optimization if we define an evaluation attribute and store the evaluated expressions in the environment.

The definition of inherited attribute lev is straightforward: it starts with value 0 at the root which is incremented when passed to a nested Let. In all other cases lev is just a copy of its parent definition. Its definition looks as follows:

```
lev :: Zipper Root → Int
lev ag = case (constructor ag) of
           Let_Let → case (constructor (parent ag)) of
                       NestedLet_List → (lev (parent ag)) + 1
                       Root_P → 0
           _       → lev (parent ag)
```

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. Next, we present the definition of $expand$:

```
expand :: (Name, Int) → Env → Maybe Exp
expand (i, l) e = case level of
  ((nE, lE, Just (Const c)) : _) → Just (Const c)
  _                              → Nothing
```

```
where vars = filter (λ(nE, lE, _) → nE ≡ i ∧ lE ≤ l) e
   level = sortBy (λ(nE1, lE1, _) (nE2, lE2, _) → compare lE2 lE1) vars
```

While the code might be a bit daunting for non-*Haskell* programmers, the concept is simple: we remove everything from the environment except names that are equal to the variable i we wish to expand. Then, we sort by nesting level. Lastly, we take the first of these results, and if it is a constant value, we return it; else we signal failure.

In fact, we take care to only expand variables that will be replaced by constants. If we expand any variable into the expression it is defined as, and if the result contains other variables, we risk introducing a variable from a different block into the current block, which is an error as their value might have been re-defined in the current scope. We solve this by only replacing variables by constants. Alternatively, we could use an *eval* attribute to evaluate expressions before storing them in the environment, which would solve this issue, but would make the definition of *expand* even more complicated. Note that, even if we only expand specifically when the definition of a variable is just a constant, the other 6 optimization rules will make variable definitions tend towards a single constant, making this solution elegant and effective.

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 (Macedo et al., 2022). However, this embedding of AGs has a severe limitation since when decorating the tree, it recomputes 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 (Fernandes et al., 2019) 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) (Vogt et al., 1989) 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 Saraiva (2002). The zipper based embedding of AGs supports this extension (Martins et al., 2016). 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 (Söderberg and Hedin, 2013).

```
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 $optHAG = circ$. In fact

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 cannot 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 (Fernandes et al., 2019). 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 re-defined 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))
```

Note that we opt to memoize all attributes in this example. However, it is not strictly necessary to do so — we can still define simple attributes and use them in conjunction with memoized attributes, and for this we do not use the **memo** function. If an attribute value would be changed frequently, its value would not be reused properly and thus it is preferable to define it as a regular, non-memoized attribute.

3. Combining attribute memoization with zippers

As shown in Fernandes et al. (2019) the attribute evaluation of memoized zipper-based AGs is much faster than the non-memoized one. As we will show in Section 6, the combined embedding proposed in Macedo et al. (2022) 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. Here, *fmap* just ensures that *buildMemoTreeExp* is applied correctly to the result of *expand*.

```
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 (Adams, 2010). This library includes the **toZipper**:: $Data\ a \Rightarrow a \rightarrow$ **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** $a \rightarrow Maybe$ (**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** :: $Typeable\ b \Rightarrow$ **Zipper** $a \rightarrow Maybe\ b$ and **setHole** :: $Typeable\ a \Rightarrow a \rightarrow$ **Zipper** $b \rightarrow$ **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 behavior.

```
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 behavior. 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 Macedo et al. (2022), 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 Macedo et al. (2022), 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 version of $Ztrategic$ is available 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 6.

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 behavior, 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 behavior of not skipping any nodes, for the Let datatype:

```
instance StrategicData Let
```

We could optimize this by instead of defining our own behavior 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 6. We also require this change to define memoization in a strategic setting.

4. New directions

Due to the fact that our tools are embedded as libraries in a programming language, our strategic definitions are first class citizens and can easily be extended with new useful and reusable traversal schemes.

In this section we introduce three new strategic combinators, which exploit the possibility that the zipper gives us to navigate in any direction of the tree.

4.1. Outwards traversals

While typical strategic traversal libraries navigate inwards from the starting position, next we present strategies that also enables outwards traversals, i.e. outside the starting position. This is enabled by previously mentioned functional Zippers that offers a powerful generic mechanism for navigation on heterogeneous data structures. In fact, Zippers support upwards navigation, which is not possible on the typical recursion traversal. As such, we introduce three new traversal strategies[2]:

- **atRoot** — Apply a given strategy at the root of a data structure that is accessed from any of its nodes. This guarantees a traversal of the whole data structure.
- **full_uptdTP** — Apply a given strategy to all nodes in the path from the root of the data structure to the current node, in a top-down fashion. Similar strategies exist for bottom-up traversal, and for once-only application, for both Type-Preserving and Type-Unifying strategies.
- **full_tdTPupwards** — Navigate to the root and then apply a given strategy to the data structure, but not traversing from the starting node downwards. In fact, this is the complementary traversal of *full_tdTP*, such that $full\_tdTPUp\, p \cup full\_tdTPUp\, p \equiv atRoot$.

[2] The API of Ztrategic in Appendix includes all these new combinators.

These outwards strategies provide powerful abstractions to express transformations that rely on context information. Let us recall optimization rule 7. Fig. 2 presented the visual definition of the AG collecting the environment (attribute *env*) needed to expand an identifier, i.e. to replace it by its definition. In the straightforward AG definition we collected a global environment with all definitions occurring in the input program, which were inherited (as attribute *env*) by all nodes in the AST. Moreover, we used attribute *lev* to distinguish declarations of the same identifier at different nested levels (recall, for example, the definitions of type *Env* and function *expand*, presented previously).

