Zipping Strategies and Attribute Grammars

José Nuno Macedo[1], Marcos Viera[2], and João Saraiva[1]

[1] Dep. of Informatics & HASLab/INESC TEC, University of Minho, Braga, Portugal
jose.n.macedo@inesctec.pt , saraiva@di.uminho.pt
[2] Universidad de la República, Montevideo, Uruguay
mviera@fing.edu.uy

Abstract. Strategic term rewriting and attribute grammars are two powerful programming techniques widely used in language engineering. The former relies on *strategies* (recursion schemes) to apply term rewrite rules in defining transformations, while the latter is suitable for expressing context-dependent language processing algorithms. Each of these techniques, however, is usually implemented by its own powerful and large processor system. As a result, it makes such systems harder to extend and to combine.
We present the embedding of both strategic tree rewriting and attribute grammars in a zipper-based, purely functional setting. The embedding of the two techniques in the same setting has several advantages: First, we easily combine/zip attribute grammars and strategies, thus providing language engineers the best of the two worlds. Second, the combined embedding is easier to maintain and extend since it is written in a concise and uniform setting. We show the expressive power of our library in optimizing Haskell let expressions, expressing several Haskell refactorings and solving several language processing tasks for an Oberon-0 compiler.

Keywords: Attribute Grammars, Zippers, Strategic Term Rewriting

1 Introduction

Since Algol was designed in the 60's, as the first high-level programming language [38], languages have evolved dramatically. In fact, modern languages offer powerful syntactic and semantic mechanisms that improve programmers productivity. In response to such developments, the software language engineering community also developed advanced techniques to specify such new mechanisms.

Strategic term rewriting [19] and Attribute Grammars (AG) [14] have a long history in supporting the development of modern software language analysis, transformations and optimizations. The former relies on *strategies* (recursion schemes) to traverse a tree while applying a set of rewrite rules, while the latter is suitable to express context-dependent language processing algorithms. Many language engineering systems have been developed supporting both AGs [11, 26, 16, 23, 9, 8, 36] and rewriting strategies [5, 4, 17, 6, 30, 37]. These powerful systems, however, are large systems supporting their own AG or strategic specification language, thus requiring a considerable development effort to extend and combine.

A more flexible approach is obtained when we consider the embedding of such techniques in a general purpose language. Language embeddings, however, usually rely on advanced mechanisms of the host language, which makes them difficult to combine. For example, Strafunski [17] offers a powerful embedding of strategic term rewriting in Haskell, but it can not be easily combined with the Haskell embedding of AGs as provided in [25, 21]. The former works directly on the underlying tree, while the latter on a *zipper* representation of the tree.

In this paper, we present the embedding of both strategic tree rewriting and attribute grammars in a zipper-based, purely functional setting. Generic zippers [12] is a simple generic tree-walk mechanism to navigate on both homogeneous and heterogeneous data structures. Traversals on heterogeneous data structures (i.e. data structures composed of different data structures) is the main ingredient of both strategies and AGs. Thus, zippers provide the building block mechanism we will reuse for expressing the purely-functional embedding of both techniques. The embedding of the two techniques in the same setting has several advantages: First, we easily combine/zip attribute grammars and strategies, thus providing language engineers the best of the two worlds. Second, the combined embedding is easier to maintain and extend since it is written in a concise and uniform setting. This results in a very small library (200 lines of Haskell code) which is able to express advanced (static) analyses and transformation tasks. The purpose of this paper is three-fold:

- Firstly, we present a simple, yet powerful embedding of strategic term rewriting using generic zippers. This results in a concise library, named Ztrategic, that is easy to maintain and update. Moreover, our embedding has the expressiveness of the Strafunski library [17], as we showcase in section 4.
- Secondly, this new strategic term rewriting embedding can easily be combined with an existing zipper-based embedding of attribute grammars [22, 10]. By relying on the same generic tree-traversal mechanism, the zipper, (zipper-based) strategies can access (zipper-based) AG functional definitions, and vice versa. Such a joint embedding results in a multi-paradigm embedding of the two language engineering techniques. We show two examples of the expressive power of such embedding: First, we access attribute values in strategies to express non-trivial context-dependent tree rewriting. Second, strategies are used to define *attribute propagation patterns* [11, 8], which are widely used to eliminate (polluting) copy rules from AGs.
- Thirdly, we apply Ztrategic in real language engineering problems, namely, in optimizing Haskell let expressions, expressing a set of refactorings that eliminate several Haskell smells, and solving the LDTA Tool Challenge [35] tasks for name binding, type checking and desugaring of Oberon-0 programs.

This paper is organized as follows: Section 2 presents generic zippers and describes Ztrategic, our zipper-based embedding of strategic term rewriting. In Section 3, we describe zipper-based embedding of attribute grammars and we show how the two techniques/embeddings can be easily combined. In Section 4 we use the library to define several usage examples, such as refactorings of

Haskell source code and name binding, type checking and desugaring Oberon-0 source code. Section 5 discusses related work, and in Section 6 we present our conclusions.

2 Ztrategic: Zipper-Based Strategic Programming

Before we present our embedding in detail later in the section, let us consider a motivating example we will use throughout the paper. Consider the (sub)language of *Let* expressions as incorporated into most functional languages, including Haskell. Next, we show an example of a valid Haskell **let** expression and we define the heterogeneous data type *Let*, taken from [22], that models such expressions in Haskell itself.

