The Fun of Programming with Attribute Grammars

Detailed summary of lecture as per Art. 8.o Dec.-Lei 239/2007

João Saraiva

Department of Informatics & HASLab/INESC TEC
Universidade do Minho, Braga, Portugal
saraiva@di.uminho.pt

Abstract. Attribute grammars have proven to be a suitable formalism to express complex programming language analysis and manipulation algorithms. In this document we follow Doaitse Swierstra's vision and we reason and structure complex, functional programs using the elegant attribute grammar programming style. We present a simple, but powerful embedding of attribute grammars in the *Haskell* language, allowing programmers to write attribute grammars directly as functional programs. We present in detail how attribute grammars describing complex algorithms are expressed in this new functional setting. Moreover, we also present both strict and lazy *Haskell* counterpart solutions which do show the use of additional intrusive gluing data structures in the former, and counter-intuitive circular definitions in the later. We show that these drawbacks are absent in the natural style of attribute grammar programming as provided by our embedding. Moreover, we combine our embedding of attribute grammar with the embedding of strategic term rewriting, thus providing a convenient setting to express large scale program transformations.
This document is written in memory of Doaitse Swierstra, Professor of Computer Science at the University of Utrecht; and mentor, friend and the main inspiration behind this research.

1 Introduction

Attribute Grammars (AGs) [1] are a well-known and convenient formalism for specifying the semantic analysis phase of a compiler and for modeling complex multiple traversal algorithms. Indeed, AGs have been used not only to implement general purpose programming languages, for example *Haskell* [2], *Java* [3], or *Oberon0* [4, 5], and domain specific languages, such as *Modelica* [6], and *EasyTime* [7], but also to specify sophisticated pretty printing algorithms [8], deforestation techniques [9, 10] and powerful type systems [11].

All these attribute grammars specify complex and large algorithms that rely on multiple traversals over large tree-like data structures. Expressing all these algorithms in regular programming languages is difficult because they rely on complex recursive patterns, and, most importantly, because there are dependencies between values computed in one traversal and used in following ones. In such cases, an explicit mechanism has to be used to glue together different traversal functions. In an imperative setting those values are stored in the tree nodes as side effects (working as a gluing data structure), while in a functional setting additional intermediate data structures have to be defined and constructed. For example, the large attribute grammar specifying the *SSL* AG language [12], used to bootstrap the *Lrc* AG system [13], produces a twelve tree-traversal functional program, whose traversal functions are glued together by eleven gluing data structures needed to explicitly pass information between them! Obviously, it would be extremely complex and time consuming to implement such a twelve traversal program by hand. In an AG setting, programmers do not have to concern themselves with scheduling complex traversal functions, nor do they have to define gluing data structures: such complex (functional) programs are generated by AG-based systems.

In fact, in his very first research paper, Swierstra already advocated the use of attribute grammars to derive efficient functional programs [14]. Indeed, very soon he realized that *"it is quite straightforward to use the attribute grammar based way of thinking when programming in the setting of a modern, lazily evaluated functional language: it is the declarative way of thinking in both formalisms that bridges the gap, and when using Haskell you get an attribute grammar evaluator almost for free"*[8].

The aim of this document is not to present in detail the attribute grammar formalism, but to support Swierstra's vision, that is to say, to motivate programmers to reason and to structure their complex, functional programs using the elegant attribute grammar programming style.

1.1 A Zipper-based Embedding of Attribute Grammars

We present a simple, but, at the same time, powerful purely functional embedding of attribute grammars which relies on a generic mechanism to navigate on heterogeneous trees, namely generic zippers [15]. When using this simple AG embedding *you get an attribute grammar evaluator totally for free*. We express this embedding in the *Haskell* programming language [16, 17], thus allowing programmers to implement their complex traversal algorithms directly as a *Haskell* program. Since zippers do not make essential use of lazy evaluation, the presented embedding can be expressed in other functional languages, as well. Thus, we provide an attribute grammar programming style that can be implemented in any functional setting.

Throughout this document we will analyze in great detail the problem of implementing the scope rules offered by modern programming languages. We present the AG-based solution, specified using an AG visual notation and the purely functional, zipper-based AG counterpart that directly follows from the

AG specification. Furthermore, we also present both strict and lazy *Haskell* solutions which do show their drawbacks, namely, the use of additional intrusive data structures in the former, and counter-intuitive circular definitions in the later. In fact, what is intrusive in the strict solution, and counter-intuitive in the lazy solution, is absent and natural in the style of AG programming as provided by our zipper-based AG embedding.

Although functional programming is known to provide concise, clear and modular solutions, when expressing complex traversal programs the straightforward functional solutions are not concise, since strict solutions define and manipulate additional data structures. They are not clear, either, since lazy solutions use counter-intuitive circular definitions. Finally, they are not modular since the additional data structures in the strict solution, and the *tupling* of all computations in a single function definition in the lazy solution, do limit their extensibility and reusability. On the contrary, our zipper-based embedding offers a modular and extensible software development environment: a new extension can be added in a modular way, without having to change the existing solution. Moreover, such module can be independently and incrementally compiled.

1.2 Zipping Strategies and Attribute Grammars

AGs are a convenient formalism to express tree-based computations. There are, however, some contexts where more powerful abstractions are needed. Consider for example a large-scale tree/program transformation problem where work has to be performed in a few (type of) nodes while traversing a large heterogeneous abstract syntax tree. In an AG setting, a new attribute has to be defined to compute/synthesize the transformed tree. Moreover, attribute equations have to be associated to most productions of the AG. These equations define the transformation needed in the desired (type of) nodes, but also the construction of (similar) new nodes in most of the productions of the AG. A functional setting does not provide a more concise solution either: such a large-scale transformation is usually expressed as a set of mutually recursive traversal functions (one per type) defined by alternatives that pattern-match the constructors of the tree nodes. Similarly to the AG equations, such alternative definitions specify not only the work to be done in the desired (type of) node, but also the recursive calls required to traverse the original tree while constructing the new one.

In this setting, strategic term re-writing [18] offers an adequate abstraction. It relies on *strategies* (recursion schemes) to apply term rewrite rules in defining transformations. A strategy is a generic transformation function that can traverse into heterogeneous data structures while mixing uniform and type-specific behavior. Thus, in a strategic programming style only the (type-specific) tree nodes/constructors to be transformed are included in the solution, together with the desired recursion scheme. The productions/constructors where no work is needed are simply omitted from the solution. Indeed, this results in an elegant and concise style of expressing tree transformation.

Strategic term re-writing relies on a generic mechanism to traverse heterogeneous data structures. This is exactly what generic zippers offer. Thus, we express a simple, yet powerful embedding of strategic term rewriting, using generic zippers. By relying on the same generic tree-traversal mechanism, zipper-based strategic functions 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. In Section [**?**], 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 [19, 20], which are widely used to eliminate (polluting) copy rules from AGs.

1.3 A Proper Zipper-based Embedding of Attribute Grammars

Our simple zipper-based embedding has the expressiveness of classical attribute grammars and their extensions [21]. However, as Swierstra immediately noticed when we showed him our embedding, it has an important drawback: it does not guarantee that an attribute value is computed only once, as required by the AG formalism. To provide a proper embedding of attribute grammars we extend our embedding with function memoization. By memoizing the calls to the zipper functions, we do avoid attribute re-calculation, and consequently we improve the performance of our zippers: the non-memoized version behaves as a quadratic function, while the memoized behaves as a linear one.

1.4 Content and Structure of this Document

The main content of this document is based on the paper that supported the invited tutorial given at the central European school on functional programming, held in Kosice, Slovakia in February 2018.

– *The Fun of Programming with Attribute Grammars*, João Saraiva, Marcos Viera, João Paulo Fernandes, Alberto Pardo, Proceedings of the Central European School on Functional Programming (CEFP'18), LNCS, 2022 (to appear).

The first seven sections presented here include the content of the this paper. These sections present the style of programming with AGs and introduce a simple zipper-based embedding in *Haskell*. Thus, this first part of the document is structured as follows: in Section 2 we discuss multiple traversal functions in a strict and lazy functional setting. In Section 3 we show how such functions can be expressed as attribute grammars. Section 4 introduces generic zippers, which is the simple mechanism we use to write attribute grammars as *Haskell* programs. In Section 5 we present in great detail the zipper-based embedding of attribute grammars, by specifying the name analysis task required by modern programming languages. In Section 6, we present the purely functional solutions, namely the strict and lazy functional evaluators, that are generated by AG systems and/or written by a functional programmer. In Section 7, we show one important feature of our AG embedding: modularity and extensibility.

In Section 8 we combine strategic-term rewriting with the AG formalism as presented in the paper:

– *Zipping Strategies and Attribute Grammars*, José Nuno Macedo, Marcos Viera, João Saraiva, 16th International Symposium on Functional and Logic Programming (FLOPS 2022), LNCS 2022 (to appear).

Section 8 starts with a brief presentation of how to express tree transformations using generic zippers (Section 8.1). In Section 8.2 we model strategic functions as zipper-based navigation functions. Section 8.3 combines our two zipper-based embedding of AGs and strategic term re-writing. Moreover, we show how (zipper-based) strategies can access (zipper-based) attribute definitions, and how to define synthesized attributes via strategies.

In Section 9, we extend our embedding with attribute memoization to avoid attribute re-calculation as we proposed in [22].

Finally, in Section 10 we describe related work, and in Section 11, we present our conclusions.

2 Multiple Traversal Functions

The functional programming paradigm contains multiple features that makes it a perfect setting to write functions over recursive data types, namely list and tree data types. Features like pattern matching, polymorphism, predefined lists, higher-order functions and algebraic data types, are standard in any functional programming language and are crucial ingredients to express those recursive functions.

For example, the sum of a list can be expressed in *Haskell* using pattern matching, by the following two alternative definitions:

$$\begin{aligned} &\mathit{sum}\ [\,] &&= 0\\ &\mathit{sum}\ (x : \mathit{xs}) &&= x + \mathit{sum}\ \mathit{xs}\end{aligned}$$

where $[\,]$ is the predefined notation for empty list, and $(:)$ is the *cons* operator, that prepends a head element to a list.

This is a polymorphic function: it sums all elements in the given list. However, the elements in the list need to be *numbers*, so that they can be added. Thus, the type of *sum* is

$$\mathit{sum} :: \mathit{Num}\ a \Rightarrow [\,a\,] \to a$$

where the constraint *Num a* ensures that values of type a do have the addition function pre-defined $(+) :: a \to a \to a$. As the reader may have noticed, the *sum* function performs always the same operation in the *cons* alternative (the addition) and it produces a constant (zero) for the empty list alternative. In a similar way, but with another operation and constant, we can define the function *prod*:

$$\begin{aligned} &\mathit{prod}\ [\,] &&= 1\\ &\mathit{prod}\ (x : \mathit{xs}) &&= x \times \mathit{prod}\ \mathit{xs}\end{aligned}$$

As we can see by their recursive definitions, both *sum* and *prod* use a similar recursion pattern. In fact, this pattern can be expressed by a higher order function (a function that takes functions as arguments), named *foldr*, as follows:

$$\begin{aligned} &\mathit{foldr} :: (a \to b \to b) \to b \to [\,a\,] \to b \\ &\mathit{foldr}\ f\ \ c\ [\,] \qquad\quad = c \\ &\mathit{foldr}\ f\ \ c\ (x : \mathit{xs}) = f\ \ x\ (\mathit{foldr}\ f\ \ c\ \mathit{xs}) \end{aligned}$$

These and other recursive functions on lists can be easily defined by reusing this pre-defined higher order function. Next, we present the *sum* and *prod* of lists expressed as a *foldr*:

$$\begin{aligned} &\mathit{sum}\ \ = \mathit{foldr}\ (+)\ 0 \\ &\mathit{prod} = \mathit{foldr}\ (\times)\ 1 \end{aligned}$$

Functions showing similar recursion patterns can be tupled together in order to improve their efficiency by computing several values at once, and thus by eliminating unnecessary traversals. Consider for example that we wish to compute the average of a list, by first computing the *length* and *sum* of the given list. The *length* of a list can be defined as a *foldr*, where we use an anonymous function as the first argument:

$$\mathit{length} = \mathit{foldr}\ (\lambda x\ \ l \to l + 1)\ 0$$

Now, we may easily express the *average* function as follows

$$\mathit{average}\ l = \mathit{sum}\ l\ /\ \mathit{length}\ l$$

But, this straightforward solution performs two traversals over the input list! By tupling [23] we can fuse the two functions into one, and as result we do the work in a single traversal:

$$\mathit{sumLength} = \mathit{foldr}\ (\lambda x\ (s, l) \to (s + x, l + 1))\ (0, 0)$$

and, now *average* just divides the two results of *sumLength*.

$$\begin{aligned} \mathit{average}\ l = &\ \mathbf{let}\ (s, l) = \mathit{sumLength}\ l \\ &\ \mathbf{in}\ \ s\ /\ l \end{aligned}$$

Obviously, only one traversal is performed since function *average* performs just one call to *sumLength*. Thus, functional programming is an elegant and concise setting to express such problems while relying on formal techniques to eliminate unnecessary traversals. In fact, there exists extensive work on tupling [23], shortcut fusion [24–26] and deforestation [27], which are techniques aiming at eliminating tree traversals and/or intermediate data types produced/consumed by traversal functions.[1]

There are, however, problems that do *require* multiple traversals over the underlying data structure, and techniques like tupling or shortcut fusion are not able to eliminate them. Consider, for example, the well-known *repmin* problem:

[1] All these techniques are presented in two tutorials on program optimization and transformation in calculation form [28] and on shortcut deforestation [29].

to transform a binary leaf tree[2] 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.

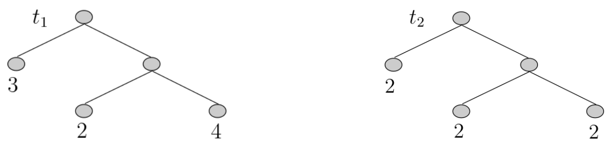


Fig. 1: The *Repmin* problem: input (left) and output (right) binary leaf trees.

To model such binary tree we consider the following algebraic data type:[3]

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

where *Leaf* and *Fork* are data type constructors, which are functions used to construct instances of binary leaf trees. For example, tree t_1 (graphically shown in Fig. 1) is defined in *Haskell* as follows:

```
t1 :: Tree
t1  = Fork (Leaf 3) (Fork (Leaf 2) (Leaf 4))
```

Next, we express the recursive function *tmin*, that computes the minimum value of a tree,

```
tmin :: Tree → Int
tmin (Leaf n)   = n
tmin (Fork l r) = min (tmin l) (tmin r)
```

where *min* computes the minimum of two given numbers.

Similarly, we can express the recursive function *replace*, that places a concrete given value in all the leaves of a tree:

```
replace :: Tree → Int → Tree
replace (Leaf n)   m = Leaf m
replace (Fork l r) m = Fork (replace l m) (replace r m)
```

We may combine the above implementations of *replace* and *tmin* in a simple function in order to obtain a solution to *repmin*:

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

[2] A tree that stores its values in the leaves.

[3] Throughout this document, we use colors to make our code more readable. Thus, this document should be printed in color, or read via its electronic form.

Regarding this implementation, the underlying tree t is traversed twice: once to compute the minimum value and a second time to replace the value stored in the leaves by the computed minimum. In fact, there are two recursive functions with t as argument.

In a lazy functional language, such as *Haskell*, this is not strictly necessary. In fact, Bird [30] showed how to remove these multiple traversals from programs such as *repmin* by combining tupling with lazy evaluation, a strategy that we review next.

2.1 Circular, Lazy Functions

As we previously did with *sum* and *length*, given the common recursive pattern of the functions *tmin* and *replace*, we can tuple them into one function *tminReplace*:

```
tminReplace :: Tree → Int → (Int, Tree)
tminReplace (Leaf n)   m = (n,            Leaf m)
tminReplace (Fork l r) m = (min ml mr, Fork tl tr)
  where (ml, tl) = tminReplace l m
        (mr, tr) = tminReplace r m
```

If we analyze this function in detail we observe that, as expected, it computes the minimum number of the input tree and the desired tree result, giving the initial tree and the (minimum) value to store in the leaves of the new tree. The value needed to be stored in the new tree, however, is part of the result of that same function!