By using our new outward traversal schemes, however, we can avoid the computation of such a global environment. By using outwards strategies we can express rule 7 using the following algorithm:

- First, we compute/synthesize the local environment of each *Let* block.
- Then, we use an outward strategy to search for the definition of an identifier in the block we need to expand it.
  - If the identifier is found in the local environment of the (same) *Let* block, then we use its definition to expand it.
  - Otherwise, the outwards strategy continues navigating upwards towards the root of the AST, while looking for the identifier in the local environments of all *Let* blocks it is traversing. As soon as it finds the definition of the desired identifier it stops traversing upwards since it found the closest definition of that identifier.

Let us now express this algorithm as an strategic attribute grammar. To synthesize the local environment of a *Let* block, we define the new type of the environment:

```
type LocalEnv = [(String, Maybe Exp)]
```

and we define a synthesized attribute *localEnv* that adds local identifier definitions to the environment.

```
localEnv :: Zipper Root → LocalEnv
localEnv ag = case (constructor ag) of
  Let_Let           → localEnv (ag.$1)
  NestedLet_List    → (Lexeme_Name ag, Nothing) : localEnv (ag.$3)
  Assign_List       → (Lexeme_Name ag, Lexeme_Exp ag) : localEnv (ag.$3)
  EmptyList_List    → [ ]
```

Now, instead of using attributes to pass the (global) environment to where it is needed (as we did in Section 2, namely when expanding identifiers), we use our outwards strategies to navigate upwards in the tree looking for such definitions.

Before we define the traversal scheme, we implement a function that searches for the name of an identifier in the synthesized attribute *localEnv* of its *Let* block.

```
definesVar :: String → Let → Zipper Root → Maybe [Exp]
definesVar var _ z = case lookup var (localEnv z) of
                       Just (Just (Const c)) → Just [Const c]
                       Just _                → Just [ ]
                       _                     → Nothing
```

This worker function requires a type unifying strategy since it (may) return the expression that is the expansion of the given identifier (its first argument).[3]

In order to express the algorithm that looks for the identifier in all *Let* blocks while upwards traversing the AST towards the root, we use the *once upwards bottom up type unifying* (*once_upbuTU*) strategy: *once* because we want to stop in the closest definition of the identifier and

[3] As we did in Section 2, to simplify the presentation we expand identifiers that are defined as constants only.

bottom up since we look for it going upwards from the current block towards the root.

```
lookupVar :: String → Zipper Root → Maybe [Exp]
lookupVar var z = applyTU (once_upbuTU step) z
          where step = failTU `adhocTUZ` (definesVar var)
```

This function applies $definesVar$ to check if the underlying block defines var, and if so, it returns its value and stops the traversal. If nothing is found, the strategy keeps on traversing upwards until a value is found or it reaches the $Root$.

Now, the outwards strategy $expC'$ that implements rule 7 is expressed as follows:

```
expC' :: Exp → Zipper Root → Maybe Exp
expC' (Var v) z = case lookupVar v z of
                       Just [e] → Just e
                       _        → Nothing
expC' _ z       = Nothing
```

This strategic attribute grammar searches for variable definitions in the environment one block at a time, without necessarily computing the entire environment. On the contrary, the AG-based definition of $expC$ (shown in Section 2) relies on the $expand$ function that searches the entire environment for the definition of a given identifier at a given level (or higher). By performing lookup operations in smaller environments this new version of rule 7 (i.e. $expC'$) is also more efficient than $expC$, as the benchmarks we present in Section 6.4.1 show.

4.2. Attribute propagation patterns

Attribute grammar systems often use a special notation (Gray et al., 1992; Reps and Teitelbaum, 1984; Dijkstra and Swierstra, 2005) to write attribute propagation patterns in order to avoid polluting the AG specification with too many copy rules: equations that just copy attribute values upwards/downwards the AST. The typical example of a propagation pattern is the remote access to an attribute value, which avoids the need to pass such attributes downwards via copy rules. Usually, AG based systems have a special syntactic notation to specify pre-defined attribute propagation patterns they support. Thus, the set of supported propagation patterns is impossible to extend without a full update of the AG system itself.

In our zipper-based embedding of AG we see attribute propagation patterns as first class citizens. Via zippers we can easily express new propagation patterns and reuse predefined ones. For example, up until now, we resort to a rather straightforward implementation of AGs where inherited and synthesized attributes are computed by operating on the immediate children and parent at any given point. We can extend this with access to remote attributes, i.e. attributes that are defined somewhere else on the tree.

Next, we present a zipper function $inherit$ that navigates upwards until a given predicate is true. When the predicate is true, the function applies a given (zipper) function, to the current focus, and stops the navigation.

```
inherit :: Data n ⇒ (n → Bool) → (Zipper a → b) → Zipper a → b
inherit p f z = if query (mkQ False p) z then f z else (inherit p f) .^ z
```

This function can be used to access a remote attribute in the AG specification. Let us consider the Let AG of the running example, where the inherited attribute env is completely specified via copy-rules in order to pass the global environment downwards to Var leaves. Recall the definition of $expC$ where an id is expanded to its definition. Next, we specify such attribute via the $inherit$ pattern as follows:

```
env' :: AGTree Env
env' t = inherit isNest dclo t
   where isNest (Let _ _) = True
         isNest _         = False
```

This zipper definition navigates upwards – function $inherit$ – until a Let node is reached, where it specifies that the env attribute is the value of $dclo$ at that node.

We also define the (.^^) combinator to compute an attribute at the root of a given tree. This could have been used for example to inherit an environment of global variables.

5. Strategies and attribute grammars in Python

The embedding of strategic programming and AGs we have presented does not rely on any advanced mechanism of the Haskell programming language. The key ingredient for the proposed embeddings is the zipper data structure. In fact, our simple embedding of strategic AGs is language-agnostic, and consequently can be expressed in any language providing functional zippers and algebraic data types.

In this section we show that, in fact, these are the building blocks of our embedding by re-implementing it in the Python language, reusing existing zippers and ADT libraries. As result, we offer a Python embedding of strategic AGs.