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

```
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 | Const Int | Var String
```

We can write p as a Haskell value with type *Let*:

```
p = Let (Assign    "a" (Add (Var "b") (Const 0))
        (Assign    "c" (Const 2)
        (NestedLet "b" (Let (Assign "c" (Const 3) EmptyList)
                            (Add (Var "c") (Var "c")))
        EmptyList)))
        (Sub (Add (Var "a") (Const 7)) (Var "c"))
```

Consider now that we wish to implement a simple arithmetic optimizer for our language. Let us start with a trivial optimization: the elimination of additions with 0. In this context, strategic term rewriting is an extremely suitable formalism, 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. In our example, the optimization is defined in *Add* nodes, and thus we express the worker function as follows:

```
expr :: Exp → Maybe Exp
expr (Add e (Const 0)) = Just e
expr (Add (Const 0) e) = Just e
expr _                 = Nothing
```

The first two alternatives define the optimization: when either of the sub-expressions of an *Add* expression is the constant 0, then it returns the other sub-expression. A type-specific transformation function returns a *Maybe* result, transformations that fail or do not change the input return *Nothing*. This is the case of the last alternative of *expr*, that defines the default behaviour.

This function applies to *Exp* nodes only. To express our *Let* optimization, however, we need a generic mechanism that traverses *Let* trees, applying this

function when visiting *Add* expressions. This is where strategic term rewriting comes to the rescue: It provides recursion patterns (*i.e.*, strategies) to traverse the (generic) tree, like, for example, top-down or bottom-up traversals. It also includes functions to apply a node specific rewriting function (like *expr*) according to a given strategy. Next, we show the strategic solution of our optimization where *expr* is applied to the input tree in a full top-down strategy. This is a Type Preserving (*TP*) transformation since the input and result trees have the same type:

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

We have just presented our first zipper-based strategic function. Here, *step* is a transformation to be applied by function *applyTP* to all nodes of the input tree *t* (of type **Zipper** *Let*) using a full top-down traversal scheme (function *full_tdTP*). The rewrite step behaves like the identity function (*idTP*) by default with our *expr* function to perform the type-specific transformation, and the *adhocTP* combinator joins them into a single function.

This strategic solution relies on our Ztrategic [20] library: a purely functional embedding of strategic term rewriting in Haskell. In this solution we clearly see that the traversal function *full_tdTP* needs to navigate heterogeneous trees, as it is the case of the *Let* expression *p*. In a functional programming setting, zippers [12] provide a simple, but generic tree-walk mechanism that we will use to embed strategic programming in Haskell. In fact, our strategic combinators work with zippers as in the definition of *opt*. In the remaining of this section, we start by briefly describing zippers, and, next, we present in detail the embedding of strategies using this powerful mechanism.

2.1 The Zipper Data Structure

Zippers were introduced by Huet [12] to represent a tree together with a subtree that is the *focus* of attention. During a computation the focus may move left, up, down or right within the tree. Generic manipulation of a zipper is provided through a set of predefined functions that allow access to all of the nodes of a tree for inspection or modification.

A generic implementation of this concept is available as the *generic zipper* Haskell library [1], which works for both homogeneous and heterogeneous data types. In order to illustrate the use of zippers, let us consider again the tree used as an example for our *Let* program. We build a zipper t_1 from the previous *Let* expression *p* through the use of the **toZipper** :: *Data a* ⇒ *a* → **Zipper** *a* function. This function produces a zipper out of any data type, requiring only that the data types have an instance of the *Data* and *Typeable* type classes[3].

```
t1 = toZipper p
```

[3] These can be easily obtained via the Haskell data type **deriving** mechanism.

We can navigate t_1 using pre-defined functions from the zipper library. The function **down'** moves the focus down to the leftmost child of a node, while **down** moves the focus to the rightmost child instead. Similarly, functions **right**, **left** and **up**, move towards the corresponding directions. They all have type **Zipper** $a \to$ *Maybe* (**Zipper** a), meaning that such functions take a zipper and return a new zipper in case the navigation does not fail.

Finally, the zipper function **getHole** :: *Typeable* $b \Rightarrow$ **Zipper** $a \to$ *Maybe* b extracts the actual node the zipper is focusing on. Notice that the type of the hole (b) can be different than the type of the root of the Zipper (a), since the tree can be heterogeneous. Using these functions, we can freely navigate through this newly created zipper. Consider our expression p, we can *unsafely*[4] move the focus of the zipper towards the $b + 0$ subexpression and obtain its value as follows:

sumBZero :: *Maybe Exp*
sumBZero = (**getHole** . *fromJust* . **right** . *fromJust* . **down'** . *fromJust* . **down'**) t_1

The zipper library also contains functions for the transformation of the data structure being traversed. The function **trans** :: *GenericT* $\to$ **Zipper** $a \to$ **Zipper** a applies a generic transformation to the node the zipper is currently pointing to; while **transM** :: *GenericM* $m \to$ **Zipper** $a \to m$ (**Zipper** a) applies a generic monadic transformation.

2.2 Strategic Programming

In this section we introduce Ztrategic, our embedding of strategic programming using generic zippers. The embedding directly follows the work of Laemmel and Visser [17] on the Strafunski library [18].

We start by defining a function that elevates a transformation to the zipper level. In other words, we define how a function that is supposed to operate directly on one data type is converted into a zipper transformation.

zTryApplyM :: (*Typeable* a, *Typeable* b) $\Rightarrow$ ($a \to$ *Maybe* b) $\to$ *TP* c

The definition of *zTryApplyM*, which we omit for brevity, relies on transformations on zippers, thus reusing the generic zipper library **transM** function.

zTryApplyM 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. For example, if we are applying transformations on a zipper built upon the *Let* data type, then those transformations are of type *TP Let*.

type *TP* a = **Zipper** $a \to$ *Maybe* (**Zipper** a)

Very much like Strafunski, we introduce the type *TU* m d for Type-Unifying operations, which aim to gather data of type d into the data structure m.

type *TU* m d = (*forall* a . **Zipper** $a \to m$ d)

[4] By using the function *fromJust* :: *Maybe* $a \to a$ we assume a *Just* value is returned.

For example, to collect in a list all the defined names in a *Let* expression, the corresponding type-unifying strategy would be of type *TU* [] *String*. We will present such a transformation and implement it later in this section.