Thus, to implement *repmin* we need to express that such value (*i.e.*, the second argument) is in fact the first component of its result. In *Haskell* we can express this via the so-called *pseudo circular definitions*, in which arguments in a function call depend on the results of that same call. That is, they contain definitions such as:

$$(..., x, ...) = f \; ... \; x \; ...$$

These counter-intuitive functional dependencies are allowed under a lazy evaluation setting, since values are computed *on demand*. Thus, as the last transformation step proposed by Bird, we use circular definitions to obtain the circular, lazy *repmin* function:[4]

```
repmin :: Tree → Tree
repmin t = nt
```

$$\text{where } (\boxed{\text{m}}, nt) = tminReplace\ t\ \boxed{\text{m}}$$

This circular definition does not induce a circularity, since in the *tminReplace* function, its second argument is only needed after the minimum of the tree is computed. In a lazy functional language, the lazy evaluation engine does the magic to evaluate functions with pseudo circular definitions in the desired order.

[4] In order to make it easier for the reader to identify circular definitions, we frame the occurrences of variables that induce them (m in this case).

Obviously, if the circular definition expresses a *real* circularity, then the evaluation order does not exist, and the lazy evaluation mechanism does not terminate. It should also be noticed that this solution consists of a single traversal function since there is a single call with the input tree t as argument.

2.2 Circular, Lazy Programs: Pros and Cons

Circular, lazy programs show the power of lazy evaluation: multiple traversal functions can be expressed with a single function definition without the programmer having to schedule the tree traversal functions. Although the simple *repmin* program is naturally expressed as a two traversal function, there are many algorithms that do rely on a complex and intermingled set of traversal functions.

When expressing such algorithms in a circular, lazy setting the programmers do not have to schedule tree traversal functions: the lazy evaluation mechanism does that for free! There is, however, a price to pay when using this style of lazy functional programming:

- *No modularity*: Clear and concise functions are tupled together into a single one, which forbids the reuse of the individual ones. For example, a widely used function like *tmin* that computes the minimum of a tree is now tupled with the computation of the new tree. As a consequence, it can not be reused *per se*.[5]
- *Complex to Write and to Evolve*: The counter-intuitive circular definitions combined with the lack of modularity makes circular, lazy programs, complex to write, to understand and to debug. Moreover, a circular definition may in fact define a *real* circularity, which is only detected at runtime since the function does not terminate. As a consequence, such programs are not only difficult to write, but also to maintain, to debug and to extend/evolve.
- *Performance*: Although we may think that one single traversal is more efficient than multiple ones, the overhead due to lazy evaluation may significantly compromise performance (both in terms of runtime and memory consumption) as shown in several papers [10, 34, 35].

Thus, circular, lazy programs are not the perfect setting to express such multiple traversal programs. Although, multiple traversal program counterparts seem to be modular, they may lead to non modular solutions as well! In fact, multiple traversal algorithms very often rely on the definition, construction and destruction of additional intermediate data structures that are needed to explicitly pass information computed in one traversal function, which has to be used by the following traversal functions. Consequently, the introduction of such intrusive data structures as additional arguments/results of the multiple traversal functions can drastically decrease their modularity, as well.

[5] One solution would be to formally calculate the circular version from strict ones as proposed in [31–33]. However, the individual traversals need to be expressed as catamorphisms, which is often not possible.

In Section 4, we will discuss in great detail a simple name analysis problem for the widely used *let in* block-based structures. This example will show how cumbersome a multiple traversal function can be in a (strict and lazy) functional programming setting.

3 Traversal Functions with an Attribute Grammar Flavor

What is counter-intuitive in a (lazy) functional programming setting, and is available to advanced programmers only, is simple and natural in the AG formalism. Indeed, this type of multiple tree traversal programs can be easily expressed as AGs. Thus, instead of pursuing a complex and counter-intuitive way to implement a problem, we can follow an easier and more natural path to solve that same problem via attribute grammars.

Next, we will present AGs in a simple visual notation: the aim of this section is not to introduce in detail the AG formalism, but instead to motivate functional programmers to structure their traversal functions in the elegant style of attribute grammar programming.[6]

3.1 The *Repmin* Problem with an Attribute Grammar Flavor

Let us return to the *repmin* problem and express it within the AG formalism. Instead of presenting the formal AG definition of *repmin*, we will adopt a visual notation that is often used by AG developers to sketch a first draft of their grammars.

Types as Grammars: Grammars specify formal languages, where a formal language consists of a (possibly non-finite) set of sentences. Similarly, types define a (possibly non-finite) set of values. A parser of the language (usually generated from its grammar) checks whether a given input is a sentence of the language, *i.e.* whether it is in its set of sentences. In functional programming, a type inference mechanism produces a type checker that checks if a value is type correct, *i.e.* if it is in the set defined by the (inferred) type. Although context-free grammars are widely used to specify syntactic parsers, in the AG formalism they are also used to define abstract tree structures. Thus, grammars can be seen as types, where a type corresponds to a non-terminal symbol and a type constructor to a production. Thus, in *repmin* *Tree* is a non-terminal and *Leaf* and *Fork* are productions (names).[7]

Function Results as Synthesized Attributes: Note that in *repmin* we need to compute the minimum number stored in the leaves of the tree. In AGs we associate

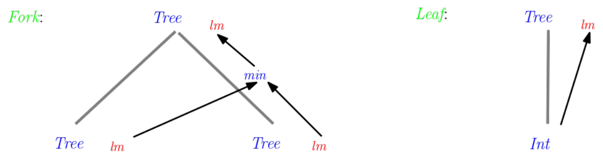
[6] For a formal definition of AGs, we recommend to the interested reader the following papers [8, 36].

[7] In fact, the well-known *SSL* AG notation [37] uses a notation for (abstract) grammars very similar to the *Haskell* data type one.

instances of attributes to tree nodes that are used to synthesize such values. In fact, *synthesized attributes* propagate attribute values upwards in the tree.

Thus, to express the computation of the minimum value we define a synthesized attribute, with name *lm* (local minimum) of type *Int*, that we associate to non-terminal *Tree*. Next, we present the Visual AG fragment (VAG_i) representing the equations that define this synthesized attribute for the two productions of *Tree*.

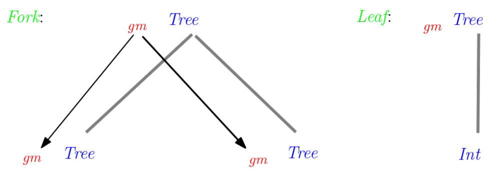
(VAG_i)



In this notation, occurrences of synthesized attributes are shown on the right of their non-terminals, and we omit their types. Moreover, the attribute equations of the productions are defined with upwards arrows as expressed by synthesized attributes. For example, in constructor/production *Fork* the (*Haskell*) *min* function gets as arguments the synthesized minimum of the two subtrees of type *Tree* (corresponding to the two right-hand side nonterminals of the production), in order to synthesize the minimum of the tree (*i.e.*, the left-hand side nonterminal, or the "parent"). For the *Leaf* production, the synthesized minimum is defined as the *Int* value stored in the leaf.

Function Arguments as Inherited Attributes: Having synthesized the minimum number of the *repmin* tree, we need to pass it downwards to the leaves, in order to compute the new tree with that number stored in the leaves. AGs use *inherited attributes* to pass information downwards the tree. Thus, we introduce an inherited attribute, named *gm* (global minimum) of type *Int*, which is inherited by all *Tree* nodes (symbols). We display occurrences of inherited attributes on the left of their non-terminals. In this case they are just copied downwards the tree, as shown next:

(VAG_{ii_a})



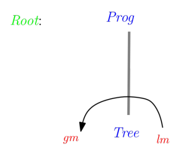
In order to express within the AG formalism that the global minimum *gm* of the *repmin* tree is the local minimum *lm* of the root of the tree, we introduce a new "root" nonterminal symbol/type *Prog*:[8]

```
data Prog = Root Tree
```

[8] The root nonterminal symbol of an AG does not have inherited attributes [1]. In other words, (attribute) grammars are context-free.

In its production/constructor *Root*, the *Tree* corresponds to the root of the *repmin* tree. *Tree* has two attributes already: the inherited *gm* and the synthesized *lm*, which we display on its left/right side, respectively. Now, we can express the equation stating that the global minimum of that non-terminal symbol is the synthesized attribute of that same symbol.

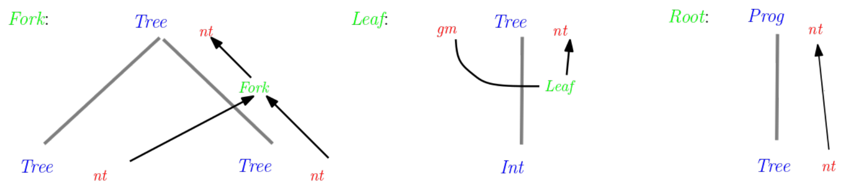
(VAG_{ii_b})



An interested reader may have already noticed that this equation - inducing a dependency from an synthesized to an inherited attribute of the same nonterminal - corresponds to the counter-intuitive circular definition in a lazy program. Such equations are, however, intuitive, natural and widely used in AG programming.

We can now synthesize the desired new tree (*nt*). In a *Fork* production, the new tree of the left subtree and the new tree of the right subtree are combined in a new *Fork* tree. For a *Leaf*, a new *Leaf* is constructed containing the inherited global minimum *gm*.

(VAG_{iii})



The visual AG fragments (VAG_i), (VAG_{ii_a}), (VAG_{ii_b}), and (VAG_{iii}) form the full *repmin* AG.

3.2 Specifying Attribute Grammars in a Functional Setting

We have shown the full (visual) AG specification of the *repmin* problem. AGs based systems accept as input a textual AG specification and generate AG implementations, the so-called attribute evaluators.

Next, we show the *repmin* written in the *Haskell*-based Utrecht University AG (UUAG) system [20], as included on the UUAG manual [38]. In UUAG we need to define the (abstract) grammar, very much like the *Haskell* data types we introduced for *repmin*.

```
DATA Prog | Root tree : Tree
DATA Tree | Fork left : Tree right : Tree
          | Leaf int : Int
```

We declare a synthesized attribute[9] *lm* representing the minimum value of a *Tree*, and we specify its equations very much like in the visual notation shown on (VAG_i).

(AG_i)
```
ATTR Tree [|| lm : Int]
SEM  Tree | Leaf
              lhs . lm = @int
          | Fork
              lhs . lm = min @left . lm @right . lm
```

To make the global minimum available at all the leaves, the inherited attribute *gm* is declared, and its equations just distribute it downwards to all the leaves.

(AG_{ii_a})
```
ATTR Tree [gm : Int || ]
SEM  Tree | Fork
              left . gm  = @lhs . gm
              right . gm = @lhs . gm
```

At the root node of a *Tree*, we use the synthesized local minimum *lm* to define the inherited attribute *gm*.

(AG_{ii_b})
```
SEM Prog | Root
             tree . gm = @tree . lm
```

As the reader may have noticed, the two AG fragments correspond literally, *i.e. textually*, to the visual fragments (VAG_{ii_a}) and (VAG_{ii_b}). To finish the *repmin* UUAG, we express fragment (VAG_{iii}) textually, too.

(AG_{iii})
```
ATTR Tree [|| nt : Tree]
ATTR Prog [|| nt : Tree]
SEM  Tree | Fork
              lhs . nt = Fork @left . nt @right . nt
          | Leaf
              lhs . nt = Leaf @lhs . gm
SEM  Prog | Root
              lhs . nt = @tree . nt
```

Functional, attribute grammar-based systems, like the UUAG and the *Lrc* system [13], produce purely functional AG implementations. In fact, these systems generate both the multiple traversal (strict), and the circular, lazy *repmin*, we presented before. In Section 6, we will discuss in detail the implementation of AGs in a functional setting.

4 Zipper-based Multiple Traversal Functions

Attribute grammars are traditionally implemented by functions that traverse the underlying abstract syntax tree while computing attribute values. Such evaluators are usually called tree-walk evaluators, that, as the name suggest, walk

[9] In an ATTR definition, there are three places to put attribute declarations [*inherited* | *inherited/synthesized* | *synthesized*].

up and down the tree to evaluate attributes [39]. In a functional programming setting, Zippers [15] provide a simple, but generic tree-walk mechanism that we will use to model AGs directly as a *Haskell* program. In other words, Zippers will be the basic machinery to embed AGs in the general purpose *Haskell* language.

4.1 Functional Zippers

Zippers were originally conceived by Huet [15] 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 the tree for inspection or modification.

In order to illustrate the use of zippers, let us consider again the binary leaf-tree used in the *repmin* problem in Section 2, and that consists of the single data type *Tree*:

```
t1 :: Tree
t1 = Fork (Leaf 3)
          (Fork (Leaf 2)
                (Leaf 4))
```

When we consider t_1 as a whole, we notice that each of the subtrees occupies a certain location in tree. In fact, a location of a subtree may be represented by the subtree itself and by the rest of the tree. Thus, the rest of the tree can be viewed as the context of that subtree.

One of the possible ways to represent this context is as a path from the top of the tree to the node which is the focus of attention (where the subtree has its root). This means that it is possible to use contexts and subtrees to define any position in the tree, while creating a setting where navigation is provided by the contextual information of a path applied to a tree.

For example, if we wish to put the focus in *Leaf* 2, its context in tree t_1 is:

```
tree = Fork (Leaf 3)
            (Fork ⊗
                  (Leaf 4))
```

where ⊗ points the exact location where *Leaf* 4 occurs. One possible way to represent this context consists in defining a path from the top of the tree to the position which is the focus of attention. To reach *Leaf* 2 in tree t_1, we need to go right (down the right branch) and then left (down the left one). In practice, this means that given the tree t_1, then the path [right, left] would suffice in indicating the location in the tree we want to focus our attention on.

This is precisely the main concept behind zippers. Using the idea of having pairs of information composed by paths and trees, we may represent any positions in a tree and, better yet, this setting allows an easy navigation as we only have to remove parts of the path if we want to go to the top of the tree, or add information if we want to go further down.

Using this idea, we may represent contexts as instances of the following datatype:

```
data Cxt a = Top
           | L (Cxt a) a
           | R a       (Cxt a)
```

A value $L\ c\ t$ represents the left part of a branch of which the right part is t and whose parent has context c. Similarly, a value $R\ t\ c$ represents the right part of a branch of which the left part is t and whose parent has context c. The value *Top* represents the top of the tree. Returning to our example, we may represent the context of *Leaf* 2 in tree as:

```
context :: Cxt Tree
context = L (R (Leaf 3) Top) (Leaf 4)
```

which is the same as saying that the focus results from going to the left in a subtree whose right side is *Leaf* 4. This subtree is obtained from going to the right from the top tree, whose left side is *Leaf* 3.

Having defined the context of a subtree, we may define a location on a tree as one of its subtrees together with its context:

```
type Loc a = (a, Cxt a)
```

We are now ready to present the definition of useful functions that manipulate locations on binary trees. First, we can define a function that goes down the left branch of a tree:

```
left :: Loc Tree → Loc Tree
left (Fork l r, c) = (l, L c r)
```

This function takes a tuple with a tree and a context (a *Loc*) and creates a new tuple, where the left side of the tree becomes the new tree, and a new context is created, extending the previous one with a data constructor L whose right sides becomes the right side of the previous tree. Similarly, we introduce a function that goes down the right branch:

```
right :: Loc Tree → Loc Tree
right (Fork l r, c) = (r, R l c)
```

Going up in the tree corresponds to remove "directions" from the path. The function *parent* goes up on a tree location:

```
parent :: Loc Tree → Loc Tree
parent (t, L c r) = (Fork t r, c)
parent (t, R l c) = (Fork l t, c)
```

This function has two alternatives: when it gets as argument a location whose context is a left side (L), then it creates a new location where the right side of the context is moved to a new tree and the left side is kept as the context (c). The second alternative does the opposite when the context of the location is a right side (R).