To embed strategic AGs in Python we follow a similar approach to the one described in Section 2, namely:

- We reuse a zipper library available in the host language, in this case the Python library named Zipper,[4] which we integrate with a Python library providing support for algebraic data types.[5] The zipper library is the building block that supports the definition of functions to model the attribute grammars and strategic programming.
- In order to avoid attribute re-calculation and provide a proper embedding of AGs, we use memoization to store attribute values in a memoization table.

Although the embedding of Haskell and Python rely on the same zipper data structure, there are major differences between the two host languages that we needed to consider:

- Firstly, the arguably less elegant syntax of Python does not allow to define AG combinators using non-alphanumeric identifiers in infix notation, which in Haskell allowed to have functions/combinators very close to the AG notation. Moreover, the Python ADT library offers a poor pattern matching mechanism when compared to Haskell, which makes the definition of type-specific worker functions less elegant.
- Secondly, we naturally handle failure in Haskell using the $MonadPlus$ class, which allows for easy handling and propagation of failure; with no such mechanism available in Python, we resort to throwing and catching a *StrategicError* exception.
- Thirdly, while in Haskell we used a memoization table that was strongly typed inside a new data type built specifically for memoization, in Python we directly store attribute values in the original tree nodes as side effects with no concerns for type checking.

Let us return to our **let** optimization program of Section 2. Next, we define Python version of the type specific function $expr$, that implements the optimization rules 1 to 6. The Python library for algebraic data types includes a $match$ function that allows the use of pattern matching. Unfortunately, this function does not allow to express a partial matching, so we always need to specify a behavior for every case. The Python exception *StrategicError* is used to signal that the strategy has to fail in that node.

[4] https://pypi.org/project/zipper/.
[5] https://pypi.org/project/algebraic-data-types/.

```
def expr(exp):
    x = exp.match(
        add=lambda x, y: y if (x == exp.CONST(0)) else
                         x if (y == exp.CONST(0)) else
                         Exp.CONST(x.const() + y.const()) if
                         (lambda a, b: x == exp.CONST() and
                                       y == exp.CONST()) else
                         st.StrategicError,
        sub=lambda x, y: Exp.ADD(x, Exp.NEG(y)),
        neg=lambda x: x.neg() if (lambda a: x == Exp.NEG()) else
                      exp.CONST(-x.neg())
                              if (lambda b: x == Exp.CONST()) else
                      st.StrategicError,
        var=lambda x: st.StrategicError,
        const=lambda x: st.StrategicError
    )
    if x is st.StrategicError:
        raise x
    else:
        return x
```

Now, we define optimization rule 7, that needs to access attributes *lev* and *env* to expand an identifier introduced in a *var* node. This is implemented in function *expC*, when a *var* node is matched. The implementation of the memoized AG-based Python functions *lev* and *env* will be discussed in Section 5.1.

```
def expC(exp, z):
    x = exp.match(
        add=lambda x, y: st.StrategicError,
        sub=lambda x, y: st.StrategicError,
        neg=lambda x: st.StrategicError,
        var=lambda x: expand((x, lev(z)), env(z)),
        const=lambda x: st.StrategicError
    )
    if x is st.StrategicError:
        raise x
    else:
        return x
```

Having defined the worker functions that model all optimization rules, we express the full **let** optimization function *opt* by combining *expr* and *expC* using the Python *adhocTP* and *adhocTPZ* strategic combinators.[6] Such *step* function is applied using the Python *innermost* strategy while traversing the tree. In Python we do not need to define the *apply* function to apply strategies, and actually this is the main difference to the Haskell version of *opt′* shown in Section 2.

```
def opt(z):
    def exp1(y):
        return st.adhocTPZ(st.failTP, expC, y)

    def step(x):
        return st.adhocTP(exp1, expr, x)

    return st.innermost(step, z).node()
```

5.1. Attribute memoization in Python

Similarly to the approach we tackled in Haskell, we optimize the Python implementation of attribute grammars with memoization. For this, we initially explored the already available solutions for memoization in Python. The `functools`[7] library contains several tools for higher-order functions, including memoization annotations. In fact, we achieved linear performance improvements of around 50% with very minimal effort, namely adding a *@cache* annotation before each attribute definition. While this is interesting by itself, we expected performance improvements to grow with input size, that is, the bigger the input, the more impactful the optimization should be. By using this library, we also have no control over how the memoization is achieved, meaning the discussion in Section 3.2 would not translate to this new implementation.

Due to this, we implement memoization in Python using the same approach we use in Haskell, we defined a memotable and associated it with each node and then re-defined the attributes to use memoization.

Next, we show the implementation of the *dclo* attribute in Python.

```
def dclo(x):
    return memo(Att_DCLO(), dcloAux, x)


def dcloAux(x):
    match constructorM(x):
        case Constructor.CRoot:
            return at(dclo, x.z_dollar(1))
        case Constructor.CLet:
            return at(dclo, x.z_dollar(1))
        case Constructor.CNestedLet:
            return at(dclo, x.z_dollar(3))
        case Constructor.CAssign:
            return at(dclo, x.z_dollar(3))
        case Constructor.CEmpty:
            return dcli(x)
```

In this example, the *at* function corresponds to .@. combinator in Haskell and the memoization takes place with the **memo** function.

Using memotables we achieved performance improvements even better than the ones achieved with the `functools` library and, in contrast, our improvements grow with the input size. Despite that, as we will discuss in the next section, our results show that the Haskell library is more efficient, with even our memoized results in Python being slower than the non-memoized results in Haskell.

6. Performance

In this section, we compare the performance of our zipper-based embedding of Strategic AGs in Haskell and Python with state-of-the-art libraries Strafunski and Kiama. In terms of expressiveness, both our zipper-based embeddings are 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 8. All tests were run 10 times and averaged in a ThinkPad 13 (2nd Gen, Intel i7-7500U (4) 3.500 GHz) laptop with 8 Gb RAM and EndeavourOS Linux x86 64 bits.