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.

```
adhocTP :: Typeable a ⇒ TP e → (a → Maybe a) → TP e
adhocTP f g z = maybeKeep f (zTryApplyM g) z
```

The *adhocTP* function receives transformations f and g as parameters, as well as zipper z. It converts g, which is a simple (*i.e.* non-zipper) Haskell function, into a zipper. Then, the zipper transformations f and g are passed as arguments to *maybeKeep*, which is an auxiliary function that applies the transformations in sequence, discarding either failing transformation (*i.e.* that produces *Nothing*). We omit the definition of *maybeKeep* for brevity.

Next, we use *adhocTP*, written as an infix operator, which combines the zipper function *failTP* with our basic transformation *expr* function:

```
step = failTP `adhocTP` expr
```

Thus, we do not need to express type-specific transformations as functions that work on zippers. It is the use of *zTryApplyM* in *adhocTP* that transforms a Haskell function (*expr* in this case) to a zipper one, hidden from these definitions.

The transformation *failTP* is a pre-defined transformation that always fails (returning *Nothing*) and *idTP* is the identity transformation that always succeeds (returning the input unchanged). They provide the basis for construction of complex transformations through composition. We omit here their simple definitions.

The functions we have presented already allow the definition of arbitrarily complex transformations for zippers. Such transformations, however, are always applied on the node the zipper is focusing on. Let us consider a combinator that navigates in the zipper.

```
allTPright :: TP a → TP a
allTPright f z = case right z of
                   Nothing → return z
                   Just r  → fmap (fromJust . left) (f r)
```

This function is a combinator that, given a type-preserving transformation f for zipper z, will attempt to apply f to the node that is located to the right of the node the zipper is pointing to. To do this, the zipper function **right** is used to try to navigate to the right; if it fails, we return the original zipper. If it succeeds, we apply transformation f and then we navigate **left** again. There is a similar combinator *allTPdown* that navigates downwards and then upwards.

With all these tools at our disposal, we can define generic traversal schemes by combining them. Next, we define the traversal scheme used in the function *opt* we defined at the start of the section. This traversal scheme navigates through the whole data structure, in a top-down approach.

```
full_tdTP :: TP a → TP a
full_tdTP f = allTPdown (full_tdTP f) `seqTP` allTPright (full_tdTP f) `seqTP` f
```

We skip the explanation of the *seqTP* operator as it is relatively similar to the *adhocTP* operator we described before, albeit simpler; we interpret this as a sequence operator. This function receives as input a type-preserving transformation f, and (reading the code from right to left) it applies it to the focused node itself, then to the nodes below the currently focused node, then to the nodes to the right of the focused node. To apply this transformation to the nodes below the current node, for example, we use the *allTPdown* combinator we mentioned above, and we recursively apply *full_tdTP f* to the node below. The same logic applies in regards to navigating to the right.

We can define several traversal schemes similar to this one by changing the combinators used, or their sequence. For example, by inverting the order in which the combinators are sequenced, we define a bottom-up traversal. By using different combinators, we can define choice, allowing for partial traversals in the data structure. We previously defined a rewrite strategy where we use *full_tdTP* to define a full, top-down traversal, which is not ideal. Because we intend to optimize *Exp* nodes, changing one node might make it possible to optimize the node above, which would have already been processed in a top-down traversal. Instead, we define a different traversal scheme, for repeated application of a transformation until a fixed point is reached:

```
innermost :: TP a → TP a
innermost s = repeatTP (once_buTP s)
```

We omit the definitions of *once_buTP* and *repeatTP* as they are similar to the presented definitions. The combinator *repeatTP* applies a given transformation repeatedly until a fixed point is reached, that is, until the data structure stops being changed by the transformation. The transformation being applied repeatedly is defined with the *once_buTP* combinator, which applies s once, anywhere on the data structure. When the application *once_buTP* fails, *repeatTP* understands a fixed point is reached. Because the *once_buTP* bottom-up combinator is used, the traversal scheme is *innermost*, since it prioritizes the innermost nodes. The pre-defined *outermost* strategy uses the *once_tdTP* combinator instead.

Let us return to our *Let* running example. Obviously there are more arithmetic rules that we may use to optimize let expressions. In Fig. 1 we present the rules given in [15].

In our definition of the function *expr*, we already defined rewriting rules for optimizations 1 and 2. Rules 3 through 6 can also be trivially defined in Haskell:

```
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
```

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

Fig. 1: Optimization Rules

Rule 7, however, is context dependent and it is not easily expressed within strategic term rewriting. In fact, this rule requires to first compute the environment where a name is used (according to the scope rules of the *Let* language). We will return to this rule in Section 3.

Having expressed all rewriting rules from 1 to 6 in function *expr*, now we need to use our strategic combinators that navigate in the tree while applying the rules. 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' combines all the steps we have built until now. We define an auxiliary function *step*, which is the composition of the *failTP* default failing strategy with *expr*, the optimization function; we compose them with *adhocTP*. Our resulting Type-Preserving strategy will be *innermost step*, which applies *step* to the zipper repeatedly until a fixed-point is reached. The use of *failTP* as the default strategy is required, as *innermost* reaches the fixed-point when *step* fails. If we use *idTP* instead, *step* always succeeds, resulting in an infinite loop. We apply this strategy using the function *applyTP* :: *TP c* → **Zipper** *c* → *Maybe* (**Zipper** *c*), which effectively applies a strategy to a zipper. This function is defined in our library, but we omit the code as it is trivial.