Finally, we define a function *zipper* that creates a tree location from a tree (where we define an empty context):

$$\begin{aligned}&\mathit{zipper} :: \mathit{Tree} \to \mathit{Loc}\ \mathit{Tree}\\ &\mathit{zipper}\ t = (t, \mathit{Top})\end{aligned}$$

and we also define another function that extracts a tree from a tree location, by simply taking the first element of the location pair:

$$\begin{aligned}&\mathit{value} :: \mathit{Loc}\ \mathit{Tree} \to \mathit{Tree}\\ &\mathit{value} = \mathit{fst}\end{aligned}$$

Having defined a zipper-based function to navigate in binary-trees, we can now focus the attention on subtree *Leaf* 2 of t_1 as follows:

$$\begin{aligned}&\mathit{leafWith2} :: \mathit{Loc}\ \mathit{Tree}\\ &\mathit{leafWith2} = \mathit{left}\ (\mathit{right}\ (\mathit{zipper}\ t_1))\end{aligned}$$

Once the subtree that is the focus of our attention is reached, we may perform several actions on it. One such action would be to edit that subtree: we may want, for example, to decrement its leaf value by one. Using the zipper functions defined so far, this is expressed as follows:

$$\begin{aligned}\mathit{edit} = \mathbf{let}\ &(\mathit{Leaf}\ v, \mathit{cxt}) = \mathit{leafWith2}\\ &\mathit{subtree}' \quad\;\; = (\mathit{Leaf}\ (v-1), \mathit{cxt})\\ \mathbf{in}\ \ &\mathit{value}\ (\mathit{parent}\ (\mathit{parent}\ \mathit{subtree}'))\end{aligned}$$

with *edit* producing the result:

$$\begin{aligned}\mathit{edited} = \mathit{Fork}\ &(\mathit{Leaf}\ 3)\\ &(\mathit{Fork}\ (\mathit{Leaf}\ 1)\\ &\qquad\quad (\mathit{Leaf}\ 4))\end{aligned}$$

In conclusion, we have just presented an instance of a zipper for the *Tree* data type, and we showed how to navigate and to express transformations on such trees. In the next section, we present a generic zipper *Haskell* library that offers this functionality to any data type.

4.2 Generic Zippers

Generic zippers are available as a *Haskell* library [40], which works for both homogeneous and heterogeneous data types. In order to show the expressiveness of this library we shall consider the heterogeneous trees consisting of nodes of two types *Prog* and *Tree*, as defined in Section 3. We define rt_1 as follows:

rt_1 :: *Prog*
rt_1 = *Root* t_1

In a straightforward functional implementation, traversing this tree would be performed by two functions: one that processes the *Prog* node and another one that recurses on *Tree* nodes. Generic zippers provide an uniform way to navigate in such heterogeneous data structures, without the need of distinguishing which type of nodes it is traversing.

In order to navigate on tree rt_1, first, it is needed to wrap it up using the provided function toZipper :: *Data* $a \Rightarrow a \rightarrow$ Zipper a. Any data type that has an instance of the *Data* and *Typeable* type classes can be navigated using generic zippers [41].

t'_1 = Zipper *Prog*
t'_1 = toZipper rt_1

This function produces an aggregate data structure which is easy to traverse and update. For example, we may move the focus of attention on t'_1 from the topmost node to the leaf containing the number 3, as follows:

leafWith3 :: *Tree*
leafWith3 = (*fromJust* . getHole . *fromJust* . down' . *fromJust* . down') t'_1

where the zipper library function down' :: Zipper $a \rightarrow$ *Maybe* (Zipper a) goes down to the leftmost (immediate) child of a node, whereas function getHole :: *Typeable* $b \Rightarrow$ Zipper $a \rightarrow$ *Maybe* b extracts the node under focus from a zipper. The library ialso ncludes a function down to go to the rightmost (immediate) child of a node, and functions up, left, and right to navigate in the directions their names indicate. All these functions wrap the result inside a *Maybe* data type to make them total. To simplify our example we are unwrapping it using the function *fromJust*.[10] The function *leafWith3* navigates from the tree's root via the shortest path to the desired leaf. Next, we navigate to that leaf after visiting all nodes in the tree as expressed by *fullVisit* in a compositional style. To avoid the use of *fromJust* and also to obtain a more natural *left-to-right* writing/reading of the navigation on trees, we can adopt an applicative functional style as expressed by *fullVisit'*, where the predefined *monadic binding* function ($\gg=$) allows the value returned from the first computation to influence the choice of the next one.

[10] Actually, we are not really checking for totality, otherwise we would have to test each function call against a set of possible results.

$$\begin{array}{ll}
\mathit{fullVisit} :: \mathit{Tree} & \mathit{fullVisit'} :: \mathit{Tree} \\
\mathit{fullVisit} = (\mathit{fromJust}\ .\ \mathsf{getHole}\ . & \mathit{fullVisit'} = \mathit{fromJust}\ \$\ \mathsf{down'}\ t'_1 \\
\qquad \mathit{fromJust}\ .\ \mathsf{left}\ . & \qquad \gg\!=\ \mathsf{down} \\
\qquad \mathit{fromJust}\ .\ \mathsf{up}\ . & \qquad \gg\!=\ \mathsf{down} \\
\qquad \mathit{fromJust}\ .\ \mathsf{left}\ . & \qquad \gg\!=\ \mathsf{left} \\
\qquad \mathit{fromJust}\ .\ \mathsf{down}\ . & \qquad \gg\!=\ \mathsf{up} \\
\qquad \mathit{fromJust}\ .\ \mathsf{down}\ . & \qquad \gg\!=\ \mathsf{left} \\
\qquad \mathit{fromJust}\ .\ \mathsf{down'}\ )\ t'_1 & \qquad \gg\!=\ \mathsf{getHole}
\end{array}$$

Obviously, not all tree paths are valid. Next, we present an unfeasible one:

$$\begin{array}{l}
\mathit{noPath} :: \mathit{Maybe\ Tree} \\
\mathit{noPath} = (\mathsf{getHole}\ .\ \mathit{fromJust}\ .\ \mathsf{left}\ .\ \mathit{fromJust}\ .\ \mathsf{down'})\ t'_1
\end{array}$$

On top of the generic zippers library, we have implemented a small set of simple combinators which allow programmers to write zipper-based functions very much like AG writers do. That is to say that, we are defining a *Haskell* embedding of AGs, fully relying on zippers, which resemble the AG notation [42, 43].

More precisely, we have defined the following combinators:

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

$$(.\$) :: \mathsf{Zipper}\ a \to \mathit{Int} \to \mathsf{Zipper}\ a$$

Thus, when using the basic combinators, *tree*.$$i$ corresponds to apply once (*fromJust* . down') *tree* and then $i - 1$ times (*fromJust* . right) to the result. For example, if we consider the following tree:

$$\begin{array}{l}
t' :: \mathsf{Zipper}\ \mathit{Tree} \\
t' = \mathsf{toZipper}\ (\mathit{Fork}\ (\mathit{Leaf}\ 3)\ (\mathit{Fork}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 4)))
\end{array}$$

then, we can access its second child as follows:

$$\begin{array}{l}
\mathit{sndChild} :: \mathit{Maybe\ Tree} \\
\mathit{sndChild} = \mathsf{getHole}\ (t'.\$2)
\end{array}$$

where $\mathit{sndChild} \equiv \mathit{Just}\ (\mathit{Fork}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 4))$, which corresponds to the same subtree as the following definition via basic combinators:

$$\mathit{sndChild} = (\mathsf{getHole}\ .\ \mathit{fromJust}\ .\ \mathsf{right}\ .\ \mathit{fromJust}\ .\ \mathsf{down'})\ t'$$

In this simple example, the second child is also the rightmost child, and it could be directly accessed via the down function, but, this is not the general case.

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

  $\mathsf{parent} :: \mathsf{Zipper}\ a \to \mathsf{Zipper}\ a$

  that corresponds to the up basic combinator.
- The combinators .\$< (left) and .\$> (right) navigate to the i^{th} sibling on the left/right of the current node:

  $(.\$<), (.\$>) :: \mathsf{Zipper}\ a \to \mathit{Int} \to \mathsf{Zipper}\ a$

- The combinator (.|) checks whether the current location is a sibling of a tree node, or not.

  $(.|) :: \mathsf{Zipper}\ a \to \mathit{Int} \to \mathit{Bool}\ .$

These combinators are generic and, obviously, can be reused to define any zipper-based program with an AG flavor. However, there is some boilerplate code that needs to be generated. Such code is specific of each of the programs we wish to express in this setting. Moreover, such boilerplate code can be automatically generated via template meta-programming [44]. Next, we discuss this code.

The generic zipper library requires that the data types defining the underlying program, have to provide instances of type class *Data* and *Typeable*. In *Haskell* this can be easily implemented via the data type **deriving** primitive. Let us return to the *repmin* problem, and extend the data types accordingly:

```
data Prog = Root Tree
          deriving (Data, Typeable)
data Tree = Leaf Int
          | Fork Tree Tree
          deriving (Data, Typeable)
```

As we have shown before, in our visual notation we structure the computations according to the constructor names. Thus, while navigating in heterogeneous trees we need to know the constructor of the node we are visiting. Thus, a new boilerplate function needs to be produced to indicate which is the constructor of the root's node of the subtree, so that we are able to do pattern matching in our zipper-based trees.

The *repmin* data types contain three constructors, that we group into a single data type, named *Constructor*, where the constructor names get as suffix the type's name (so that name conflicts are avoided). Thus, the *repmin* date type produces the following data type:

```
data Constructor = Root_Prog
                 | Fork_Tree
                 | Leaf_Tree
```

And now we can produce the function constructor that matches the root node of the given zipper tree with the constructors of the original tree. It returns one of the newly defined constructors.

```
constructor :: Typeable a ⇒ Zipper a → Constructor
constructor a = case (getHole a :: Maybe Tree) of
                   Just (Fork _ _) → Fork_Tree
                   Just (Leaf _)   → Leaf_Tree
                   otherwise       → case (getHole a :: Maybe Prog) of
                                         Just (Root _) → Root_Prog
```

In order to access tree values in the original tree, like we do when defining the local minimum in a *Leaf* node, we define the function lexeme that computes the value stored in a leaf of a tree.

```
lexeme :: Zipper a → Int
lexeme z = case (getHole z :: Maybe Tree) of
               Just (Leaf i) → i
```

Finally, to express in a more AG friendly setting the types of the zipper functions that perform the repmin tree-walk evaluation on *Prog* trees, we define the following AGTree type synonym:

```
type AGTree a = Zipper Prog → a
mkAG = toZipper
```

4.3 The Zipper-based Repmin Problem

Having defined the AG-like zipper-based combinators and after having produced the boilerplate code induced by the *repmin* data types, we can now focus on defining the zipper-based *repmin* solution that directly follows the visual definitions presented in Section 3.1 (and also the UUAG textual specification presented in Section 3.2).

First, we defined the computation of the local minimum of a tree. We use the constructor function to identify which node we are visiting in. In the case of a *Leaf* node the local minimum *lm* is the (value returned by) lexeme of that leaf node. In the case of a *Fork* node, we compute *lm* for each children, and return the minimum of the obtained values. The next zipper-based function directly follows the visual and textual notation presented in (VAG_i) and (AG_i), respectively.

(EAG_i)

```
lm :: AGTree Int
lm tree = case constructor tree of
              Leaf_Tree → lexeme tree
              Fork_Tree → min (lm (tree.$1)) (lm (tree.$2))
```

Now, the local minimum computed at the root of the tree is passed downwards as the global minimum *gm*.

(EAG_{ii_a}), (EAG_{ii_b})

```
gm :: AGTree Int
gm tree = case constructor tree of
              Root_Prog → lm (tree.$1)
              Leaf_Tree → gm (parent tree)
              Fork_Tree → gm (parent tree)
```

Finally, we compute/synthesize the desired tree:

$$
\begin{array}{ll}
 & nt :: \mathsf{AGTree}\ Tree \\
 & nt\ tree = \textbf{case}\ \mathsf{constructor}\ tree\ \textbf{of} \\
(EAG_{iii}) & \quad Root_{Prog} \rightarrow nt\ (tree.\$1) \\
 & \quad Leaf_{Tree} \rightarrow Leaf\ (gm\ tree) \\
 & \quad Fork_{Tree} \rightarrow Fork\ (nt\ (tree.\$1))\ (nt\ (tree.\$2))
\end{array}
$$

As we can cleary see, the Haskell function defined in (EAG_i), (EAG_{ii_a}), (EAG_{ii_b}), and (EAG_{iii}) are very similar to the visual and textual AG fragments $(VAG_i)/(AG_i)$, $(VAG_{ii_a})/(AG_{ii_a})$, $(VAG_{ii_b})/(AG_{ii_b})$, and $(VAG_{iii})/(AG_{iii})$, respectively.

Having defined these three fragments/functions, we now define the zipper-based *repmin* function as follows:

$$
\begin{array}{l}
repmin :: Tree \rightarrow Tree \\
repmin\ t = nt\ (\mathsf{mkAG}\ (Root\ t))
\end{array}
$$

5 Zipper-based Embedding of Attribute Grammars

In the previous section we discussed the lazy and strict version of the *repmin* program. We have also introduced zippers as a functional setting to express multiple traversal functions. Although *repmin* is widely used to introduce circular, lazy programs, it is a very simple example. Consequently, it does not show the full complexity that may arise when more elaborated and real world multiple traversal algorithms have to be expressed in a functional setting.

In this section, we consider the name analysis task as required by most modern programming languages [45, 46], which actually was in the genesis of Knuth's definition of the attribute grammar formalism [47]. Name analysis uses the scope rules of a language to associate uses of identifiers with their definitions. To make our example simple, we consider the (sub)language of **let** expressions as incorporated in most functional languages, including *Haskell*. While being a concise example, the **let** language holds central characteristics of programming languages, such as (nested) block-based structures and mandatory but unique declarations of names. In addition, the semantics of **let** does not force a *declare-before-use* discipline, meaning that a variable can be declared after its first use. The following is an example of a program in the **let** language, which corresponds to correct *Haskell* code.

$$
\begin{array}{lll}
program = \textbf{let} & a = b + 3 & \text{-- decl a , use b} \\
 & c = 2 & \text{-- decl c} \\
 & b = c \times 3 - c & \text{-- decl b , use c, use c} \\
\textbf{in} & (a + 7) \times c & \text{-- use a, use c}
\end{array}
$$

We observe that the value of program is $(a + 7) \times c$, and that a depends on b, which itself depends on c. It is important to notice that a is declared before b, a variable on which it depends. This expression evaluates to 28.

Obviously, not all **let** expressions are valid. Thus, a **let** is valid when all names used are indeed declared, and a name is not declared more than once.

We use *Haskell* comments to clarify the declaration and use of names. Next, we show an invalid **let** expression, since z is used but not declared:

```
invalid = let a = z + 3           -- decl a , use z
              c = 2               -- decl c
              b = c × 3 − c       -- decl b , use c, use c
          in  (a + 7) × c         -- use a, use c
```

Usually, **let** expressions allow nesting as shown by the following valid *Haskell* expression:

```
nestedprogram = let a  = b + 3            -- decl a , use b
                    c  = 2                -- decl c
                    w  = let c = a × b    -- decl w, decl c, use a, use b
                         in  c × b        -- use c, use b
                    b  = c × 3 − c        -- decl b , use c, use c
                in  (a + 7) × c + w       -- use a, use c, use w
```

Because we have a nested definition, then the use of a and b in the inner **let** block comes from the outer block expression since they are not defined in the inner one. Since c is defined both in the inner and in the outer block, then we must use the inner c (defined to be $a \times b$) when calculating $c \times b$, but we use the outer c (defined to be 2) when calculating $(c \times 3) - c$. Thus, the expression evaluates to 140.