[6] The definition of the strategic library in Python follows a similar naming scheme to the Haskell version.

[7] https://docs.python.org/3/library/functools.html.

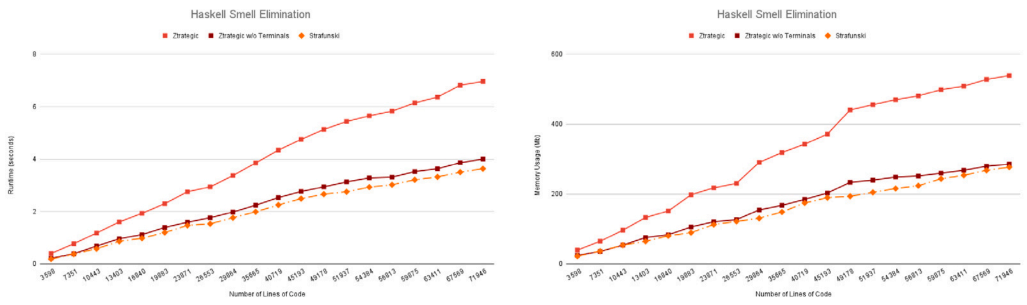


Fig. 3. Haskell Smells elimination: Ztrategic versus Strafunski.

6.1. *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. To parse Haskell code we reused the Haskell front-end available as one of its libraries. Thus, in this benchmark we consider the Ztrategic and Strafunski libraries, only. The two developed tools detect and eliminate all Haskell smells as reported in Cowie (2005).

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 Almeida et al. (2018). In these projects there are 1139 Haskell files totaling 82 124 lines of code, of which exactly 1000 files were syntactically correct.[8] Both Ztrategic and Strafunski smell eliminators detected and eliminated 850 code smells in those files. Fig. 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.

6.2. *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 (1984). The goal of this program 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. 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 Fig. 4 we show the results of implementing these solutions using strategies in Haskell and Python Ztrategic libraries, Strafunski and Kiama. Here, *Repmin size* refers to the number of nodes the input binary tree contains. Again, the Haskell Ztrategic embedding behaves very similar to Strafunski and both outperform the Kiama and Python implementations in terms of runtime. The memory consumption of all implementations is similar, being Kiama the clear poor exception.

6.3. *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 Haskell Ztrategic version of *repmin* in Fernandes et al. (2019), which we use here.

Fig. 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 Fig. 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.[9] However, when we zoom in to the behavior of the other implementations in Fig. 5 right, we can notice that memoized Ztrategic outperforms Kiama. In fact, for a tree with 40 000 nodes, the memoized Ztrategic AG runs in 0.3 s, while Kiama needs 0.78 s to perform the same task. When we compare these results with the strategic solution (shown in Fig. 4), where Kiama is outperformed by Ztrategic and Strafunski, we see that, for 40 000 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 s, 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.

6.4. *Let optimization*

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

Figs. 6–8 show 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. As clearly shown in Fig. 6 left, both non-memoized and memoized Python embeddings present the poorest performance. On the contrary, the Haskell Ztrategic implementations once again vastly outperform 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, in Figs. 6 (right) and 7 we also include Kiama's baseline execution: it just generates the input AST and prints it without performing any optimization. Fig. 7 removes the Python values so that it is easier to compare Ztrategic and Kiama. Kiama's baseline task already takes more time than the optimization

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

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

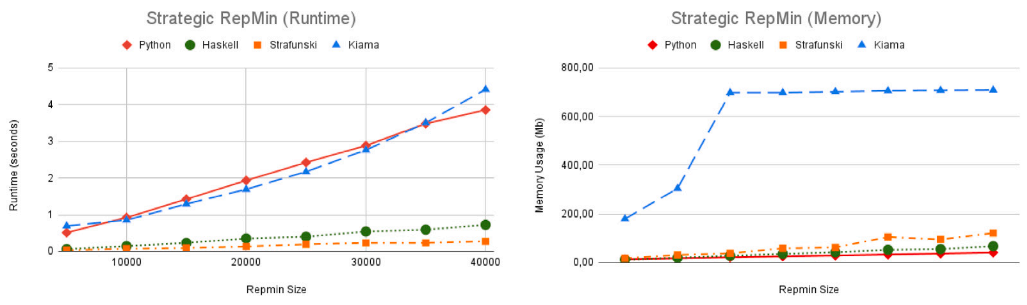


Fig. 4. Strategic Repmin: Ztrategic versus Strafunski versus Kiama versus Python.

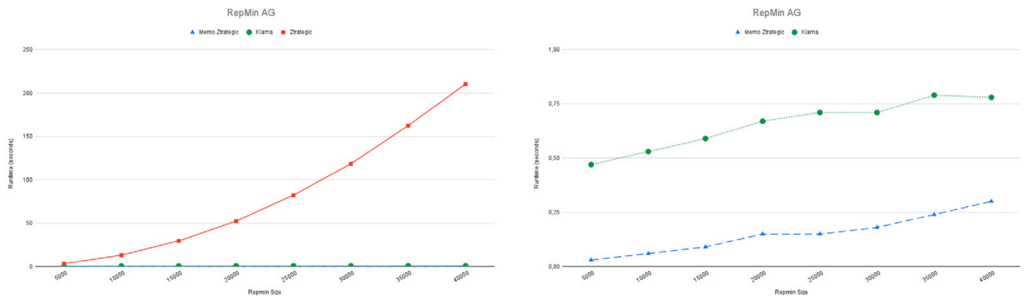


Fig. 5. Runtime of the AG Repmin. Left: Ztrategic versus memoized Ztrategic and Kiama. Right: memoized Ztrategic and Kiama.