Next, we show an example using a Type-Unifying strategy. We define a function *names* that collects all defined names in a *Let* expression. First, we define a function *select* that focuses on the *Let* tree nodes where names are defined, namely, *Assign* and *NestedLet*. This function returns a singleton list (with the defined name) when applied to these nodes, and an empty list in the other cases.

```
select :: List → [String]
select (Assign s _ _)    = [s]
select (NestedLet s _ _) = [s]
select _                 = []
```

Now, *names* is a Type-Unifying function that traverses a given *Let* tree (inside a zipper, in our case), and produces a list with the declared names.

```
names :: Zipper Let → [String]
names r = applyTU (full_tdTU step) r
    where step = failTU `adhocTU` select
```

The traversal strategy influences the order of the names in the resulting list. We use a top-down traversal so that the list result follows the order of the input. This is to say that $names\ t_1 \equiv$ ["a", "c", "b", "c"] (a bottom-up strategy produces the reverse of this list).

As we have shown, our strategic term rewriting functions rely on zippers built upon the data (trees) to be traversed. This results in strategic functions that can easily be combined with a zipper-based embedding of attribute grammars [22, 10], since both functions/embedding work on zippers. In the next section we present in detail the zipping of strategies and AGs.

3 Strategic Attribute Grammars

Zipper-based strategic term rewriting provides a powerful mechanism to express tree transformations. There are, however, transformations that rely on contextual information that needs to be collected so the transformation can be applied. Our optimization rule 7 of Fig. 1 is such an example. In this section we will briefly explain the Zipper-based embedding of attribute grammars, through the *Let* example. Then, we are going to explain how to combine strategies and AGs, ending with an implementation of rule 7.

3.1 Zipper-based Attribute Grammars

The attribute grammar formalism is particularly suitable for specifying language-based algorithms, where contextual information needs to be collected before it can be used. Language-based algorithms such as name analysis [22], pretty printing [34], type inference [24], etc. are elegantly specified using AGs.

Our running example is no exception and the name analysis task of *Let* is a non-trivial one. Despite being a concise example, it has central characteristics of software languages, such as (nested) block-based structures and mandatory but unique declarations of names. In addition, the semantics of this implementation 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 name analysis naturally leads to a processor that traverses each block twice: once for processing 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. The uniqueness of identifiers is efficiently checked in the first traversal: for each newly encountered name it is checked whether that it has already been declared at the same lexical level (block). As a consequence, semantic errors resulting from duplicate definitions are

computed during the first traversal, and errors resulting from missing declarations in the second one. In fact, expressing this straightforward algorithm is a complex task in most programming paradigms, since it requires a complex scheduling of tree traversals[5], and intrusive code may be needed to pass information computed in one traversal to a specific node and used in a subsequent one[6].

In the attribute grammar paradigm, the programmer does not need to be concerned with scheduling of traversals, nor the use of intrusive code to glue traversals together. As a consequence, they do not need to adapt algorithms in order to avoid those issues. AGs associate *attributes* to grammar symbols (types in a functional setting), which are called *synthesized attributes* if they are computed bottom-up or *inherited attributes* if they are computed top-down.

Very much like strategic term rewriting, AGs also rely on a generic tree walk mechanism, usually called tree-walk evaluators [2], to walk up and down the tree to evaluate attributes. In fact, generic zippers also offer the necessary abstractions to express the embedding of AGs in a functional programming setting [22, 10]. Next, we briefly describe this embedding, and after that we present the embedded AG that express the scope rules of *Let*. It also computes (attribute) *env*, that is needed by the optimization rule 7.

To allow programmers to write zipper-based functions as AG writers do, the generic zippers library [1] is extended with some combinators:

- The combinator "*child*", written as the infix function **.$** to access the child of a tree node given its index (starting from 1).

  $(.\$) :: \textbf{Zipper}\ a \rightarrow \mathit{Int} \rightarrow \textbf{Zipper}\ a$

- The combinator parent to move the focus to the parent of a tree node.

  $\mathsf{parent} :: \textbf{Zipper}\ a \rightarrow \textbf{Zipper}\ a$

Having presented these zipper-based AG combinators, we now show in Fig. 3 the scope rules specified in the *Let* AG directly as a Haskell-based AG. We also show a visual representation of the AG in Fig. 2. Productions are shown with the parent node above and children nodes below, inherited attributes are on their left and synthesized attributes on their right, and arrows show how information flows between productions and their children to compute attributes.

In this AG 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 *Name* is synthesized in attribute *dclo*, which at the *Root* node is passed down as the environment (inherited attribute *env*). The type of the three attributes is a list of triples, associating the *Name* to the level it is defined (used to distingish declarations with the same name) and its *Let* expression definition[7]. Thus, we define a type synonym

---

[5] Note that only after building the environment of an outer block can nested ones be traversed: they inherited that environment. Thus, traversals are intermingled.

[6] This is the case when we wish to produce a list of errors that follows the sequential structure of the input program [27].

[7] We will use this definition to expand the *Name* as required by optimization rule 7.