Haskell **let** expressions reuse scope rules that were introduced in one of the very first high level programming languages, namely Algol 68 [48]. To simplify the specification of such scope rules, we shall consider a simpler setting where we abstract from the possibly complex right-hand side of a name definition (arithmetic expressions in our example). Thus, we wish to focus our analysis on the declaration and use of identifiers. That is to say that we express the name analysis of **let** expressions over the list of declared and used identifiers as annotated in comments of the previously shown **let** expressions. The previous three **let** expressions define the following three lists:

```
[decl a, use b            [decl a, use z            [decl a , use b
,decl c                   ,decl c                   ,decl c
,decl b, use c, use c     ,decl b, use c, use c     ,decl w, [decl c, use a, use b
,use a, use c]            ,use a  , use c]                   ,use c  , use b]
                                                    ,decl b , use c, use c
                                                    ,use a  , use c, use w]
```

In this simpler list notation, nested **let** expressions induce nested lists. We call this list-based language *Block*, which consists of a (possibly empty) list of *items*. An item is one of the following three things: a *declaration* of an identifier (such as `decl` a), the *use* of an identifier (such as `use` a), or a nested *block*.

We leave as an exercise (see Exercise 2) the definition of a (zipper-based) single traversal function to map **let** expressions to *Block* lists with the declarations and uses of identifiers.

In order to present in detail the Algol 68 scope rules, let us consider the following *Block* program:

$$\begin{aligned} example = [&\texttt{decl}\ x, \texttt{use}\ y, \texttt{use}\ x \\ ,&[\texttt{decl}\ y, \texttt{use}\ y, \texttt{use}\ w] \\ ,&\texttt{decl}\ y, \texttt{decl}\ x] \end{aligned}$$

Block does not require that declarations of names occur before their first use. Note that this is the case in the first use of y: it refers to its (later) definitions on the outermost block. Note also that the local block defines a second name y. Consequently, the second applied occurrence of y (in the local block) refers to the inner definition and not to the outer definition. In a block, however, a name may be declared once, at the most. So, the second definition of identifier x in the outermost block is invalid. Furthermore, the *Block* language requires that only defined names may be used. As a result, the applied occurrence of w in the local block is invalid, since w has no binding occurrence at all.

We aim to develop a program that analyses *Block* programs and computes a list containing the identifiers which do not obey to the scope rules of the language. Moreover, to make it easier to detect which names are being incorrectly used in a *Block* program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is $[w, x]$.

This language does not force a *declare-before-use* discipline. Consequently, a conventional implementation of the required analysis naturally leads to a program 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 checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared at the same *lexical level* (block). In that case, the name has to be added to a list reporting the detected errors. Thus, a straightforward algorithm for the scope rules processor of the *Block* language looks as follows:

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

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.

Before we implement the *Block* processor, let us present the (abstract) syntax of the language via the following *Haskell* data types:

```
data P = Root Its
       deriving (Typeable, Data)
```

$$
\begin{aligned}
&\textbf{data}\ \mathit{Its} = \mathit{ConsIts}\ \mathit{It}\ \mathit{Its}\\
&\qquad\quad\ \mid\ \mathit{NilIts}\\
&\qquad\quad\ \textbf{deriving}\ (\mathit{Typeable}, \mathit{Data})\\
&\textbf{data}\ \mathit{It} \ \ = \mathit{Decl}\ \ \mathit{Name}\\
&\qquad\quad\ \mid\ \mathit{Use}\ \ \ \mathit{Name}\\
&\qquad\quad\ \mid\ \mathit{Block}\ \mathit{Its}\\
&\qquad\quad\ \textbf{deriving}\ (\mathit{Typeable}, \mathit{Data})
\end{aligned}
$$

where P is the root symbol consisting of items *Its*, that is a user-defined list of item *It*. An *It* is a decl or a use of a *Name*, or a nested block. *Name* is a type synonym for *String*.

According to the two traversal algorithm presented before we aim at implementing the following two functions:

$$
\begin{aligned}
&\mathit{environment} :: P \to (\mathit{Env}, \mathit{Errors})\\
&\mathit{errors} \qquad\ \ :: P \to \mathit{Env} \to \mathit{Errors} \to \mathit{Errors}
\end{aligned}
$$

where the types *Env* and *Errors* represent lists of declared names and errors, respectively. They are defined as follows:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Env} \quad\ = [\mathit{Name}]\\
&\textbf{type}\ \mathit{Errors} = [\mathit{Name}]
\end{aligned}
$$

The function *environment* computes (synthesizes) the total environment of a block, while computing errors due to duplicated declarations. This function can be easily expressed in two ways: in a bottom-up strategy by synthesizing the environment, or by starting with an empty list where it accumulates the declarations while traversing a block. These two strategies are displayed in Fig. 2.

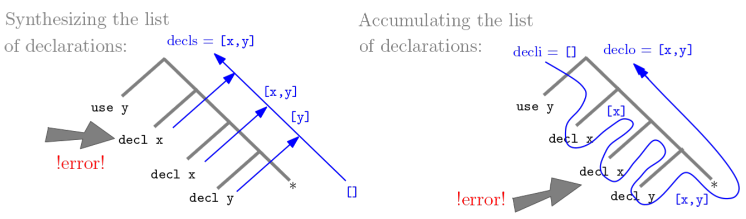


Fig. 2: Synthesizing versus accumulating the environment.

As we observe in Fig. 2, the use of an accumulator does report the second declaration as duplicated and not the first one. Thus, we refine the types of our *Block* processor functions in order to use an accumulator, as follows:

$$
\begin{aligned}
&\mathit{environment} :: P \to \mathit{Env} \to (\mathit{Env}, \mathit{Errors})\\
&\mathit{errors} \qquad\ \ :: P \to \mathit{Env} \to \mathit{Errors} \to \mathit{Errors}
\end{aligned}
$$

Moreover, we have also to distinguish definitions of the same name at different block levels: inner blocks may re-define names introduced by their outer ones.

Thus, we tuple the names with the level of the block where they are defined. As a result, the environment has now type

type *Env* = [(*Name*, *Int*)]

Fig. 3 displays the *Block* input, where the initial, and final (accumulated) list of declarations is shown for each block. They are named *dcli* and *dclo*, receptively. Identifier names are paired with the level where they are introduced.

```
                dcli = []
                dclo = [(x,0),(y,0)]
example  =  [    decl x , use y , use x ,
                    dcli = [(x,0),(y,0)]
                    dclo = [(y,1),(x,0),(y,0)]
                  [ decl y , use y , use w ] ,

                  decl y , decl x
            ]
```

Fig. 3: Block example where blocks are labeled with their initial and final accumulated list of declarations.

As we can see, while the outer block *starts*, *i.e.*inherits an empty environment, the inner block starts building its environment from the synthesized environment of the outer block. The *dcli* of the inner block is the *dclo* of its outer block. In other words, we say that inner blocks inherit the environment of their outer ones.

We may also notice that only in the second traversal of an outer block the inherited list of declarations needed by nested blocks is known. That is to say that only in the second traversal to an outer block, the inner blocks are traversed for the first time! As a result, traversal functions *environment* and *errors* are intermingled and it is not straightforward how to schedule them.

The reader may have already noticed another problem in these two functions: *environment* produces the list of invalid declarations, that is given as argument to *errors*. The two lists have to be *combined* so that the final list of errors follows the input program! So the question is *how can we combine the errors computed in the first traversal (i.e., duplicated definitions), with the errors computed in the second one (i.e., invalid uses)?* In an imperative setting the values computed in the one traversal are usually stored in the original tree, as side effects, so that following traversals may use them. Thus, the duplicated declarations detected in the first traversal can easily be stored in the tree node, where they occur, and the second traversal just collects them in the order they occur in the tree/input. In a strict, purely functional setting this is not possible and an additional intermediate data structure has to be defined and constructed in order to pass such values to following traversal functions, and, in this way, glue them together [49]. As a consequence, the types of our functions would look as follows:

$$\begin{aligned} &environment :: P \;\to Env \to (P_2, Env)\\ &errors \qquad\;\;\; :: P_2 \to Env \to Errors \end{aligned}$$

That is to say, that function *environment* should return a new version of the tree with the additional information stored in the nodes.

Indeed, the implementation of the straightforward algorithm of the *Block* processor in a (non-lazy) functional setting is a complex task: complex traversal functions need to be scheduled, additional gluing data types may have to be defined, and intermediate values need to be explicitly passed between traversals. A circular, lazy solution can also be defined which overcomes those issues, but there is a price to pay: first, it relies on the counter-intuitive circular definitions that are also difficult to define. Secondly, all aspects of the *Block* language are tupled into a single function definition, and, as consequence, such a lack of modularity makes it hard to write, to understand, and to (incrementally) extend the language with new features.

We will present the complex strict and lazy implementations of the *Block* processors in sections 6.2 and 6.1, respectively. Now, let us provide an elegant, concise, and modular specification of the *Block* language via our zipper-based embedding of AGs, which does overcome the limitations of lazy and non-lazy solutions.

5.1 The Zipper-based Block Program

In this section we will present the full definition of the zipper-based *Block* program. We will present the visual AG notation in detail, before we define its AG-based zipper counterpart. We omit the boilerplate code induced by the heterogeneous *Block* data types, namely the data type constructor *Constructor* and the functions constructor and lexeme because they directly follow from the *Block* data types, and, as mentioned before, they can be generated via template *Haskell* [44].

Computing the Environment of Block: Let us start by defining an attribute, named *dclo*, that synthesizes the list of declared names of a block. As we show in Fig. 4, the only constructor that contributes to the synthesized list of declarations is *Decl*, where the defined *Name* is tupled with the lexical level of the block it occurs on. This pair is then added to the inherited list of declarations *dcli*: the accumulated declarations thus far. As we can see in the visual notation in the other constructors the value of *dclo* is just a copy of *dcli* or the *dclo* of a child (*ConsIts*).

The corresponding zipper-based function *dclo* closely follows this visual AG notation.

```
dclo :: AGTree Env
dclo t = case constructor t of
              NilIts_Its   → dcli t
              ConsIts_Its  → dclo (t.$2)
              Decl_It      → (lexeme t, lev t) : (dcli t)
              Use_It       → dcli t
              Block_It     → dcli t
```

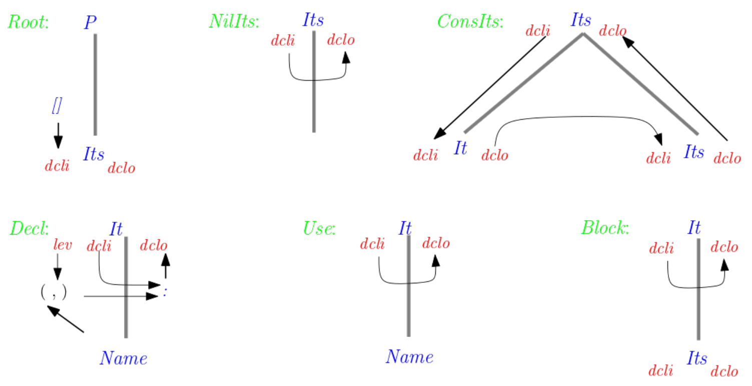


Fig. 4: Attribute equations defining the accumulation of declared names in the environment.

The attribute (function) *dclo* depends on the value (call) of (to) *dcli*. Thus, it defines an inherited attribute, which is typically inherited from the parent of its (sub)tree.

Let us define now the equations of attribute *dcli*. As shown in Fig. 4, the initial list of declarations *dcli* at the *Root* of the tree is the empty list, since the outermost block is context-free. Let us consider now subtrees of type *It*. Such subtrees only occur as the first child of a *ConsIts* node, and in this case, *dcli* is defined as a copy of *dcli* of the parent. Subtrees of type *Its*, however, may inherit *dcli* in three different ways: when the node was constructed with *Root*, then its value is the empty list. If the node corresponds to a nested *Block*, then the initial list of declarations is the environment of the outer block (represented by the attribute *env*). Finally, if the node was constructed with *ConsIts*, then *dcli* gets a copy of the declarations synthesized by its left sibling. These equations are shown in Fig. 5.

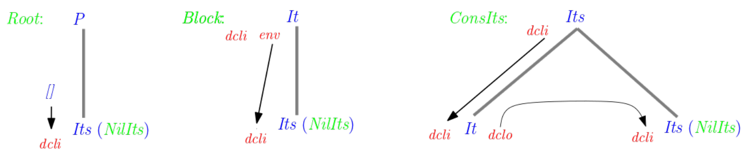


Fig. 5: Passing down the initial list of declarations.

When visiting nodes of type *Its* (recall that *Its* type constructors are *NilIts* and *ConsIts*), the zipper function has to consider the three alternatives where those subtrees inherit *dcli* from. Thus, the zipper-based function *dcli* is expressed as follows:

```
dcli :: AGTree Env
dcli t = case constructor t of
           NilIts_Its  → case (constructor $ parent t) of
                            ConsIts_Its → dclo (t.$<1)
                            Block_It    → env (parent t)
                            Root_P      → [ ]
           ConsIts_Its → case (constructor $ parent t) of
                            ConsIts_Its → dclo (t.$<1)
                            Block_It    → env (parent t)
                            Root_P      → [ ]
           Block_It    → dcli (parent t)
           Use_It      → dcli (parent t)
           Decl_It     → dcli (parent t)
```

In the definition of *dcli*, we explicitly include all constructors and respective equations that follow from the visual AG notation. It should be noticed that the two branches $NilIts_{Its}$ and $ConsIts_{Its}$ of the case statement are the same, and such duplicated code can be eliminated by using the default alternative of the case statement. In the next definitions we will structure our zipper-based functions in this way.

As we can see in visual notation of the *Decl* constructor (see Fig. 4), defined names are tupled with the lexical level where they occur, before added to *dclo*. In constructor *ConsIts*, the second child (of type *Its*) inherits the list of declarations *dcli* from the total declarations *dclo* of its left sibling. The left sibling is also the first child of the parent, thus both definitions are equivalent *dclo* $(t.\${<}1) \equiv$ *dclo* ((parent t).$1) and could be used in our definition of *dcli*.

The inherited attribute *lev* (of type *Int*) will be used to assign a level to each block. The outermost block has level 0 and in almost all cases the level is copied/passed downwards. The exception is constructor *Block*, since a nested block is one level higher than its outer one. The equations of attribute *lev* are visually defined in Fig. 6.

The zipper function *lev* has also to consider the context of the occurrence of *Its* subtrees and their three different type constructors. Once again, this function closely resembles the AG definition.

```
lev :: AGTree Int
lev t = case constructor t of
          Block_It → lev (parent t)
          Use_It   → lev (parent t)
          Decl_It  → lev (parent t)
          _        → case (constructor $ parent t) of
                        Block_It    → (lev (parent t)) + 1
                        ConsIts_Its → lev (parent t)
                        Root_P      → 0
```

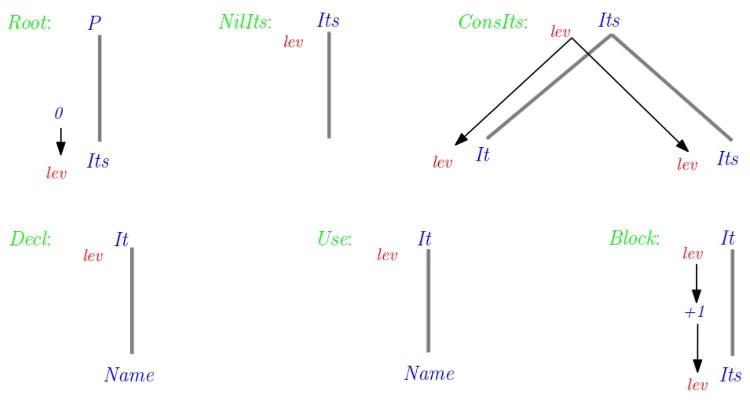


Fig. 6: Attribute equations defining inherited attribute *lev*.

Distributing the Environment of *Block*: The previous AG fragments, expressed as zipper-based functions, concerned the computation of the environment (*i.e.*, available names) of a block. Fig. 7 defines an inherited attribute *env*, of type *Env*, to distribute it to all the items of a block. In the *Root* and *Block* constructors, this environment corresponds to the list of accumulated declarations.