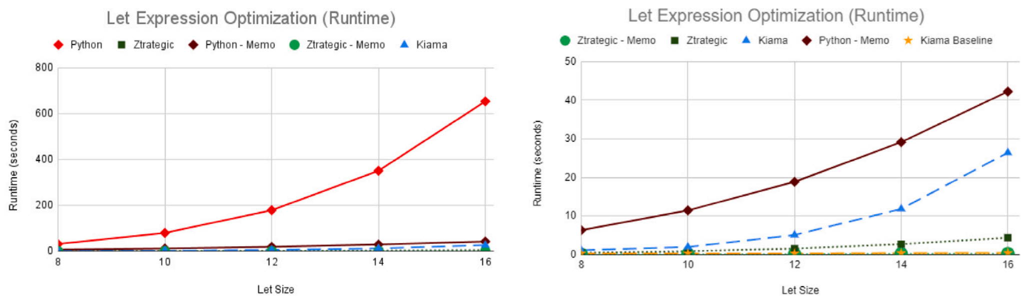


Fig. 6. Let Optimization — Runtime: Ztrategic versus Kiama versus Python.

in the memoized Ztrategic. Kiama uses an advanced mechanism to combine strategies and AGs where ASTs are defined by reachability relations (Sloane et al., 2014). 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 also see in the Kiama's solution for the *repmin* problem in Fig. 4.

6.4.1. Local vs. global attributes in Ztrategic

In Section 4.1, we suggest that our implementation of a *Let* program optimization can use a local attribute *localEnv* instead of a global attribute *env*, which we expect to be more efficient. We assume so because *localEnv*, in the best case, only computes declarations of a block, instead of every declared variable as is the case with *env*.

We present the performance of the *Let* optimization using both *env* and *localEnv* in Fig. 9. The global attribute implementation takes between 27% and 64% more runtime when compared to the local attribute implementation. It should be noticed that the benchmarks used to gather this data are *Let* blocks which benefit greatly from the way *localEnv* is implemented, by having variables declared close to their usage location. In terms of memory consumption, *env* also performs worse, however there is less discrepancy of values here when compared to the runtime values.

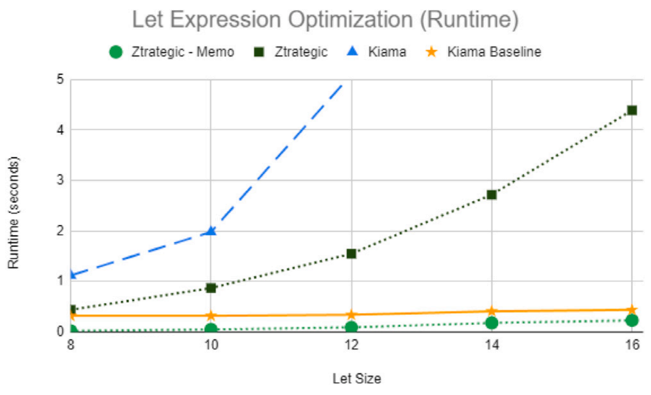
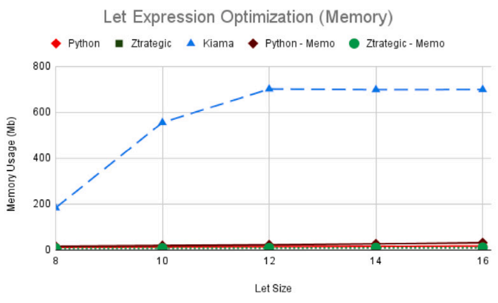


Fig. 7. Let Optimization — Runtime: Ztrategic versus Kiama.

6.5. Multiple layout pretty printing

We have expressed the large and complex optimal pretty printing AG, presented in Swierstra et al. (1999), both in the Ztrategic and Kiama libraries. This AG specifies a multiple layout pretty printer that



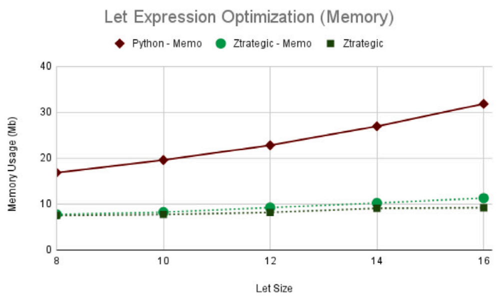
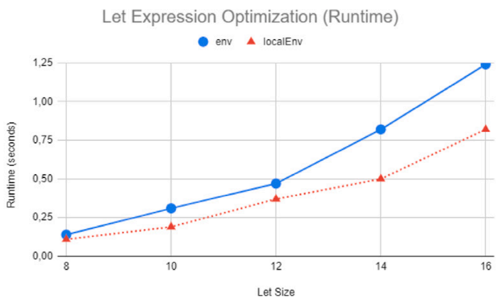


Fig. 8. Let Optimization — Memory: Ztrategic versus Kiama versus Python.



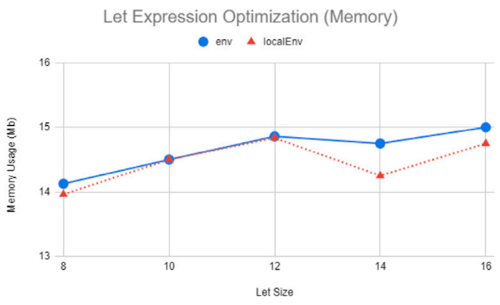


Fig. 9. Let Optimization — Ztrategic: using AG-based env versus using outwards-based strategy $localEnv$.

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	–	–	–

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.

7. Related work

This paper is inspired by the work of Sloane who developed Kiama (Sloane et al., 2010): 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 (Van Wyk et al., 2008) has also been extended to support strategic term re-writing (Kramer and Van Wyk, 2020). 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 (Sheard and Jones, 2002). 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. Silver translates strategies into equations for higher-order attributes, while our strategies are just data traversal mechanisms built on top of zippers, and its only relation to AGs is that our AGs also depend on zippers and therefore we can combine them.