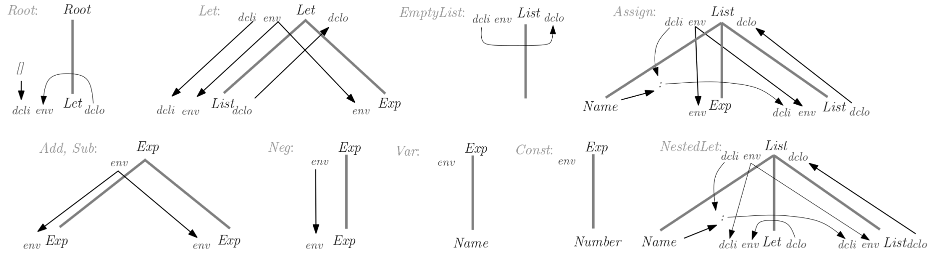

Fig. 2: Attribute Grammar Specifying the Scope Rules of *Let*

```
dclo :: Zipper Root → 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

lev :: Zipper Root → Int
lev t = case (constructor t) of
  Let_Let → case (constructor (parent t)) of
    NestedLet_List → (lev (parent t)) + 1
    Root_Root → 0
  _         → lev (parent t)

dcli :: Zipper Root → Env
dcli t = case (constructor t) of
  Let_Let → case (constructor (parent t)) of
    Root_Root        → [ ]
    NestedLet_List   → env (parent t)
  _       → case (constructor (parent t)) of
    Let_Let          → dcli (parent t)
    NestedLet_List   → (lexeme_Name (parent t), lev (parent t), Nothing) : (dcli (parent t))
    Assign_List      → (lexeme_Name (parent t), lev (parent t), lexeme_Exp (parent t))
                        : (dcli (parent t))

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

Fig. 3: Definitions of *dclo*, *lev*, *dcli* and *env* Attributes

type $Env = [(Name, Int, Maybe\ Exp)]$

We start by defining the equations of the synthesized attribute *dclo*. For each definition of an occurrence of *dclo* we define an equation in our zipper-based function. For example, in the diagrams of the *NestedLet* and *Assign* productions in Fig. 2 we see that *dclo* is defined as the *dclo* of the third child. Moreover, in production *EmptyList* attribute *dclo* is a copy of *dcli*. Let us consider the case of defining the inherited attribute *env*. 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* where *Let* subtrees occur. In both cases, *env* gets its value from the synthesized attribute *dclo* of the same non-terminal/type. We use the default rule of the case statement to express similar AG copy equations.

The inherited attribute *lev* is used to distinguish declarations with the same name in different scopes. We omitted this attribute in the visual AG of Fig. 2 since its equations are simple. This attribute is passed downwards as a copy of

the parent node/symbol, with two exceptions: when visiting a *Let* subtree whose parent is a Root, and when visiting a *NestedLet*. In the former the (initial) level is 0, while in the latter since we are descending to a nested block, we increment the level of the outer one.

Finally, let us define now the accumulator attribute *dcli*. The zipper function, when visiting nodes of type *Let* (which have *dcli* attributes) has to consider two alternatives: the parent node can be a *Root* or a *NestedLet* (the two occurrences of *Let* as a child in the diagrams of Fig. 2). This happens because the rules to define its value differ: in the *Root* node it corresponds to an empty list (our outermost *Let* is context-free), while in a nested block, the accumulator *dcli* starts as the *env* of the outer block. When visiting all other subtrees (expressed by the default rule), we need to define the inherited attribute *dcli* of *List* subtrees. There are three different cases: when the parent is a *Let* node, *dcli* is a copy of the parent. When the parent is an *Assign* then the *Name*, *level* and the associated *Exp* are accumulated in the *dcli* of the parent. Finally, in the case of *NestedLet* the *Name*, *level* and a *Nothing* expression is accumulated in *dcli*[8].

In order to specify the complete name analysis task of *Let* expression we need to report which names violate the scope rules of the language. We can modular and incrementally extend our AG [28], and define a new attribute *errors* to report such violations. In the next section *errors* is expressed as a strategic function.

3.2 Strategic Attribute Grammars

By having embedding both strategic term rewriting and attribute grammars in the same zipper-based setting, and given that both are embedded as first-class citizens, we can easily combine these two powerful language engineering techniques. As a result, attribute computations that do useful work on few productions/nodes can be efficiently expressed via our Ztrategic library, while rewriting rules that rely on context information can access attribute values.

Accessing Attribute Values from Strategies: As we mentioned in Section 3, rule 7 of Fig. 1 cannot be implemented using a trivial strategy, since it depends on the context. The rule states that a variable occurrence can be changed by its definition. Thus, we need to compute an environment of definitions, which is what we have done with the attribute *env*, previously. If we had access to such attribute in the definition of a strategy, we would be able to implement this rule.

Given that both attribute grammars and strategies use the zipper to walk through the tree, such combinations can be easily performed if the strategy exposes the zipper, so it can be used to apply the given attribute. This is done in our library by the *adhocTPZ* combinator:

$adhocTPZ :: Typeable\ a \Rightarrow TP\ e \rightarrow (a \rightarrow \mathbf{Zipper}\ e \rightarrow Maybe\ a) \rightarrow TP\ e$