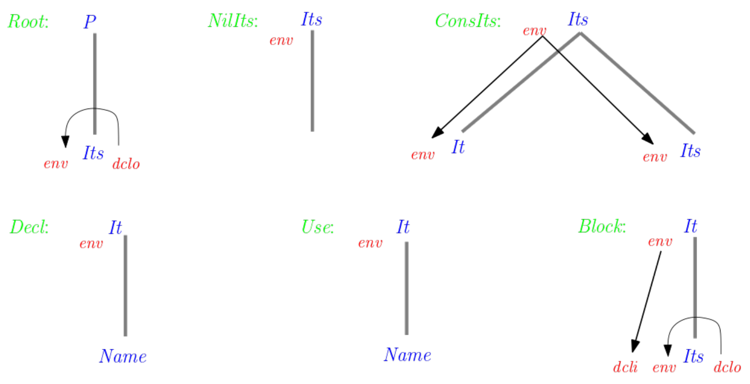


Fig. 7: Attribute equations defining the total environment of *Block*.

And, this is expressed as the *env* zipper function:

```
env :: AGTree Env
env t = case constructor t of
          Block_It → env (parent t)
          Use_It   → env (parent t)
          Decl_It  → env (parent t)
          _        → case (constructor $ parent t) of
                       Block_It      → dclo t
                       ConsIts_Its   → env (parent t)
                       Root_P        → dclo t
```

Computing the Errors of *Block*: We are now able to synthesize the desired list of errors that follow the sequential structure of the input. There are two constructors that contribute to the list of errors: *Decl* and *Use*. In a declaration, the *Name* must not be in the accumulated list of declarations at the same lexical level (recall that it may be defined in an outer level). In the use of a *Name*, it must be in the (full) environment of its block. The AG fragment that synthesizes the *errors* is displayed in Fig. 8.

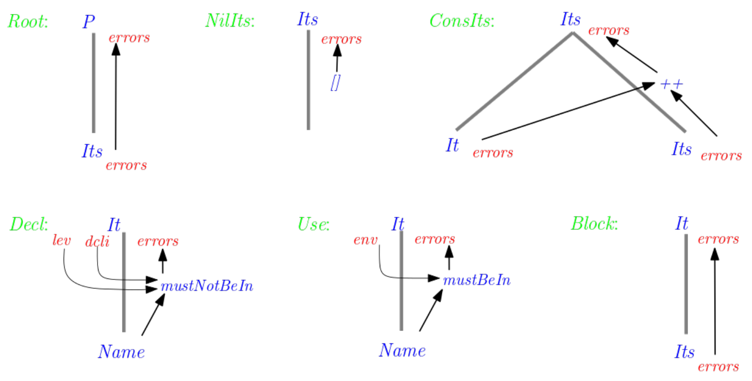


Fig. 8: Attribute equations synthesizing the list of errors.

where *mustBeIn* and *mustNotBeIn* are (semantic) functions that define simple lookup operations on lists.[11] Next, *mustBeIn* is a trivial *Haskell* (well-formed) lookup function expressed by the (higher-order) function *filter*, as follows:

[11] Semantic functions are usually expressed outside the AG formalism, usually in a functional language (this is the case in most AG-based systems, namely [12, 13, 50, 51]). Because they are not part of the AG formalism, they are not analyzed by the powerful static analysis AG techniques, which provide important guarantees, namely the existence of an attribute evaluation order. As a consequence, ill-formed semantic functions may induce non termination of the AG implementations. Such functions may, however, be defined within the AG formalism, via its higher-order extension as proposed in [52, 53].

```
mustBeIn :: Name → Env → Errors
mustBeIn n e = if null (filter ((≡ n) . fst) e) then [n] else []
```

whereas *mustNotBeIn* is easily expressed via the predefined *elem* function:

```
mustNotBeIn :: (Name, Int) → Env → Errors
mustNotBeIn p e = if p ∈ e then [fst p] else []
```

As expected, the zipper-based function directly follows the AG definition.

```
errors :: AGTree Errors
errors t = case constructor t of
              Root_P       → errors (t.$1)
              NilIts_Its   → []
              ConsIts_Its  → (errors (t.$1)) ++ (errors (t.$2))
              Block_It     → errors (t.$1)
              Use_It       → (lexeme t) `mustBeIn` (env t)
              Decl_It      → (lexeme t, lev t) `mustNotBeIn` (dcli t)
```

Finally, we can define our *Block* program as follows

```
block :: P → Errors
block p = errors (mkAG p)
```

As expected, running *block* with our input *example* produces the desired list of errors `["w", "x"]`.

In this document, we only have defined the name analysis task of the *Block*. In [21], we defined in this AG style of programming the zipper-based solution to compute the value of (valid) **let** expressions. It is expressed as a zipper-based fixpoint attribute evaluation strategy, which shows that our embedding does support the class of *circular, but well-defined, attribute grammars* [54].

6 Purely Functional Implementation of Attribute Grammars

In the previous section, we presented the full specification of an attribute grammar using a visual notation. We have also expressed such AG straightforwardly as a *Haskell* program, via the introduced zipper-based AG combinators. As a result, the zipper-based program can be directly executed, and it does produces the desired list of errors.

Traditionally, however, attribute grammar specifications are compiled into a target implementation, the so-called attribute evaluator, before they can be executed. Attribute grammar based systems, like [13, 20, 50, 19, 55], use powerful static analysis techniques to generate very efficient attribute evaluators. In fact, the attribute grammar community developed advanced scheduling algorithms that based on the dependencies induced by the attribute equations of the grammar, statically compute an evaluation order of attributes. Indeed, the pioneer work of Kastens on ordered attribute grammars [56] proposed a scheduling algorithm which is not only able to detect circularities in the AG, but also adequate to produce strict attribute evaluators for the so-called class of *ordered attribute grammars*. Kastens scheduling algorithm is pessimistic in the sense that some not circular attribute grammars are reported as circular ones. As a consequence, it fails to order their attribute dependencies and, thus, to derive an evaluation order for them. However, when an attribute grammar is reported as *ordered*, then an evaluation order exists which guarantees the termination of the resulting attribute evaluator.[12]

Both the *repmin* and *Block* aGs are *ordered*. Actually, the two traversal solution of *repmin* we presented in Section 1 corresponds to the attribute evaluator that Kastens algorithm generates.

6.1 The Strict, Two Traversal Block Evaluator

Because the *Block* AG is an *ordered attribute grammar*, Kastens scheduling algorithm infers an order of attribute evaluation, which can be implemented as a strict, purely functional attribute evaluator [49]. In fact, such evaluator corresponds to a multiple traversal solution that a functional programmer would write by hand. Next, we discuss this solution as generated by the *Lrc* system from a textual version (*i.e.* in the AG notation used by *Lrc*) of the *Block* AG.

Based on the attribute dependencies occurring in the productions of the grammar, Kastens scheduling algorithm produces a fixed set of abstract computations, the so-called *visit-sequences*. They abstractly describe which computations have to be performed on every visit of the evaluator to a particular type of nodes in the tree: the instances of the constructor. In such evaluators, the number of visits to a non-terminal (*i.e.*, type) is fixed and it is computed statically. For each visit, it is also defined which attributes are computed. For the *Block* AG, Kastens algorithm, as expected, infers two visits to non-terminals *Its* and *It*, and it defines which attributes are computed in each visit as shown next.

```
visit 1 : inh dcli, lev
          syn dclo
visit 2 : inh env
          syn errors
```

The nonterminal *P* has a single visit to synthesize attribute *errors*.

Visit-sequences can be expressed as a strict, multiple traversal functional program by first computing the attributes that are defined in a visit and used in a following one, the so-called *inter-traversal attribute dependencies* [49]. Under an imperative setting, such values can be stored in the tree nodes where they are defined and later used [57]. However, in a functional setting they have to be explicitly passed via additional data structures [49].

[12] Assuming that the semantic functions defined in the AG, which are external to the formalism (like *mustBeIn* and *mustNotBeIn* in the *Block* AG), are well-formed and also do terminate.

The following data types are needed to glue the two functions that perform those traversals.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Its}_2 = \mathit{ConsIts}_2\ \mathit{It}_2\ \mathit{Its}_2\\
&\qquad\quad\ \mid\ \mathit{NilIts}_2\\
&\textbf{data}\ \mathit{It}_2\ \ = \mathit{Block}_2\quad \mathit{Its}\ \mathit{Int}\\
&\qquad\quad\ \mid\ \mathit{Decl}_2\quad \mathit{Errors}\\
&\qquad\quad\ \mid\ \mathit{Use}_2\quad\ \mathit{Name}
\end{aligned}
$$

Kastens scheduling induces two inter-traversal attribute dependencies: the *errors* computed in the first traversal to *Decl* nodes, needs to be passed to the second one. [13] Since the evaluator descends to its inner blocks, only in the second visit to a *Block* node, the attribute *lev* (of type *Int* and that is available in the first traversal) also needs to be passed on to the second traversal. Moreover, inner blocks are needed in the second traversal, and such trees (types) have to remain unchanged in the gluing tree.

The *block* program is now implemented based on the fixed, per constructor, visit-sequences and the gluing data types induced by the inter-traversal dependencies. Thus, two functions traverse *Its* trees: the first traversal $\mathit{eval\_Its}_1$ tuples the expected result (*i.e.*, *dclo*) with the instance of the gluing data type, which is the tree that is the argument (and thus traversed) by the second traversal function $\mathit{eval\_Its}_1$.

$$
\begin{aligned}
&\mathit{eval\_P}\quad\ \ :: P \to \mathit{Errors}\\
&\mathit{eval\_P}\quad\ \ (\mathit{Root}\ \mathit{its}) = \mathit{errors}\\
&\quad \textbf{where}\ (\mathit{its}_2, \mathit{dclo}) = \mathit{eval\_Its}_1\ \mathit{its}\ [\,]\ 0\\
&\qquad\qquad\ \ \mathit{errors}\qquad\ = \mathit{eval\_Its}_2\ \mathit{its}_2\ \mathit{dclo}
\end{aligned}
$$

In the first visit to the original *Block* tree, the values of *errors* and *lev* are defined, and are stored in the gluing tree.

$$
\begin{aligned}
&\mathit{eval\_Its}_1\ :: \mathit{Its} \to \mathit{Env} \to \mathit{Int} \to (\mathit{Its}_2, \mathit{Env})\\
&\mathit{eval\_Its}_1\ (\mathit{ConsIts}\ \mathit{it}\ \mathit{its})\ \mathit{dcli}\ \mathit{lev} = (\mathit{ConsIts}_2\ \mathit{it}_2\ \mathit{its}_2, \mathit{dclo}_2)\\
&\quad \textbf{where}\ (\mathit{it}_2, \mathit{dclo})\quad\ = \mathit{eval\_It}_1\ \mathit{it}\ \mathit{dcli}\ \mathit{lev}\\
&\qquad\qquad\ \ (\mathit{its}_2, \mathit{dclo}_2) = \mathit{eval\_Its}_1\ \mathit{its}\ \mathit{dclo}\ \mathit{lev}\\
&\mathit{eval\_Its}_1\ \mathit{NilIts}\ \mathit{dcli}\ \mathit{lev}\qquad\qquad\ = (\mathit{NilIts}_2, \mathit{dcli})\\
&\mathit{eval\_It}_1\ \ :: \mathit{It} \to \mathit{Env} \to \mathit{Int} \to (\mathit{It}_2, \mathit{Env})\\
&\mathit{eval\_It}_1\ \ (\mathit{Block}\ \mathit{its})\ \mathit{dcli}\ \mathit{lev} = (\mathit{Block}_2\ \mathit{its}\ (\boxed{\mathit{lev} + 1}), \mathit{dcli})\quad \text{-- defining lev}\\
&\mathit{eval\_It}_1\ \ (\mathit{Decl}\ n)\ \mathit{dcli}\ \mathit{lev}\quad = (\mathit{Decl}_2\ \mathit{errors}, \mathit{dclo})
\end{aligned}
$$

[13] *Lrc* uses a slightly improved version of Kastens algorithm. For example, Kastens algorithm schedules the evaluation of attribute *errors* to the second traversal. In *Decl* productions, however, *Lrc* schedules it to the first traversal since all attributes it depends on are already available. Thus, it does not have to pass attributes *lev*, *dcli* and the declared identifier to the second traversal. Because only the attribute *errors* is stored in the new tree, it decreases memory consumption. The details of this scheduling algorithm are included in Chapter 3 of [49].

$$\begin{array}{lll}
\quad \textbf{where}\ \mathit{dclo} & = (n, \mathit{lev}) : \mathit{dcli} & \\
\quad \phantom{\textbf{where}}\ \boxed{\mathit{errors}} & = (n, \mathit{lev})\ `\mathit{mustNotBeIn}`\ \mathit{dcli} & \text{-- defining errors} \\
\mathit{eval\_It_1}\ (\mathit{Use}\ n)\ \mathit{dcli}\ \mathit{lev} & = (\mathit{Use_2}\ n, \mathit{dcli}) &
\end{array}$$

Finally, in the second traversal, the evaluator finds all values it needs in the tree nodes of the tree constructed in the first traversal.

$$\begin{array}{lll}
\mathit{eval\_Its_2}\ :: \mathit{Its_2} \to \mathit{Env} \to \mathit{Errors} & & \\
\mathit{eval\_Its_2}\ (\mathit{ConsIts_2}\ \mathit{it_2}\ \mathit{its_2})\ \mathit{env} = \mathit{errors} & & \\
\quad \textbf{where}\ \mathit{errors}_1 = \mathit{eval\_It_2}\ \mathit{it_2}\ \mathit{env} & & \\
\quad \phantom{\textbf{where}}\ \mathit{errors}_2 = \mathit{eval\_Its_2}\ \mathit{its_2}\ \mathit{env} & & \\
\quad \phantom{\textbf{where}}\ \mathit{errors}\ \ = \mathit{errors}_1 +\!\!+\ \mathit{errors}_2 & & \\
\mathit{eval\_Its_2}\ \mathit{NilIts_2}\ \mathit{x\_env}\qquad = [\,] & & \\
\mathit{eval\_It_2}\ :: \mathit{It_2} \to \mathit{Env} \to \mathit{Errors} & & \\
\mathit{eval\_It_2}\ (\mathit{Block_2}\ \mathit{its}\ \mathit{lev})\ \mathit{env} = \mathit{errors} & & \\
\quad \textbf{where}\ (\mathit{its_2}, \mathit{dclo}) = \mathit{eval\_Its_1}\ \mathit{its}\ \mathit{env}\ \boxed{\mathit{lev}} & & \text{-- using lev} \\
\quad \phantom{\textbf{where}}\ \mathit{errors} = \mathit{eval\_Its_2}\ \mathit{its_2}\ \mathit{dclo} & & \\
\mathit{eval\_It_2}\ (\mathit{Decl_2}\ \mathit{errors})\ \mathit{env}\ \ = \boxed{\mathit{errors}} & & \text{-- using errors} \\
\mathit{eval\_It_2}\ (\mathit{Use_2}\ n)\ \mathit{env}\qquad = \mathit{errors} & & \\
\quad \textbf{where}\ \mathit{errors} = n\ `\mathit{mustBeIn}`\ \mathit{env} & &
\end{array}$$

6.2 The Circular, Lazy Block Evaluator

The previous *Block* evaluator has a key advantage: it guarantees termination, while a hand-written circular, lazy program does not! As we have mentioned before, it is possible and easy to write a truly circular definition, and as a consequence the produced circular lazy program will loop and will fail to terminate. The *Block* AG, however, is ordered, and it can be directly defined as a well-formed circular, lazy program: it does terminate, as the strict one.