RACR (Bürger, 2015) 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 (Ekman and Hedin, 2007). It supports most of AG extensions, namely reference and circular AGs (Söderberg and Hedin, 2013). 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. The zipper-based AG embedding we integrate in *Ztrategic* supports all modern AG extensions, including reference and circular AGs (Martins et al., 2016; Fernandes et al., 2019). 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 (Lämmel and Visser, 2002). 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. Moreover, we can traverse not only into a data structure, but also outwards, that is, traverse regressing into previously-visited nodes. This is possible also due to the powerful abstraction provided by the zippers data structure. We show in Section 4.1 these outward traversals and elaborate on how to implement rule 7 of our running example using them.

8. 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. The zipper data structure is the key ingredient of such an embedding, and we have expressed it as libraries in the Haskell and Python programming languages.

We introduced new strategies that navigate upwards in the data structure. We provide an usage example by defining an attribute for the local environment of each block and by using the new *once_upbuTU* strategy with it to search for an identifier in the current block or by traversing upwards the AST. This approach is around 30% faster when compared to the full AG-based definition of the full environment.

We compared the performance of both our embeddings with state-of-the-art libraries Strafunski and Kiama. We have implemented in these four libraries several language engineering tasks, namely, a let optimizer, a code refactor, and an advanced pretty printing algorithm. Our results show that the Haskell embedding strategic term re-writing behaves very similar to Strafunski. Due to the dynamic nature of the Python language, our Python embedding produces the slowest implementations. On the contrary, our Haskell embedding of SAGs vastly outperforms Kiama's solutions.

Replication packages

All the necessary resources to replicate this study, as well as the full set of results, are publicly available for the base ZippersAG,[10] Strafunski,[11] Ztrategic[12] and Kiama[13] libraries as well as Examples[14] used in this paper.

CRediT authorship contribution statement

José Nuno Macedo: Conceptualization, Software, Writing – original draft. **Emanuel Rodrigues:** Software, Investigation. **Marcos Viera:** Writing – review & editing, Supervision. **João Saraiva:** Writing – review & editing, Supervision.

[10] https://hackage.haskell.org/package/ZipperAG-0.9.
[11] https://hackage.haskell.org/package/Strafunski-StrategyLib.
[12] https://bitbucket.org/zenunomacedo/ztrategic/.
[13] https://github.com/inkytonik/kiama.
[14] https://tinyurl.com/VSISLE2023tools.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The paper contains a replication package with links to all material.

Acknowledgments

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 sponsored by FCT grant `2021.08184.BD`. We would like to thank Tony Sloane for his validation of the Kiama implementations used in Section 6.

Appendix. Ztrategic API

Strategy types

$\mathbf{type}\ TP\ a = \mathbf{Zipper}\ a \to Maybe\ (\mathbf{Zipper}\ a)$
$\mathbf{type}\ TU\ m\ d = (forall\ a\ .\ \mathbf{Zipper}\ a \to (m\ d, \mathbf{Zipper}\ a))$

Strategy Application

$applyTP :: TP\ a \to \mathbf{Zipper}\ a \to Maybe\ (\mathbf{Zipper}\ a)$
$applyTP\_unclean :: TP\ a \to \mathbf{Zipper}\ a \to Maybe\ (\mathbf{Zipper}\ a)$
$applyTU :: TU\ m\ d \to \mathbf{Zipper}\ a \to (m\ d, \mathbf{Zipper}\ a)$
$applyTU\_unclean :: TU\ m\ d \to \mathbf{Zipper}\ a \to (m\ d, \mathbf{Zipper}\ a)$

Primitive strategies

$idTP\ \ :: TP\ a$
$constTU :: d \to TU\ m\ d$
$failTP\ \ :: TP\ a$
$failTU\ \ :: TU\ m\ d$
$tryTP\ \ :: TP\ a \to TP\ a$
$repeatTP :: TP\ a \to TP\ a$

Strategy Construction

$monoTP\ \ :: (a \to Maybe\ b) \to TP\ e$
$monoTU\ \ :: (a \to m\ d) \to TU\ m\ d$
$monoTPZ\ \ :: (a \to \mathbf{Zipper}\ e \to Maybe\ (\mathbf{Zipper}\ e)) \to TP\ e$
$monoTUZ\ \ :: (a \to \mathbf{Zipper}\ e \to (m\ d, \mathbf{Zipper}\ e)) \to TU\ m\ d$
$adhocTP\ \ :: TP\ e \to (a \to Maybe\ b) \to TP\ e$
$adhocTU\ \ :: TU\ m\ d \to (a \to m\ d) \to TU\ m\ d$
$adhocTPZ :: TP\ e \to (a \to \mathbf{Zipper}\ e \to Maybe\ (\mathbf{Zipper}\ e)) \to TP\ e$
$adhocTUZ :: TU\ m\ d \to (a \to \mathbf{Zipper}\ c \to (m\ d, \mathbf{Zipper}\ c)) \to TU\ m\ d$

Composition/Choice

$seqTP\ \ :: TP\ a \to TP\ a \to TP\ a$
$choiceTP :: TP\ a \to TP\ a \to TP\ a$
$seqTU\ \ :: TU\ m\ d \to TU\ m\ d \to TU\ m\ d$
$choiceTU :: TU\ m\ d \to TU\ m\ d \to TU\ m\ d$

AG Combinators

$(.\$) :: \mathbf{Zipper}\ a \to Int \to \mathbf{Zipper}\ a$
$\mathsf{parent} :: \mathbf{Zipper}\ a \to \mathbf{Zipper}\ a$
$(.|) :: \mathbf{Zipper}\ a \to Int \to Bool$
$(.\hat{}) :: (\mathbf{Zipper}\ a \to b) \to \mathbf{Zipper}\ a \to b$
$(.\hat{}\hat{}) :: (\mathbf{Zipper}\ a \to b) \to \mathbf{Zipper}\ a \to b$
$inherit :: (n \to Bool) \to (\mathbf{Zipper}\ a \to b) \to \mathbf{Zipper}\ a \to b$