[8] In this AG function we use boilerplate code $lexeme_{Name}$ and $lexeme_{Exp}$, which implement the so-called *syntactic references* in attribute equations [26]. They return the *Name* and *Exp* arguments of constructor *Assign*, respectively.

Notice that instead of taking a function of type $(a \to Maybe\ a)$, as does the combinator *adhocTP* introduced in Section 2, it receives a function of type $(a \to \textbf{Zipper}\ e \to Maybe\ a)$, with the zipper as a parameter. Then, we can define a worker function with this type, that implements rule 7:

$$\begin{array}{l} expC :: Exp \to \textbf{Zipper}\ Root \to Maybe\ Exp \\ expC\ (Var\ x)\ z = expand\ (i, lev\ z)\ (env\ z) \\ expC\ _\qquad\quad _ = Nothing \end{array}$$

where *expand* is a simple lookup function that replaces a name x for its definition in the environment (given by attribute *env*). This strategic function also uses attribute *lev* to look for the current or closest scope where name x is defined. As a final step, we combine this rule with the previously defined *expr* (rules 1 to 6) and apply them to all nodes.

$$\begin{array}{l} opt''\ :: \textbf{Zipper}\ Root \to Maybe\ (\textbf{Zipper}\ Root) \\ opt''\ r = applyTP\ (innermost\ (failTP\ `adhocTPZ`\ expC\ `adhocTP`\ expr))\ r \end{array}$$

Synthesizing Attributes via Strategies: We showed how attributes and strategies are combined by using the former while defining the latter. Now we show how to combine them the other way around; i.e. to express attribute computations as strategies. As an example, let us define the *errors* attribute, that returns the list of names that violate the scope rules. Note that duplicated definitions are efficiently detected when a new *Name* (defined in nodes *Assign* and *NestedLet*) is accumulated in *dcli*. The newly defined *Name must not be in* the environment *dcli* accumulated prior to that definition. Invalid uses are detected when a *Name* is used in an arithmetic expression (*Exp*). In this case, the *Name must be in*[9] the accumulated environment *env*. This is expressed by the following zipper functions:

$$\begin{array}{l} decls :: List \to \textbf{Zipper}\ Root \to [Name] \\ decls\ (Assign\qquad _\ _\ _)\ z = mNBIn\ (lexeme_{Name}\ z, lev\ z)\ (dcli\ z) \\ decls\ (NestedLet\ _\ _\ _)\ z = mNBIn\ (lexeme_{Name}\ z, lev\ z)\ (dcli\ z) \\ decls\ _\ _\qquad\qquad\qquad\quad = [\,] \end{array}$$

$$\begin{array}{l} uses :: Exp \to \textbf{Zipper}\ Root \to [Name] \\ uses\ (Var\ _)\ z = mBIn\ (lexeme_{Name}\ z)\ (env\ z) \\ uses\ _\ z\qquad\quad = [\,] \end{array}$$

Now, we define a type-unifying strategy that produces the list of errors.

$$\begin{array}{l} errors :: \textbf{Zipper}\ Root \to [Name] \\ errors\ t = applyTU\ (full_tdTU\ (failTU\ `adhocTUZ`\ uses\ `adhocTUZ`\ decls))\ t \end{array}$$

Although the applied function combines *decls* and *uses* in this order, the resulting list does not report duplicates first, and invalid uses after. The strategic function *adhocTUZ* combines the two functions and the default failing function into one, which is applied while traversing the tree in a top-down traversal, producing the errors in the order they occur. If we define *errors* as an attribute,

[9] Functions *mNBIn* and *mBIn* are trivial lookup functions. They are presented in [20].

most of the attribute equations are just propagating attribute values upwards without doing useful work! This is particularly relevant when we consider the *Let* sub-language as part of a real programming language (such as Haskell with its 116 constructors across 30 data types). Thus, combining attribute grammars with strategic term rewriting allows the leverage of the best of both worlds.

4 Expressiveness and Performance

In order to evaluate our combined zipper-based embedding of attribute grammars and strategic term-rewriting we consider three language engineering problems: First, we define a refactoring that eliminates the monadic *do-notation* from Haskell programs. Second, we evaluate the performance of our library by comparing the runtimes of an implementation in Ztrategic of a Haskell smell eliminator with its Strafunski counterpart when processing a large set of smelly Haskell programs. Third, we express in Ztrategic the largest language specification developed in this setting: the Oberon-0 language. The construction of a processor for Oberon-0 was proposed in the LDTA Tool Challenge [35], and it was concisely and efficiently specified using AGs and strategies in Kiama [31].

Do-notation elimination: We start by defining a refactoring that eliminates the syntactic sugar introduced by the monadic *do-notation*. In order to automate this refactoring, a type-preserving strategy is used to perform a full traversal in the Haskell tree, since such expressions can be arbitrarily nested. The rewrite step behaves like the identity function by default with a type-specific case for pattern matching the *do-notation* in the Haskell tree (constructor *HsDo*).

The following type-specific transformation function *doElim* just matches *HsDo* nodes and returns the correct desugared node, expressed at abstract syntax tree level. We omit the details of its representation as Haskell data types.