$$\begin{array}{l}
\mathit{eval\_P}\quad :: P \to \mathit{Errors} \\
\mathit{eval\_P}\quad (\mathit{Root}\ \mathit{its}) = \mathit{errors} \\
\quad \textbf{where}\ (\boxed{\mathit{dclo}}, \mathit{errors}) = \mathit{eval\_Its}\ \mathit{its}\ [\,]\ 0\ \boxed{\mathit{dclo}} \\
\mathit{eval\_Its}\quad :: \mathit{Its} \to \mathit{Env} \to \mathit{Int} \to \mathit{Env} \to (\mathit{Env}, \mathit{Errors}) \\
\mathit{eval\_Its}\quad (\mathit{ConsIts}\ \mathit{it}\ \mathit{its})\ \mathit{dcli}\ \mathit{lev}\ \mathit{env} = (\mathit{dclo}_2, \mathit{errors}) \\
\quad \textbf{where}\ (\mathit{dclo}, \mathit{errors}_2)\ \ = \mathit{eval\_It}\ \ \mathit{it}\ \ \mathit{dcli}\ \ \mathit{lev}\ \mathit{env} \\
\quad \phantom{\textbf{where}}\ (\mathit{dclo}_2, \mathit{errors}_3) = \mathit{eval\_Its}\ \mathit{its}\ \mathit{dclo}\ \mathit{lev}\ \mathit{env} \\
\quad \phantom{\textbf{where}}\ \mathit{errors}\qquad\quad = \mathit{errors}_2 +\!\!+\ \mathit{errors}_3 \\
\mathit{eval\_Its}\quad \mathit{NilIts}\ \mathit{dcli}\ \mathit{lev}\ \mathit{env}\qquad\quad = (\mathit{dcli}, [\,]) \\
\mathit{eval\_It}\quad :: \mathit{It} \to \mathit{Env} \to \mathit{Int} \to \mathit{Env} \to (\mathit{Env}, \mathit{Errors}) \\
\mathit{eval\_It}\quad (\mathit{Block}\ \mathit{its})\ \mathit{dcli}\ \mathit{lev}\ \mathit{env} = (\mathit{dcli}, \mathit{errors}) \\
\quad \textbf{where}\ (\boxed{\mathit{dclo}}, \mathit{errors}) = \mathit{eval\_Its}\ \mathit{its}\ \mathit{env}\ (\mathit{lev} + 1)\ \boxed{\mathit{dclo}} \\
\mathit{eval\_It}\quad (\mathit{Decl}\ n)\ \mathit{dcli}\ \mathit{lev}\ \mathit{env}\quad = (\mathit{dclo}, \mathit{errors})
\end{array}$$

```
    where dclo   = (n, lev) : dcli
          errors = (n, lev) `mustNotBeIn` dcli
  eval_It  (Use n) dcli lev env   = (dcli, errors)
    where errors = n `mustBeIn` env
```

It is worthwhile to note how the AG scheduling algorithm statically broke up pseudo-circular definitions into multiple traversal functions. For example, the circular definition in this version of *eval_P*

$(\boxed{dclo}, errors) = eval\_Its\ its\ [\,]\ 0\ \boxed{dclo}$

was unraveled, and it can be strictly evaluated by the two functions calls occurring in the body of the previous multiple traversal definition of *eval_P*, as follows:

$errors = uncurry\ eval\_Its_2\ (eval\_Its_1\ its\ [\,]\ 0)$

The transformation from pseudo-circular definitions to strict multiple traversal ones, is presented in a formal setting in [35]. Alternatively, [32, 33, 58] present shortcut fusion rules for the derivation of circular definitions from strict, multiple traversal ones.

6.3 Attribute Grammar Optimization

An attribute grammar programmer with a keen eye for program optimization, may have already noticed that our attribute grammar, and its zipper-based functional definition, can be further optimized. If we look more carefully to the inherited attribute *env*, we can see that it is just a copy of the synthesized attribute *dclo* defined in the *NilIts* constructor, *i.e.*, at the end of the block. Looking at the AG fragment, shown in Fig. 9, where *dcli* and *dclo* are defined, we realize that the two attribute occurrences of *dclo* (of *Its*) actually share the value of *env*. This value is then passed downwards as the inherited *env* of *It* and *Its*, without changing it.

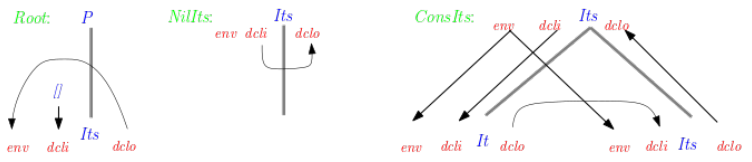


Fig. 9: Passing down the accummulated list of declarations at the root's node.

In fact, instead of passing *dclo* upwards all way to the *Root* where it is copied to *env*, we can take a shortcut and pass it immediately at the *ConsIts* constructor, as displayed in Fig. 10.

The attribute *env* is now an inherited attribute of *It* only, and it is always defined as the *dclo* of its right sibling. Thus, the zipper-based function *env* is now redefined as follows:

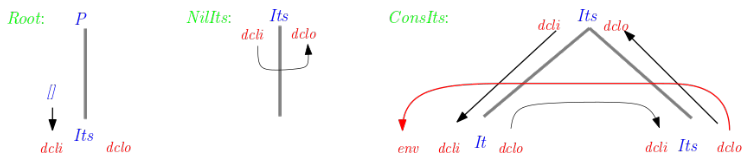


Fig. 10: Short-cutting the passing down of the accummulated list of declarations.

$env :: \mathsf{AGTree}\ Env$
$env\ t = dclo\ (t.\${>}1)$ -- dclo ((parent t).$2)

This new version of the *Block* AG is also an *ordered attribute grammar*. However, it should be noticed that it includes a pattern of attribute propagation that very often induces the pessimistic Kastens algorithm to report a (non-existent) circularity, the so-called *type III circularity* [56, 37, 49]. This type of circularity is well explained on page 31 of [37]: *"Perhaps the most common way type III circularities arise is to write a grammar in which there are two disjoint threadings of attribute dependencies through the same productions, one threaded left-to-right and the other threaded right-to-left; such a grammar is not circular but has a type III circularity"*.[14]

As the reader may have noticed, this is exactly the case of the dependencies shown in production/constructor *ConsIts*! For this particular dependency, Kastens algorithm does compute an evaluation order which statically guarantees termination. However, AG programmers do need to be careful when expressing such attribute propagation in a production, to avoid false circularities being reported, and consequently no evaluators are produced!

Next, we present the strict, multiple traversal evaluator produced for this AG.

```
$eval\_P :: P \to Errors$
$eval\_P\ (Root\ its) = errors$
   where $(dclo, errors) = eval\_Its\ its\ [\,]\ 0$

$eval\_Its :: Its \to Env \to Int \to (Env, Errors)$
$eval\_Its\ (ConsIts\ it\ its)\ dcli\ lev = (dclo_2, errors)$
   where $(it_2, dclo)$          $= eval\_It_1\ it\ dcli\ lev$
         $(dclo_2, errors_2)$    $= eval\_Its\ its\ dclo\ lev$
         $errors_1$              $= eval\_It_2\ it_2\ dclo_2$
         $errors$                $= errors_1 +\!\!+\ errors_2$
$eval\_Its\ NilIts\ dcli\ lev = (dcli, [\,])$
```

Looking at constructor *ConsIts*, we realize that *Its* subtrees are now processed in a single traversal: given the accumulated declarations, the total list is

[14] Attribute grammars whose type III circularities can be eliminated are called *arranged orderly attribute grammars* [56]. For more details of all type of circularities during the Kastens construction, the reader is referred to [56] or Chapter 3 of [49].

computed. Subtrees rooted *It*, however, need two traversals: A first one to compute its total list of declarations (to pass it to the sibling *Its* as the accumulated list), and a second traversal where the total list of declarations of the sibling is given as the inherited environment. To pass the *errors* detected in the first traversal to the second one, a gluing data type is produced. It corresponds to the data type It_2 presented before.

```
eval_It1 :: It → Env → Int → (It2, Env)
eval_It1 (Block its) dcli lev = (Block2 its (lev + 1), dcli)
eval_It1 (Use n) dcli lev     = (Use2 n, dcli)
eval_It1 (Decl n) dcli lev    = (Decl2 errors, dclo)
  where dclo   = (n, lev) : dcli
        errors = (n, lev) `mustNotBeIn` dcli

eval_It2 :: It2 → Env → Errors
eval_It2 (Block2 its lev) env = errors
  where (dclo, errors) = eval_Its its env lev
eval_It2 (Decl2 errors) env  = errors
eval_It2 (Use2 n) env        = errors
  where errors = n `mustBeIn` env
```

6.4 The Indirect Circular, Lazy Block Evaluator

An ordered AG or an AG where a type III circularity was reported by Kastens algorithm, are guaranteed to be not circular, and consequently can be easily implemented in a lazy functional setting. The corresponding lazy evaluator is presented next. This evaluator does not have a direct circular definition, as we have seen previously, but it does include an *indirect circular definition* (shown by the framed variables).

```
eval_P    :: P → Errors
eval_P    (Root its) = errors
  where (dclo, errors) = eval_Its its [] 0

eval_Its :: Its → Env → Int → (Env, Errors)
eval_Its (ConsIts it its) dcli lev = (dclo2, errors)
  where ([dclo], errors2)  = eval_It  it  dcli lev [dclo2]
        ([dclo2], errors3) = eval_Its its [dclo] lev
        errors             = errors2 ++ errors3
eval_Its NilIts dcli lev            = (dcli, [])

eval_It :: It → Env → Int → Env → (Env, Errors)
eval_It (Block its) dcli lev env = (dcli, errors)
  where (dclo, errors) = eval_Its its env (lev + 1)

eval_It (Decl n) dcli lev env   = (dclo, errors)
  where dclo   = (n, lev) : dcli
        errors = (n, lev) `mustNotBeIn` dcli
```

```
eval_It (Use n) dcli lev env    = (dcli, errors)
  where errors = n `mustBeIn` env
```

6.5 The Strict, Glue-Free Block Program

Let us look in detail to the strict, optimized evaluator shown in Section 6.3, more precisely, let us analyze how the list of errors is being computed. If we look at the function calls in the alternative of constructor *ConsIts* of function *eval_Its*, we see the following: a first visit is performed to the *It* subtree, to accumulate the declaration and to compute an eventual duplicated declaration (which is "stored" in the gluing tree). Next, the *Its* subtree is visited to continue accumulating the declarations and computing occurring errors. Finally, a second visit to *It* is done to compute invalid uses.

A functional programmer with a keen eye for program optimization, may have already noticed that the gluing data structure is not really necessary in this solution, and, thus, can be eliminated. Note that the first visit to *It* subtrees can return the *errors* due to duplicated declarations, whilst the second visit returns the *errors* due to invalid uses. Moreover, the visit to *Its* produces the *errors* occurring deeper in the tree. Then, the three (lists of) *errors* can be concatenated in the desired order. Next, we present this solution for the *block* program:

```
eval_P :: P → Errors
eval_P (Root its) = errors
  where (dclo, errors) = eval_Its its [] 0

eval_Its :: Its → Env → Int → (Env, Errors)
eval_Its (ConsIts it its) dcli lev = (dclo₂, errors)
  where (dclo, [errors₁]) = eval_It₁ it dcli  lev
        (dclo₂, errors₃)  = eval_Its its dclo lev
        [errors₂]         = eval_It₂ it dclo₂ lev
        errors            = errors₁ ++ errors₂ ++ errors₃
eval_Its NilIts dcli lev            = (dcli, [])

eval_It₁ :: It → Env → Int → (Env, Errors)
eval_It₁ (Block its) dcli lev = (dcli, [])
eval_It₁ (Use n)     dcli lev = (dcli, [])
eval_It₁ (Decl n)    dcli lev = (dclo, errors)
  where dclo   = (n, lev) : dcli
        errors = (n, lev) `mustNotBeIn` dcli
eval_It₂ :: It → Env → Int → Errors
eval_It₂ (Block its) env lev = errors
  where (dclo, errors) = eval_Its its env (lev + 1)
eval_It₂ (Decl n)    env lev = []
eval_It₂ (Use n)     env lev = errors
  where errors = n `mustBeIn` env
```

Although this solution performs two traversals, it does not rely on any additional data structure. It should be noticed that this is not the straightforward solution a functional programmer would immediately write. Function *eval_Its* visits twice the same *It* subtree to compute possible errors: the first visit, using the accumulated list of declarations, to compute duplicated declarations, and a second visit, using the total environment, to compute invalid uses. Although it visits *It* trees twice, only in one of the visits an error can occur: either a duplication or an invalid use! Between those visits, it needs to go to the *Its* subtree to compute the total environment, as well as the errors occurring latter in the input tree/program. Finally, the three computed errors are concatenated in the right order.

This solution can be further transformed by relying on the powerful *Haskell* pattern matching mechanism: alternatives of *eval_Its* can be defined to express *ConsIts* nodes where the first child *It* is a *Decl*, *Use*, or *Block*. Once again, this solution would tuple the computations of the environment and errors in a single definition, together with the complex scheduling of the visits. It should also be noticed, that we are expressing the name analysis task of *let* expressions, in the simpler *Block* language. If name analysis was implemented directly in the *let* (or in a real programming language), then the chances to optimize our simple original algorithm could be obfuscated by the more elaborated *let* data type, and other necessary definitions to describe additional *let* tasks.

Let us look at this efficient implementation from an AG perspective. In function *eval_Its* we can see that there is a first visit to subtree *it*, performed by $eval\_It_1$, that computes the $errors_1$ (corresponding to duplicated definitions). There is also a second visit to *it*, performed by $eval\_It_2$, that computes $errors_2$ (corresponding to invalid uses). In fact, in terms of attribute grammars *eval_Its* computes the very same attribute *errors* of nonterminal *It* twice (see framed *errors*). A proper implementation of an attribute grammar, however, *should compute the value of an attribute only once!* Thus, this *block* program is not a proper AG *Block* evaluator, and in fact it is not an evaluator produced by any AG system.

The reader may have already noticed that our zipper-based implementations also re-compute attribute values, and show a similar problem. We will discuss this subject in detail in Section 9.

7 Modularity in Attribute Grammars

Modularity is of utmost importance in software development. In fact, modern languages provide powerful abstractions and modularity mechanisms so that programmers maintain and evolve their software in a modular and incremental way. Abstractions such as *type classes* in *Haskell* or *classes* and *interfaces* in Java, hide from programmers implementation details of reusable components. In fact, software programs are nowadays designed and implemented as combinations of several *generic components*, all physically and conceptually separated from each other. The benefits of such an organization are ease of specification, clearness of description, interchangeability between different "plug-compatible" components, reuse of components across applications and separate analysis/-compilation of components.

Modularity and extensibility is also an important aspect when we wish to implement a complex multiple traversal algorithm in any programming paradigm. As our functional solutions for the **let**/*block* program show, both the strict, multiple traversal program and the circular, lazy one are not modular: while the former includes intrusive gluing data structures and problem specific scheduling of traversals, thus violating the physical and conceptual separation of concerns between traversals functions, the later tuples all computations together!

On the contrary, AG programmers do not define gluing structures, do not schedule traversals, nor tuple all computations in a single component. The AG formalism also offers an efficient form of modularity when each semantic domain is encapsulated in a single AG component [59, 60]. Attribute grammars powerful static analysis techniques, however, require that all individual AG components (such as (AG_i), (AG_{ii_a}), (AG_{ii_b}), and (AG_{iii}) of the *repmin* AG) are first *combined* into an equivalent monolithic AG, so that: all attribute dependencies are analyzed, an attribute evaluation order is computed, and finally an evaluator is produced. In fact, this is a limited form of modularity since a single change in one component may induce an attribute circularity in a different one, and render the entire evaluator invalid! Because our zipper-based embedding of AGs does not rely on a static definition of the attribute evaluation order, it does offer a more powerful form of modularity than traditional AGs systems.

Let us consider again the *Block* language. To make it even easier to locate the errors in an input program, we would like that the desired list of errors not only would follow the sequential structure of the input, but also its nesting structure, while showing whether the error was an invalid declaration or use. Thus, for the input example, the processor should produce the following output $[[\,Use\ w], Decl\ x]$. In fact, we would like to produce as result a *block* program that contains the errors identified in the given input[15].