Traversal Combinators

$$\begin{aligned}
&allTPright :: TP\ a \to TP\ a\\
&oneTPright :: TP\ a \to TP\ a\\
&allTUright :: TU\ m\ d \to TU\ m\ d\\
&allTPdown :: TP\ a \to TP\ a\\
&oneTPdown :: TP\ a \to TP\ a\\
&allTUdown :: TU\ m\ d \to TU\ m\ d\\
&atRoot :: TP\ a \to TP\ a
\end{aligned}$$

Traversal Strategies

$$\begin{aligned}
&full\_tdTP :: TP\ a \to TP\ a\\
&full\_buTP :: TP\ a \to TP\ a\\
&once\_tdTP :: TP\ a \to TP\ a\\
&once\_buTP :: TP\ a \to TP\ a\\
&stop\_tdTP :: TP\ a \to TP\ a\\
&stop\_buTP :: TP\ a \to TP\ a\\
&full\_uptdTP :: TP\ a \to TP\ a\\
&full\_upbuTP :: TP\ a \to TP\ a\\
&once\_uptdTP :: TP\ a \to TP\ a\\
&once\_upbuTP :: TP\ a \to TP\ a\\
&full\_tdTPupwards :: Proxy\ a \to TP\ a \to TP\ a\\
&innermost :: TP\ a \to TP\ a\\
&outermost :: TP\ a \to TP\ a\\
&full\_tdTU :: TU\ m\ d \to TU\ m\ d\\
&full\_buTU :: TU\ m\ d \to TU\ m\ d\\
&once\_tdTU :: TU\ m\ d \to TU\ m\ d\\
&once\_buTU :: TU\ m\ d \to TU\ m\ d\\
&stop\_tdTU :: TU\ m\ d \to TU\ m\ d\\
&stop\_buTU :: TU\ m\ d \to TU\ m\ d\\
&full\_uptdTU :: TU\ m\ d \to TU\ m\ d\\
&full\_upbuTU :: TU\ m\ d \to TU\ m\ d\\
&once\_uptdTU :: TU\ m\ d \to TU\ m\ d\\
&once\_upbuTU :: TU\ m\ d \to TU\ m\ d\\
&full\_tdTUupwards :: Proxy\ a \to TU\ m\ d \to TU\ m\ d\\
&foldr1TU :: TU\ m\ d \to \mathbf{Zipper}\ e \to (d \to d \to d) \to d\\
&foldl1TU :: TU\ m\ d \to \mathbf{Zipper}\ e \to (d \to d \to d) \to d\\
&foldrTU :: TU\ m\ d \to \mathbf{Zipper}\ e \to (d \to c \to c) \to c \to c\\
&foldlTU :: TU\ m\ d \to \mathbf{Zipper}\ e \to (c \to d \to c) \to c \to c
\end{aligned}$$

Memoized AG Combinators

$$\begin{aligned}
&(.@.) :: (\mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)) \to \mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)\\
&atParent :: (\mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)) \to \mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)\\
&atRight :: (\mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)) \to \mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)\\
&atLeft :: (\mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)) \to \mathbf{Zipper}\ a \to (r, \mathbf{Zipper}\ a)\\
&\mathbf{memo} :: attr \to AGTree\_m\ d\ m\ a \to AGTree\_m\ d\ m\ a
\end{aligned}$$

References

Adams, M.D., 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, pp. 13–24. http://dx.doi.org/10.1145/1863495.1863499.

Almeida, J.B., Cunha, A., Macedo, N., Pacheco, H., Proença, J., 2018. Teaching how to program using automated assessment and functional glossy games (experience report). Proc. ACM Program. Lang. 2 (ICFP), URL https://doi.org/10.1145/3236777.

Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A., 2007. Tom: Piggybacking rewriting on java. In: Baader, F. (Ed.), Term Rewriting and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 36–47.

Bird, R.S., 1984. Using circular programs to eliminate multiple traversals of data. Acta Inform. (21), 239–250, URL https://doi.org/10.1007/BF00264249.

Bürger, C., 2015. Reference attribute grammar controlled graph rewriting: Motivation and overview. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering. In: SLE 2015, Association for Computing Machinery, New York, NY, USA, pp. 89–100, URL https://doi.org/10.1145/2814251.2814257.

Cordy, J.R., 2004. TXL - a language for programming language tools and applications. Electron. Notes Theor. Comput. Sci. 110, 3–31, URL https://www.sciencedirect.com/science/article/pii/S157106610405217X. Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004).

Cowie, J., 2005. Detecting Bad Smells in Haskell. Tech. Rep., University of Kent, UK.

Dijkstra, A., Swierstra, S.D., 2005. Typing haskell with an attribute grammar. In: Vene, V., Uustalu, T. (Eds.), Advanced Functional Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–72.

Ekman, T., Hedin, G., 2007. The JastAdd extensible java compiler. SIGPLAN Not. 42 (10), 1–18, URL http://doi.acm.org/10.1145/1297105.1297029.

Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M., 2019. Memoized zipper-based attribute grammars and their higher order extension. Sci. Comput. Program. 173, 71–94, URL https://doi.org/10.1016/j.scico.2018.10.006.

Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M., 1992. Eli: A complete, flexible compiler construction system. Commun. ACM 35 (2), 121–130, URL https://doi.org/10.1145/129630.129637.

Huet, G., 1997. The zipper. J. Funct. Programming 7 (5), 549–554.

Knuth, D.E., 1968. Semantics of context-free languages. Math. Syst. Theory 2 (2), 127–145.

Knuth, D.E., 1990. The genesis of attribute grammars. In: Deransart, P., Jourdan, M. (Eds.), Attribute Grammars and their Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–12, URL http://dx.doi.org/10.1007/3-540-53101-7_1.