```
refactor :: Zipper HsModule -> Maybe (Zipper HsModule)
refactor h = applyTP (innermost step) h where step = failTP `adhocTP` doElim

doElim  :: HsExp -> Maybe HsExp
doElim (HsDo [HsQualifier e])                = Just e
doElim (HsDo (HsQualifier e : stmts))
  = Just ((HsInfixApp e (HsQVarOp (hsSymbol ">>")) (HsDo stmts)))
doElim (HsDo (HsGenerator _ p e : stmts)) = Just (letPattern p e stmts))
doElim (HsDo (HsLetStmt decls : stmts))   = Just (HsLet decls (HsDo stmts))
doElim _                                   = Nothing
```

We conclude that our library allows for the definition of powerful source code transformations in a concise manner. We also include a list desugaring implementation in our work's repository.

Smells Elimination: Source code smells make code harder to comprehend. A smell is not an error, but it indicates a bad programming practice. For example, inexperienced Haskell programmers often write $l \equiv [\,]$ to check whether a list is empty, instead of using the predefined *null* function. Next, we present a strategic function that eliminates several Haskell smells as reported in [7].

```
smellElim h = applyTP (innermost step) h
  where step = failTP `adhocTP` joinList `adhocTP` nullList
                     `adhocTP` redundantBoolean `adhocTP` reduntantIf
```

where *joinList* detects patterns where list concatenations are inefficiently defined, *nullList* detects patterns where a list is checked for emptiness, *redundantBoolean* detects redundant boolean checks, and *reduntantIf* detects redundant **if** clauses.

In order to assess the runtime performance of our zipper-based strategic term rewriting implementation, we compare it with the state-of-the-art, fully optimized Strafunski system. A detailed analysis of runtime performance of the zipper-based embedding of AGs is presented in [10], in which *memoized* zipper-based attribute grammars with very large inputs are benchmarked, showing that this AG embedding is not only concise and elegant, but also efficient.

Let us consider the Haskell smell eliminator expressed in both Ztrategic and Strafunski. To run both tools with large *smelly* inputs, we consider 150 Haskell projects developed by first-year students as presented in [3]. In these projects there are 1139 Haskell files totaling 82124 lines of code, of which exactly 1000 files were syntactically correct [10]. Both Ztrategic and Strafunski smell eliminators detected and eliminated 850 code smells in those files. To compare the runtime performance of both implementations, we computed an average of 5 runs, on a Ubuntu 16.04 machine, i5-7200U Dual Core, with 8 GB RAM. In this case, the very first version of Ztrategic, while being more expressive, is only 60% slower than the Strafunski library.

	Ztrategic	Strafunski
Lines of Code	22	22
Runtime	16.2s	10.2s
Average Memory	6607Kb	6580Kb

Table 1: Haskell Smell Eliminators in Ztrategic and Strafunski.

Oberon-0 in Ztrategic: The LDTA Tool Challenge [35] was a challenge focused on the construction of a compiler for the Oberon-0 language, with the goal of comparing the formalisms and toolsets used in it. The challenge was divided into 5 tasks: parsing and pretty-printing, name binding, type checking, desugaring and C code generation. These tasks were to be performed on the Oberon-0 language, which in itself was divided into 5 increasingly complex levels. We consider the level 2 (L2) of the Oberon-0 problem, and we specified the name binding, type checking and desugaring tasks in our Ztrategic AG approach. We use at-

Task	Ztrategic	Kiama
Oberon-0 Tree	57	99
Name analyzer	50	222
Type analyzer	34	117
Lifter	6	23
Desugarer	76	123
Total	223	584

Table 2: Numbers of Lines of Code for the Oberon-0 L2 tasks.

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

tributes for contextual information when needed, for example in name analysis to check whether a used name has been declared. This language level requires the desugaring of *For* and *Case* statements into semantically equivalent *While* and (nested) *If* statements. Such desugaring is implemented using Ztrategic type-preserving strategies, and the result is a new tree in which name analysis and type checking is performed through strategic traversals that use attributes. Because desugaring a *For* statement induces a new assignment (before the new *WhileStmt* statement) whose variable needs to be added to the declarations part of the original AST, we use the attribute *numForDown* which is a synthesized attribute of the original tree. Having the desugared AST and the number of *For* statements refactored, then we return the final higher-order tree where the induced variables are properly declared.