This extension to the *Block* language can easily be expressed in our setting: we just define a new zipper-based function, named *errorsAsBlock*, where we express this new AG aspect. We omit here the visual definition of the AG since it is very similar to the definition of the attribute *errors* presented before. Instead of using the built-in list constructor functions, we use the correspondent *Its* constructors. The list concatenation (++) is implemented by the recursive (semantic) function *concatErrors*.

```
errorsAsBlock :: AGTree Its
errorsAsBlock t = case constructor t of
                       NilIts_Its  → NilIts
                       ConsIts_Its → concatErrors (errorsAsBlock (t.$1))
                                                  (errorsAsBlock (t.$2))
```

[15] It should be noticed that if the input is an ill-formed program only consisting of *Use* of identifiers, then the output is exactly the same as the input.

```
Use_It     → let error = (lexeme t) `mustBeIn` (env t)
               in if (null error) then NilIts
                  else ConsIts (Use (head error)) NilIts
Decl_It    → let error = (lexeme t, lev t) `mustNotBeIn` (dcli t)
               in if (null error) then NilIts
                  else ConsIts (Decl (head error)) NilIts
Block_It   → ConsIts (Block (errorsAsBlock (t.$1))) NilIts
```

At the root of *Block* we need to apply the *Root* constructor so that we produce as result a value of type *P*.

```
errorsAsBlockP :: AGTree P
errorsAsBlockP t = case constructor t of
                     Root_P → Root (errorsAsBlock (t.$1))
```

For completion we show the simple list concatenation semantic function *concatErrors*:

```
concatErrors :: Its → Its → Its
concatErrors NilIts             its_2 = its_2
concatErrors (ConsIts it its) its_2 = ConsIts it (concatErrors its its_2)
```

The extended version of the *block* program is defined as follows:

```
block :: P → P
block p = errorsAsBlockP (mkAG p)
```

It should be noticed, that we did not change any component of our existing functional solution. In AG words, we did not have to change any AG module, nor did we have to deal with gluing data structures, nor to schedule traversal functions, nor to extend a single definition where all computations are tupled together. We modularly extended our definition by introducing a new function (*i.e.*, AG module), and, as result, we do have an AG implementation that produces the desired results. Note also that our approach also supports incremental compilation: *errorsAsBlock* can be defined in a separate *Haskell* module that is compiled independently.

Nevertheless, it should also be clear that there is a price to pay by using an embedding of AGs: all nice static guarantees provided by the AG formalism are lost, namely the Kastens ordered AG algorithm that statically guarantees termination for the class of ordered attribute grammars. However, the lack of syntactic and semantic domain specific guarantees is a well-know limitation of the shallow embedding of a domain specific language in a host, general purpose one [61–63].

We leave as an exercise (Exercise 3) the implementation of this simple extension of the *Block* program in all strict and lazy functional implementations we have presented in Section 6.

7.1 Higher-Order Attribute Grammars

As we have just shown, AGs offer a modular and extensible software development setting: a new extension can be added in a modular way, without having to change the existing solution. Moreover, in our embedding the added module can be compiled independently. AGs, however, have a severe limitation: when an algorithm is not easily expressed over the underlying AG data structure, a better suited structure can not be used/computed.

This is the case, for example, if we consider the name analysis task of **let** expressions, as presented in Section 5. In fact, because of the simplicity of the *Block* data type/structure we expressed the non-trivial scope rules of **let** expressions in the data type *P*. Indeed, it will be more complex to express such rules in a data type defining **let** expressions.

Swierstra noticed this limitation and introduced *Higher-Order Attribute Grammar* (HOAG) [64, 65], where conventional AGs are augmented with *higher-order attributes*, the so-called *attributable attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have four main characteristics:

- First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. This allows, for example, to transform a **let** expression to a *Block* program, defining the declaration and use of names. The attribute equations define a (synthesized) higher-order attribute representing the *Block* tree. As a result, the decoration of a **let** tree constructs a higher-order tree: the *Block* tree. The attribute equations of the *Block* AG define the scope rules of the **let** language.
- Second, semantic functions are redundant. In higher-order attribute grammars every computation can be modeled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs. Moreover, because we express inductive functions by attributes and attribute equations, the termination of such functions is statically checked by standard AG techniques (*e.g.*, the circularity test).
- The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.
- Finally, as we advocated in [52, 60], attribute grammar components can be "glued" via higher-order attributes.

Let us consider again the *Haskell* **let** expressions (sub)language. Next, we define a heterogeneous data type *Let*, taken from [21], that models such expressions in *Haskell* itself.

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

In order to express the name analysis task of **let** expressions we use the first key feature of HOAG: we synthesize a more suitable tree where the scope rules are easier to express. Because such rules are already specified as the *Block* AG, we juts have to associate attributes and equations to *Let* symbols and productions in order to synthesize a higher-order tree of type *P*. Such attributable attribute is then decorated by the *Block* AG.

We start by defining a (first-order) AG fragment where we synthesize a list of declarations and uses of names in a *List* of **let** expressions. Thus we write attribute equations such that a *Var* constructor induces a *Use* of the respective name, while an *Assign* induces a *Decl* of that name. Next, we show the zipper-based definition of the required equations.

```
letAsBlock :: AGTree Its
letAsBlock t = case constructor t of
  Assign_List    → ConsIts (Decl (lexeme_Name t))
                           (concatIts (letAsBlock (t.$2)) (letAsBlock (t.$3)))
  NestedLet_List → ConsIts (Decl (lexeme_Name t))
                           (ConsIts (Block (concatIts (letAsBlock (t.$2))
                                                      (letAsBlock (t.$3))))
                            NilIts)
  EmptyList_List → NilIts
  Const_Exp      → NilIts
  Var_Exp        → ConsIts (Use (lexeme_Name t)) NilIts
  Neg_Exp        → letAsBlock (t.$1)
  _              → concatIts (letAsBlock (t.$1)) (letAsBlock (t.$2))
```

where *concatIts* ≡ *concatErrors*. Now, we define an attributable attribute in the *Let* production to synthesize the desired *Block* higher-order tree. We name this attribute *ata*. To decorate it according to the *Block* scope rules, we need to convert that higher-order tree into a zipper, first (recall function mkAG). Finally, we may use any zipper-based AG function to compute the attributes of that tree. In this case, we reuse attribute/function *errors* as defined in Section 5.1.

```
letErrors :: AGTree Errors
letErrors t = case constructor t of
  Let_Let → let ata :: Zipper P
                ata = mkAG (Root (letAsBlock (t.$1)))
            in errors ata
```

It should be noticed that the *Block* AG is an off-the-shelf AG component that is directly integrated into the **let** AG specification. Indeed, higher-order AGs provide a powerful and incremental style for language design and implementation [60].

8 Zipping Strategies and Attribute Grammars

As we have shown, AGs are a convenient formalism to express tree-based computations, where context information needs to be collected first, before it can be used. AGs can also be used to performed tree transformations, as showed in the AGs specifying and implementing the *repmin* problem: the original tree is transformed into a similar one (with the minimum value in the leaves). If we look in more detail to the AG fragment where the new tree is constructed (see for example (AG_{iii}) and/or (EAG_{iii})), we see that interesting work is done when visiting *Leaf* productions/constructors, only. The equation in the *Fork* production just uses the same constructor to build the new tree from the transformed subtrees. A similar pattern occurs in the functional solution. For example, in function *replace* (page 7) the work is done in the definition of the *Leaf* alternative, while the *Fork* alternative just forces recursion and the construction of the new tree. This problem is even more visible in the transformation of *Let* tree into a *Block* tree. As we can see in the HOAG definition *letAsBlock*, only a few productions of *Let* contribute to *Block*, more precisely productions *Assign*, *NestedLet* and *Var*!

In fact, when we consider a large-scale tree/program transformation problem, both the AG and the purely functional settings do not offer a proper abstraction to express such transformations. As an example, consider that we wish to refactor *Haskell* programs to transform the use of equality to check if a list l is empty, such as $l \equiv [\,]$, into the use of the proper *null* function: *null* l. *Haskell* consists of 116 constructors across 30 data types,[16] which will result in AG or functional solutions where only in a few productions/constructors interesting work is defined. However, most of the other constructors have to be part of the solution just to force the traversal of *Haskell* syntax tree and the construction of the refactored tree. The problem with this approach is that the size of traversal code is proportional to the number of constructors regardless of the specific problem. At the end of this section we will return to this problem and we will show a concise solution for this problem which consists of 6 lines of code.

In this context, strategic term rewriting is an extremely suitable formalism [18, 66], 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. The key idea underlying strategic programming is the separation of problem-specific ingredients of traversal functionality (*i.e.*, type specific basic actions) and reusable traversal schemes (*i.e.*, traversal control).

To present this powerful abstraction to navigate and transform data structures, we will consider a simpler example: Consider that we wish to implement

[16] `https://hackage.haskell.org/package/haskell-src-1.0.4/docs/Language-Haskell-Syntax.html`

a simple arithmetic optimizer for our **let** language. Let us start with a trivial optimization: the elimination of additions with 0. In our example, the optimization is defined in *Add* nodes, only, and thus we express such a worker function as follows:

```
expr :: Exp → Maybe Exp
expr (Add e (Const 0)) = Just e
expr (Add (Const 0) e) = Just e
expr e                 = Just e
```

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. The last alternative defines the **default** behavior for all other cases, returning the original expression. Because we also need to express transformations that may fail (that is, do nothing), a type-specific transformation function returns a *Maybe* result.

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 joining them into a single function. In this solution, we clearly see that the traversal function full_tdTP needs to navigate heterogeneous trees, as it is the case of *Let* expressions. Moreover, the traversal functions also need to apply transformation at specific nodes, as it is the case of function *expr*. Thus, we will rely again on zippers to provide such a generic transformation and tree-walk mechanism to embed strategic programming in *Haskell*. In fact, our strategic combinators work with zippers as in the definition of *opt*.

In the next section we describe how data structure transformations are directly supported by zippers. Then, in Section 8.2 we present in detail the embedding of strategies using the generic zipper mechanism. By having defined both the embedding of AGs and strategic term-rewriting in the same zipper-based setting, we are able to combine these two powerful techniques as we will show in Section 8.3.

8.1 Transformations with Zippers

The two key ingredients for expressing strategic term re-writing are: a generic data structure traversal mechanism and a generic data structure transformation mechanism. Zippers do not only support navigation on heterogeneous trees (as showed before), but they also support generic transformations.

In fact, the zipper library 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.

In order to show a zipper-based transformation, let us consider that we wish to increment a constant in a **let** expression. We begin by defining a function *incConstant* that increments constants, and use the generic function *mkT* (from the generics library [41]) to generalize this type-specific function to all types:

```
incConstant :: Exp → Exp
incConstant (Const n) = Const (n + 1)
incConstantG :: GenericT
incConstantG = mkT incConstant
```

This function has type *GenericT* (meaning *Generic Transformation*) that is the required type of *trans*. To transform the assignment $c = 2$ (in p) to $c = 3$ we just have to navigate to the desired constant and then apply *trans*, as follows[17]:

```
incrC :: Maybe (Zipper Let)
incrC = do t2 ← down'  t1
           t_3 ← down' t2
           t_4 ← right  t_3
           t_5 ← right  t_4
           t_6 ← down' t_5
           t_7 ← right  t_6
           return (trans incConstantG t_7)
```

8.2 Strategic Programming with Zippers

In this section we introduce Ztrategic, our embedding of strategic programming using generic zippers. The embedding directly follows the work of Laemmel and Visser on the Strafunski library [67, 68]. Before we present the powerful and reusable strategic functions providing control on tree traversals, such as top-down, bottom-up, innermost, etc., let us show some simple basic combinators that work at the zipper level, and are the building blocks of our embedding.

[17] Given that *Maybe* is a monad, this time we avoided the repeated use of *fromJust* by writing the code in a monadic style.

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 transformation that operates on a zipper.

```
zTryApplyM :: (Typeable a, Typeable b) ⇒ (a → Maybe b) → TP c
zTryApplyM f = transM (join . cast . f . fromJust . cast)
```

The definition of zTryApplyM relies on transformations on zippers, thus reusing the generic zipper library *transM* function. To build a valid transformation for the *transM* function, we use the *cast* :: $a \rightarrow$ *Maybe* b function, that tries to cast a given data from type a to type b. In this case, we use it to cast the data the zipper is focused on into a type our original transformation f can be applied to. Then, function f is applied, and its result is cast back to its original type. Should any of these casts, or the function f itself, fail, the failure propagates and the resulting zipper transformation will also fail. The use of the monadic version of the zipper generic transformation guarantees the handling of such partiality. It should be noticed that failure can occur in two situations: the type cast fails when the types do not match. Moreover, the function f fails when the function itself dictates that no change is to be applied. Signaling failure in the application of transformations is important for strategies where a transformation is applied once, only.

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 → 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 → m d)
```

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 transformation, and uses the auxiliary function *maybeKeep* to try to apply the two transformations to the zipper z, ignoring if it fails.

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) into a zipper one, hidden from these definitions.

The function failTP is a pre-defined transformation that always fails and idTP is the identity transformation that always succeeds. 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 but navigating 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*, which 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 have 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. The full API of our zipper-based strategic library is presented in Fig. 11.

Let us return to our **let** running example. Obviously there are more arithmetic rules that we may use to optimize let expressions. In Figure 12, we present the rules given in [69].

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                 -- (1)
expr (Add (Const 0) t)          = Just t                 -- (2)
expr (Add (Const a) (Const b))  = Just (Const (a + b))   -- (3)
expr (Sub a b)                  = Just (Add a (Neg b))   -- (4)
expr (Neg (Neg f))              = Just f                 -- (5)
expr (Neg (Const n))            = Just (Const (−n))      -- (6)
expr _                          = Nothing                -- default
```

Rule 7, however, is context dependent and it is not easily expressed within strategic term rewriting. In fact, this rule requires that the environment, where a name is used, is computed first (according to the scope rules of the **let** language). We will return to this rule in Section 8.3.

Having expressed all rewriting rules 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:

```
Strategy Types
  type TP a = Zipper a → Maybe (Zipper a)
  type TU m d = (forall a . Zipper a → m d)
Primitive Strategies
  idTP     :: TP a
  constTU  :: d → TU m d
  failTP   :: TP a
  failTU   :: TU m d
  tryTP    :: TP a → TP a
  repeatTP :: TP a → TP a
Strategic Construction
  monoTP   :: (a → Maybe b) → TP e
  monoTU   :: (a → m d) → TU m d
  monoTPZ  :: (a → Zipper e → Maybe b) → TP e
  monoTUZ  :: (a → Zipper e → m d) → TU m d
  adhocTP  :: TP e → (a → Maybe b) → TP e
  adhocTU  :: TU m d → (a → m d) → TU m d
  adhocTPZ :: TP e → (a → Zipper e → Maybe b)
                → TP e
  adhocTUZ :: TU m d → (a → Zipper c → m d)
                → TU m d
Composition / Choice
  seqTP    :: TP a → TP a → TP a
  choiceTP :: TP a → TP a → TP a
  seqTU    :: TU m d → TU m d → TU m d
  choiceTU :: TU m d → TU m d → TU m d
```

```
Traversal Combinators
  allTPright :: TP a → TP a
  oneTPright :: TP a → TP a
  allTUright :: TU m d → TU m d
  allTPdown  :: TP a → TP a
  oneTPdown  :: TP a → TP a
  allTUdown  :: TU m d → TU m d
Traversal Strategies
  full_tdTP  :: TP a → TP a
  full_buTP  :: TP a → TP a
  once_tdTP  :: TP a → TP a
  once_buTP  :: TP a → TP a
  stop_tdTP  :: TP a → TP a
  stop_buTP  :: TP a → TP a
  innermost  :: TP a → TP a
  outermost  :: TP a → TP a
  full_tdTU  :: TU m d → TU m d
  full_buTU  :: TU m d → TU m d
  once_tdTU  :: TU m d → TU m d
  once_buTU  :: TU m d → TU m d
  stop_tdTU  :: TU m d → TU m d
  stop_buTU  :: TU m d → TU m d
```

Fig. 11: Full Ztrategic API

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

Fig. 12: Optimization Rules

```
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 $c \to$ Zipper $c \to$ *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 → [Name]
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 → [Name]
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 ≡ ["a","c","b","c"] (if we use a bottom-up strategy, it produces the reverse of this list).

Refactoring Haskell Programs Source code smells make code harder to comprehend and they do occur in any programming language. *Haskell* is no exception. For example, inexperienced *Haskell* programmers often write $l \equiv []$ that uses equality to check whether a list is empty, instead of using the predefined *null* function. Within a strategic programming style we express the elimination of this known smell by defining the type-specific transformation which is defined in two simple alternatives: A first one, specifies the pattern of $l \equiv []$ and the transformation to be performed. A second one defines the work to be performed in all other parts of *Haskell* programs/trees: nothing in this case. In the first alternative, the original and refactored patterns are expressed using the data type defined the full abstract syntax of *Haskell*, as defined in its libraries. By just knowing the types and constructors that represent both $l \equiv []$ and *null* l, we write the following type-specific function:

```
nullList :: HsExp → Maybe HsExp
nullList (HsInfixApp a (HsQVarOp (UnQual (HsSymbol "=="))) (HsList []))
         = Just (HsApp (HsVar (UnQual (HsIdent "null"))) a)
nullList _ = Nothing
```

Now, we define a strategic-based function that applies the *nullList* transformation to a full *Haskell* abstract syntax tree. A refactoring is a type preserving transformation, and in this case we rely in a full bottom-up tree traversal.