Kramer, L., Van Wyk, E., 2020. Strategic tree rewriting in attribute grammars. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. In: SLE 2020, Association for Computing Machinery, New York, NY, USA, pp. 210–229, URL https://doi.org/10.1145/3426425.3426943.

Kuiper, M., Saraiva, J., 1998. Lrc - a generator for incremental language-oriented tools. In: Koskimies, K. (Ed.), 7th International Conference on Compiler Construction. CC/ETAPS'98, In: LNCS, vol. 1383, Springer-Verlag, pp. 298–301.

Lämmel, R., Visser, J., 2002. Typed combinators for generic traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (Eds.), Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 137–154.

Luttik, S.P., Visser, E., 1997. Specification of rewriting strategies. In: Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications. Algebraic '97, BCS Learning & Development Ltd., Swindon, GBR, p. 9.

Macedo, J.N., Rodrigues, E., Viera, M., Saraiva, J., 2023. Efficient embedding of strategic attribute grammars via memoization. In: Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation. In: PEPM 2023, Association for Computing Machinery, New York, NY, USA, pp. 41–54, URL https://doi.org/10.1145/3571786.3573019.

Macedo, J.N., Viera, M., Saraiva, J., 2022. Zipping strategies and attribute grammars. In: Hanus, M., Igarashi, A. (Eds.), Functional and Logic Programming - 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10-12, 2022, Proceedings. In: Lecture Notes in Computer Science, vol. 13215, Springer, pp. 112–132, URL https://doi.org/10.1007/978-3-030-99461-7_7.

Martins, P., Fernandes, J.P., Saraiva, J., 2013. Zipper-based attribute grammars and their extensions. In: Du Bois, A.R., Trinder, P. (Eds.), Programming Languages. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 135–149.

Martins, P., Fernandes, J.P., Saraiva, J., Van Wyk, E., Sloane, A., 2016. Embedding attribute grammars and their extensions using functional zippers. Sci. Comput. Program. 132 (P1), 2–28, URL https://doi.org/10.1016/j.scico.2016.03.005.

Mernik, M., Korbar, N., Žumer, V., 1995. LISA: A tool for automatic language implementation. SIGPLAN Not. 30 (4), 71–79, URL https://doi.org/10.1145/202176.202185.

Reps, T., Teitelbaum, T., 1984. The synthesizer generator. SIGPLAN Not. 19 (5), 42–48, URL http://doi.acm.org/10.1145/390011.808247.

Saraiva, J., 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. pp. 268–282, URL http://dx.doi.org/10.1007/3-540-45821-2_17.

Sheard, T., Jones, S.P., 2002. Template meta-programming for haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. Haskell '02, Association for Computing Machinery, New York, NY, USA, pp. 1–16, URL https://doi.org/10.1145/581690.581691.

Sloane, A.M., Kats, L.C.L., Visser, E., 2010. A pure object-oriented embedding of attribute grammars. Electron. Notes Theor. Comput. Sci. 253 (7), 205–219, URL http://dx.doi.org/10.1016/j.entcs.2010.08.043.

Sloane, A.M., Roberts, M., Hamey, L.G.C., 2014. Respect your parents: How attribution and rewriting can get along. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (Eds.), Software Language Engineering. Springer International Publishing, Cham, pp. 191–210.

Söderberg, E., Hedin, G., 2013. Circular higher-order reference attribute grammars. In: Erwig, M., Paige, R.F., Van Wyk, E. (Eds.), Software Language Engineering. Springer International Publishing, Cham, pp. 302–321.

Swierstra, S.D., Azero Alcocer, P.R., Saraiva, J., 1999. Designing and implementing combinator languages. In: Swierstra, S.D., Oliveira, J.N., Henriques, P.R. (Eds.), Advanced Functional Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 150–206.

van den Brand, M.G.J., Deursen, A.v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J., 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, pp. 365–370.

Van Wyk, E., Bodin, D., Gao, J., Krishnan, L., 2008. Silver: an extensible attribute grammar system. Electron. Notes Theor. Comput. Sci. 203 (2), 103–116, URL http://dx.doi.org/10.1016/j.entcs.2008.03.047.

Visser, E., 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, pp. 357–362.

Vogt, H.H., Swierstra, S.D., Kuiper, M.F., 1989. Higher order attribute grammars. SIGPLAN Not. 24 (7), 131–145, URL https://doi.org/10.1145/74818.74830.

José Nuno Macedo is a Ph.D. student at the MAP-i doctoral program, in the University of Minho. He obtained a M.Sc. degree from University do Minho in 2018. He was a Invited Assistant Teacher at Departmento de Informática, Universidade do Minho, Braga, Portugal, from 2018 to 2023, teaching courses of object-oriented programming, software maintenance and evolution, and software analysis and testing.

Emanuel Rodrigues is a Ph.D. student at the Doctoral Program in Informatics, in the University of Minho. He obtained a M.Sc. degree from University do Minho in 2022.

Marcos Omar Viera Larrea was born September 09 1979 in San José, Uruguay. From 1998 to 2003, he studied Computing Systems Engineering at Universidad de la República, Uruguay. From 2004 to 2007, he did a Master in Computer Science at PEDECIBA, Uruguay. From 2008 to 2013, he was a Ph.D. student at PEDECIBA, and from 2012 also at Utrecht University, under the supervision of Doaitse Swierstra and Alberto Pardo. He works doing teaching activities at Universidad de la República since 2002. Since November 2012 he is an Assistant Professor at Department of Computer Science, Engineering School, Universidad de la República.

João Saraiva is an Associate Professor at the Departmento de Informática, Universidade do Minho, Braga, Portugal, and a researcher member of HASLab/INESC TEC. He obtained a M.Sc. degree from University do Minho in 1993 and a Ph.D. degree in Computer Science from Utrecht University in 1999. His main research contributions have been in the field of programming language design and implementation, program analysis and transformation, green software, and functional programming.