```
desugar m = let numberOfFors = numForsDown (toZipper m)
                step = failTP `adhocTP` desugarFor `adhocTPZ` desugarCase
                ata = fromJust (applyTP (innermost step) (toZipper m))
            in  injectForVars numberOfFors (fromZipper ata)
```

We omit here the definition of the worker function *desugarFor*. Its definition is fully included in [20], and it is also similar to the Kiama definition presented in [31]. In Table 2, we compare our approach to the results presented in [31] for the L2 language level. Notably, we show that our approach, even in its earliest versions, is suited for large-scale and real world usage.

5 Related Work

The work we present in this paper is inspired by the pioneering work of Sloane who developed Kiama [30, 13]: an embedding of strategic term rewriting and AGs in the Scala programming language. While our approach expresses both attribute computations and strategic term rewriting as pure functions, Kiama caches attribute values in a global cache, in order to reuse attribute values computed in the original tree that are not affected by the rewriting. Such global caching, however, induces an overhead in order to keep it updated, for example, attribute values associated to subtrees discarded by the rewriting process need to be purged from the cache [32]. In our purely functional setting, we only compute attributes in the desired re-written tree (as is the case of the let example shown in section 3.1). Influenced by Kiama, Kramer and Van Wyk [15] present *strategy attributes*, which is an integration of strategic term rewriting into attribute grammars. Strategic rewriting rules can use the attributes of a tree to reference contextual information during rewriting, much like we present in our work. They present several practical application, 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, we are unable to express strategy optimizations without relying on meta-programming techniques [29].

Nevertheless, our embeddings result in very simple libraries that are easier to extend and maintain, specially when compared to the complexity of extending a full language system such as Silver [36]. JastAdd is a reference attribute grammar based system [9]. It supports most of AG extensions, including reference and circular AGs [33]. It also supports tree rewriting, with rewrite rules that can reference attributes. JastAdd, however, provides no support for strategic programming, that is to say, there is no mechanism to control how the rewrite rules are applied. The zipper-based AG embedding we integrate in Ztrategic supports all modern AG extensions, including reference and circular AGs [22, 10]. Because strategies and AGs are first-class citizens we can smoothly combine any such extensions with strategic term rewriting.

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

6 Conclusions

This paper presented a zipper-based embedding of strategic term rewriting. By relying on zippers, we combine it with a zipper-based embedding of attribute grammars so that (zipper-based) strategies can access (zipper-based) AG functional definitions, and vice versa. We developed Ztrategic, a small but powerful strategic programming library and we have used it to implement several language engineering tasks.

To evaluate the expressiveness of our approach we compared our Ztrategic solution to the largest strategic AG developed with the state-of-the-art Kiama system. In terms of runtime performance we compared our Ztrategic library to the well established and fully optimized Strafusnki solution. The preliminary results show that in fact zippers provided a uniform setting in which to express both strategic term rewriting and AGs that are on par with the state-of-the-art. Moreover, our approach can easily be implemented in any programming language in which a zipper abstraction can be defined. In order to improve performance, we are considering extending Ztrategic to work with a memoized version of the AG library.

Acknowledgements

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project `LA/P/0063/2020`. The first author is also sponsored by FCT grant `2021.08184.BD`.

References

1. Adams, M.D.: Scrap your zippers: A generic zipper for heterogeneous types. In: WGP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming. pp. 13–24. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863495.1863499
2. Alblas, H.: Attribute evaluation methods. In: Alblas, H., Melichar, B. (eds.) International Summer School on Attribute Grammars, Applications and Systems. LNCS, vol. 545, pp. 48–113 (1991)
3. Almeida, J.B., Cunha, A., Macedo, N., Pacheco, H., Proença, J.: Teaching how to program using automated assessment and functional glossy games (experience report). Proc. ACM Program. Lang. **2**(ICFP) (Jul 2018), https://doi.org/10.1145/3236777
4. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on java. In: Baader, F. (ed.) Term Rewriting and Applications. pp. 36–47. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
5. 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.: The asf+sdf meta-environment: A component-based language development environment. In: Proceedings of the 10th International Conference on Compiler Construction. p. 365–370. CC '01, Springer-Verlag, Berlin, Heidelberg (2001)
6. Cordy, J.R.: Txl - a language for programming language tools and applications. Electronic Notes in Theoretical Computer Science **110**, 3–31 (2004). https://doi.org/https://doi.org/10.1016/j.entcs.2004.11.006
7. Cowie, J.: Detecting bad smells in haskell. Tech. rep., University of Kent, UK (2005)
8. Dijkstra, A., Swierstra, S.D.: Typing Haskell with an attribute grammar. In: Vene, V., Uustalu, T. (eds.) Advanced Functional Programming. pp. 1–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
9. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. SIGPLAN Not. **42**(10), 1–18 (Oct 2007), http://doi.acm.org/10.1145/1297105.1297029
10. Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M.: Memoized zipper-based attribute grammars and their higher order extension. Sci. Comput. Program. **173**, 71–94 (2019), https://doi.org/10.1016/j.scico.2018.10.006
11. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. Commun. ACM **35**(2), 121–130 (Feb 1992), https://doi.org/10.1145/129630.129637
12. Huet, G.: The Zipper. Journal of Functional Programming **7**(5), 549–554 (Sep 1997)
13. Kats, L.C.L., Sloane, A.M., Visser, E.: Decorated attribute grammars: Attribute evaluation meets strategic programming. In: Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. p. 142–157. CC '09, Springer-Verlag, Berlin, Heidelberg (2009)
14. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory **2**(2), 127–145 (1968)
15. Kramer, L., Van Wyk, E.: Strategic tree rewriting in attribute grammars. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. p. 210–229. SLE 2020, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3426425.3426943
16. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In: Koskimies, K. (ed.) 7th International Conference on Compiler Construction, CC/ETAPS'98. LNCS, vol. 1383, pp. 298–301. Springer-Verlag (April 1998)

17. Lämmel, R., Visser, J.: Typed combinators for generic traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) Practical Aspects of Declarative Languages. pp. 137–154. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
18. Lämmel, R., Visser, J.: A strafunski application letter. In: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages. p. 357–375. PADL '03, Springer-Verlag, Berlin, Heidelberg (2003)
19. Luttik, S.P., Visser, E.: Specification of rewriting strategies. In: Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications. p. 9. Algebraic'97, BCS Learning & Development Ltd., Swindon, GBR (1997)
20. Macedo, J.N., Viera, M., Saraiva, J.: The Ztrategic Library. https://bitbucket.org/zenunomacedo/ztrategic/ (2022)
21. Martins, P., Fernandes, J.P., Saraiva, J.: Zipper-based attribute grammars and their extensions. In: Du Bois, A.R., Trinder, P. (eds.) Programming Languages. pp. 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
22. Martins, P., Fernandes, J.P., Saraiva, J., Van Wyk, E., Sloane, A.: Embedding attribute grammars and their extensions using functional zippers. Sci. Comput. Program. **132**(P1), 2–28 (Dec 2016), https://doi.org/10.1016/j.scico.2016.03.005
23. Mernik, M., Korbar, N., Žumer, V.: Lisa: A tool for automatic language implementation. SIGPLAN Not. **30**(4), 71–79 (Apr 1995), https://doi.org/10.1145/202176.202185
24. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. p. 43–52. GPCE '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1868294.1868302
25. de Moor, O., Backhouse, K., Swierstra, D.: First-class attribute grammars. Informatica (Slovenia) **24**(3) (2000), citeseer.ist.psu.edu/demoor00firstclass.html
26. Reps, T., Teitelbaum, T.: The synthesizer generator. SIGPLAN Not. **19**(5), 42–48 (Apr 1984), http://doi.acm.org/10.1145/390011.808247
27. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. Ph.D. thesis, Utrecht University, The Netherlands (December 1999)
28. Saraiva, J.: 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 (2002), https://doi.org/10.1007/3-540-45821-2_17
29. Sheard, T., Jones, S.P.: Template Meta-Programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. p. 1–16. Haskell '02, Association for Computing Machinery, New York, NY, USA (2002), https://doi.org/10.1145/581690.581691
30. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electronic Notes in Theoretical Computer Science **253**(7), 205–219 (2010), http://dx.doi.org/10.1016/j.entcs.2010.08.043
31. Sloane, A.M., Roberts, M.: Oberon-0 in kiama. Science of Computer Programming **114**, 20–32 (2015). https://doi.org/https://doi.org/10.1016/j.scico.2015.10.010, lDTA (Language Descriptions, Tools, and Applications) Tool Challenge
32. Sloane, A.M., Roberts, M., Hamey, L.G.C.: 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. pp. 191–210. Springer International Publishing, Cham (2014)

33. Söderberg, E., Hedin, G.: Circular higher-order reference attribute grammars. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) Software Language Engineering. pp. 302–321. Springer International Publishing, Cham (2013)
34. Swierstra, S.D., Azero Alcocer, P.R., Saraiva, J.: Designing and implementing combinator languages. In: Swierstra, S.D., Oliveira, J.N., Henriques, P.R. (eds.) Advanced Functional Programming. pp. 150–206. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
35. van den Brand, M.: Introduction - the LDTA tool challenge. Science of Computer Programming **114**, 1–6 (2015). https://doi.org/https://doi.org/10.1016/j.scico.2015.10.015
36. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electronic Notes in Theoretical Computer Science **203**(2), 103–116 (2008), http://dx.doi.org/10.1016/j.entcs.2008.03.047
37. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Proceedings of the 12th International Conference on Rewriting Techniques and Applications. p. 357–362. RTA '01, Springer-Verlag, Berlin, Heidelberg (2001)
38. van Wijngaarcien, A., Mailloux, B.J., Peck, J.E.L., Kostcr, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G.: Revised Report on the Algorithmic Language Algol 68. SIGPLAN Not. **12**(5), 1–70 (May 1977), https://doi.org/10.1145/954652.1781176