```
smells :: Zipper HsModule → Maybe (Zipper HsModule)
smells h = applyTP (full_buTP worker) h
      where worker = failTP `adhocTP` nullList
```

This is the full *Haskell* program that eliminates this known smell. Together with the front-end included in the *Haskell* libraries, we produced an useful refactoring tool: for example, it processed 1139 *Haskell* files (totaling 82124 lines of code), written by first year students, and eliminated 850 code smells in those files [70].

8.3 Strategic Attribute Grammars

As we have shown, our strategic term rewriting functions rely on zippers built upon the data structure (trees) to be traversed. This results in strategic functions that can easily be combined with a zipper-based embedding of attribute grammars [21, 22], since both functions/embedding work on zippers. In the next section, we present in detail the zipping of strategies and AGs.

By having embedding both strategic term re-writing and attribute grammars in the same zipper-based setting, and given that both are embedded as first-class citizen, 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 re-writing rules that rely on context information can access attribute values.

Next, we extend our **let** AG, relying on both techniques to efficiently specify such new language features.

Accessing Attribute Values from Strategies: As we discussed in Section 8.2, rule 7 of Figure 12 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. In order to do this, we need to compute an environment with defined names, which is what we have done with the attribute *env*, previously. Thus, if we have access to such attribute in the definition of a strategy, then we would be able to implement this rule.

Obviously, we have to adapt the type of *Env* (introduced in page 25) according to the definition of rule 7 in Figure 1.

```
type Env = [(Name, Zipper Root, Int)]
```

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, in order to be used to apply the given attribute. This is done in our library by the adhocTPZ combinator:

```
adhocTPZ :: TP e → (a → Zipper e → Maybe b) → TP e
```

Notice that instead of taking a function of type $(a \to Maybe\ b)$, as does the combinator adhocTP introduced in Section 8.2, it receives a function of type $(a \to \mathsf{Zipper}\ e \to Maybe\ b)$, with the zipper as a parameter. Then, we can define a function with this type, that implements rule 7:

```
expC :: Exp → Zipper Let → Maybe Exp
expC (Var x) z = case lookup x (env z) of
                   Just e    → lexeme_Assign e
                   Nothing → Nothing
expC _ z         = Nothing
```

The variable x is searched into the environment returned by the *env* attribute; in case it is found, the associated expression is returned, otherwise the optimization is not performed. The function *lexeme_Assign* is another syntactic reference that in this case takes a Zipper and, if it is focused on an *Assign*, returns its expression.

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

```
opt″  :: Zipper Let → Maybe (Zipper Let)
opt″ r = applyTP (innermost step) r
    where step = failTP ‘adhocTPZ‘ expC ‘adhocTP‘ expr
```

The reader may have already noticed that our zipper function *env* navigates on *block* trees, rather than on **let** ones. To simplify the presentation of zipper-based AGs, we assume that the name analysis of **let** expressions is expressed as a regular first order AG, directly on the **let** abstract syntax, exactly as defined in [21]. In fact, we are not considering the *Let* HOAG presented in Section 7.1. This would require the use of another extension to the AG formalism, namely, reference AGs [71, 72] so that we could access attribute *env* of a *Decl* production from a *Var* production. Recall that *Decl* is a production of the *block* AG, while *Var* is a production of the **let** AG. The automatic transformation of a higher-order AG, as the one defined in 7.1, into an equivalent first order AG, as the **let** AG in [21], is open research. In the context of functional programming, this corresponds to apply program fusion so that the intermediate higher-order trees are deforested [27, 73]. We will return to this topic on Section 11.1.

Synthesizing Attributes via Strategies: We have shown how attributes and strategies can be 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 consider the *errors* attribute, that returns the list of names that violate the scope rules

```
letWithErrors = let a  = b + 3
                    c  = 2
                    w  = let c = a - b
                         in  c + z
                    c  = c + 3 - c
                in  (a + 7) + c + w
```

to the list ["b", "b", "z", "c"]. Recall that errors occur in two situations: First, duplicated definitions that are efficiently detected when a new *Name* (defined in nodes *Assign* and *NestedLet*) is accumulated in *dcli*. Then the newly defined *Name* *must not be in* the environment *dcli* accumulated till that definition/node. This is expressed by the following zipper function:

```
decls :: List → Zipper Let → Errors
decls (Assign    s _ _) z = mustNotBeIn (lexeme z, z) (dcli z)
decls (NestedLet s _ _) z = mustNotBeIn (lexeme z, z) (dcli z)
decls _ _                 = [ ]
```

Invalid uses are detected when a *Name* is used in an arithmetic expression *Exp*. In this case, the *Name* *must be in* the total environment *env*, as it is defined in the worker function *ueses*.

```
uses :: Exp → Zipper Let → Errors
uses (Var i) z = mustBeIn (lexeme z) (env z)
uses _ z       = [ ]
```

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

```
errors :: Zipper Let → Errors
errors t = applyTU (full_tdTU step) t
  where step = failTU `adhocTUZ` uses `adhocTUZ` decls
```

Although, the step function combines *decls* and *uses* in this order, the resulting list does not report duplications first, and invalid uses after. The strategic function adhocTUZ does combine the two functions and the default failing function into one, which is applied while traversing (in a top-down traversal) the tree. In fact, it produces the errors in the proper order.

In Section 5.1 we expressed the zipper-AG version of *errors*, where most of the attribute equations are just propagating attribute values upwards without doing useful work! In fact, a type unifying strategy provides a better solution for specifying such synthesized computations since it focus on the productions/nodes where interesting work has to be defined. 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. In such a large-scale transformation, the difference in complexity between the strategic definition of *errors* and its AG counterpart is much higher.

Attribute grammar systems provide the so-called attribute propagation patterns to avoid polluting the specification with *copy-rules*. In most systems, a special notation and pre-fixed behavior is associated with a set of off-the-shelf patterns that can be reused across AGs [19, 20]. For example, in the UUAG system [20], the propagation patterns are the default rules for any attribute. Thus, only the specific/interesting equations have to be specified. However, being a special notation, hard-wired to the AG system, makes the extension or change of the existing rules almost impossible: the full system has to be updated. Our embedding of strategic term re-writing provides a powerful setting to express attribute propagation patterns: no special additional notation/mechanism is needed.

9 A Proper Zipper-based Embedding of Attribute Grammars

Despite its clear syntax and expressive power, the described zipper-based attribute grammars embedding does not ensure that attributes are computed only once, on a given node. In fact, the same attributes can be evaluated many times on the same node, which causes unnecessary overhead on computations. Under a strictly formal point of view, recomputation of attribute values is against the principles of the AG formalism.

Let us return to the zipper-based *repmin* definition shown in Section 4.3. If we look in detail to the function *nt*, labeled (*iii*), we can see that the function *gm* is called at each leaf of the original tree. To compute the global minimum, *gm* has to navigate through all the input tree. Because *gm* is called every time a new *Leaf* (of the resulting tree) is constructed, a repeated computation of the minimum is thus performed. As a consequence, the global minimum of a tree is computed as many times as the number of leaves the tree has. In our running example this results in three calls to *gm*, and thus to three repeated computations of the global minimum of the input tree! Figure 13 shows the function call chains that activate the computation of function/attribute *nt* on leaves labeled with 3 (left) and 2 (right). In fact, if we overlap the three call trees, the number of overlaps of an attribute in a node, is the number of re-calculations of that attribute value.

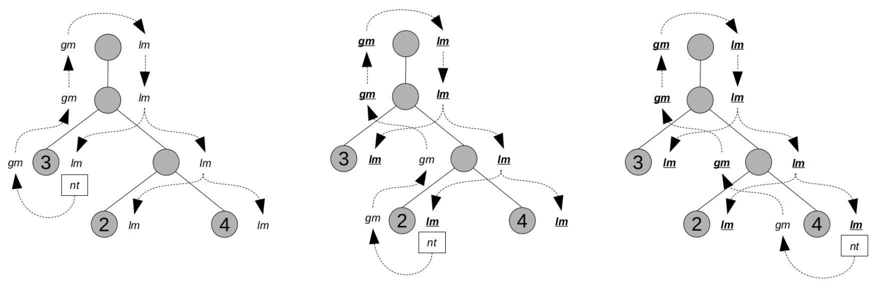


Fig. 13: Function calls to evaluate function/attribute *nt* in the three leaves of the tree: The underlined attributes correspond to attribute instances/values that are recalculated.

In an attribute grammar setting, an attribute evaluator decorates the nodes of the underlying syntax tree with attribute values, and each attribute value is computed only once. Thus, a proper implementation or embedding of an attribute grammar should not repeat the computation of the same attribute instances! That is to say, the zippers presented before do not provide a proper embedding of attribute grammars.

Both in functional programming and attribute grammar setting, repeated computations can be avoided via the use of memoization [74–78]. Thus, a proper embedding of AGs can be achieved by incorporating memoization in our zipper-based embedding, as we proposed in [22].

In order to avoid attribute recomputations, we attach a memo table to each node of a tree to store the value of the attributes associated to the node. We do so by transforming the original tree into a new one of same shape, and with a memo table attached to each node. In this document we omit the details of this implementation[18], and we just present the memoized, zipper-based version of *repmin*. It can be seen that the main differences to the definitions (EAG_i), (EAG_{ii}) and (EAG_{iii}) (presented in Section 4.3) are the use of a memo function, which introduces memoization in the evaluation of the attribute grammar, and the use of **let** to pass around the changing tree (due to values being memoized in the memo tables attached to its nodes)[19].

$$(MAG_i)\qquad \begin{array}{l} lm = \textbf{memo}\ Locmin\ \$\ \lambda t \rightarrow \textbf{case}\ constructor\_m\ t\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Leaf_{Tree} & \rightarrow (\mathsf{lexeme}\ t, t) \\ Fork_{Tree} & \rightarrow \textbf{let}\ \begin{array}{l} (l, t') = lm\ .@.\ left\_m\ \ t \\ (r, t'') = lm\ .@.\ right\_m\ t' \end{array} \\ & \quad\ \textbf{in}\ \ (min\ l\ r, t'') \end{array} \end{array}$$

$$(MAG_{ii})\qquad \begin{array}{l} gm = \textbf{memo}\ Globmin\ \$\ \lambda t \rightarrow \textbf{case}\ constructor\_m\ t\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Root_{Prog} & \rightarrow lm\ .@.\ tree\_m\ t \\ Leaf_{Tree} & \rightarrow gm\ `atParent`\ t \\ Fork_{Tree} & \rightarrow gm\ `atParent`\ t \end{array} \end{array}$$

$$(MAG_{iii})\qquad \begin{array}{l} nt = \textbf{memo}\ Replace\ \$\ \lambda t \rightarrow \textbf{case}\ constructor\_m\ t\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Root_{Prog} & \rightarrow nt\ .@.\ tree\_m\ t \\ Leaf_{Tree} & \rightarrow \textbf{let}\ (mini, t') = gm\ t \\ & \quad\ \textbf{in}\ \ (Leaf\ mini, t') \\ Fork_{Tree} & \rightarrow \textbf{let}\ \begin{array}{l} (l, t') = nt\ .@.\ left\_m\ \ t \\ (r, t'') = nt\ .@.\ right\_m\ t' \end{array} \\ & \quad\ \textbf{in}\ \ (Fork\ l\ r, t'') \end{array} \end{array}$$

To assess in terms of efficiency the memoization approach, we present the results of benchmarking memoized and non-memoized zipper-based versions of

[18] Interested readers can find a detailed account of this memoized, zipper-based AG embedding and the analysis of its performance in [22].

[19] The use of **let** can be avoided by using a *State* monad to thread the modified tree state, as we adopted in [22].

repmin. For this benchmark, we used increasingly larger balanced binary leaf trees with a number of nodes ranging from 2300 to 5000, represented on Fig. 14 in the x-axis.

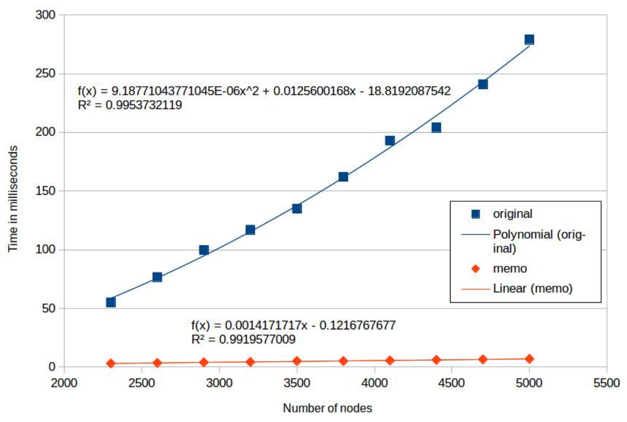


Fig. 14: Non-memoized versus Memoized *repmin* zipper-based programs

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

In [22] we present extensive benchmarks (both in terms of runtime and memory consumption) of several (higher-order) attribute grammars. Moreover, we consider full memoization and selective memoization of the zipper-based evaluators. Under selective memoization not all zipper functions are memoized. Our results show that the memoized zipper-based embedding of AGs maintains the elegance of the original embedding, while exhibiting much better performance, often by various different orders of magnitude.

10 Related Work

Since Knuth's definition of attribute grammars in the mid-60s, there has been extensive work supporting Swierstra's vision of AGs being a suitable formalism to express complex, lazy functional programs. Next, we describe the work being done on circular, lazy programming; on functional zippers; on the embedding of AGs in functional settings; and on the use of function memoization.

Circular, Lazy Programs: Circular programs were first presented by Bird as a technique to eliminate multiple traversals of data [30]. Attribute grammars and circular, lazy functional programs are closely related, as shown by Swierstra [14] and Johnsson [79]. As a consequence, lazy languages are a natural setting to implement AGs [79], and to derive efficient functional programs [14]. Circular, lazy programs can be calculated from strict, multiple traversal counterparts as we have shown in [31–33, 29]. Actually, in [29], Section 5, we calculate the *block* circular program (shown in 6.2), from the strict, two traversal one (shown in 6.1). We have also defined rules to strictify a circular, lazy program, that is to say, to transform circular programs (such as, program in 6.2) into a strict, multiple traversal program (such as program in 6.1) [35]. In the proposed zippers-based embedding of AG, we rely on function memoization to avoid the recalculation of attribute values. Lazy evaluation can also be combined with memoization [75], although the memoization of circular programs is still an open problem.

Functional Zippers: The use of zippers to model AGs has been proposed by several researchers. Uustalu and Vene have shown how to embed attribute computations using comonadic structures, where each tree node is paired with its attribute values [80]. This approach is remarkable for its use of a zipper as in our work. However, it appears that their zipper is not generic and must be instantiated for each tree structure. Laziness is used to avoid static scheduling, and no technique is defined to avoid attribute re-computations.

Badouel *et al.* define attribute evaluation as zipper transformers [81]. While their encoding is simpler than that of Uustalu and Vene, they also use laziness as a key aspect and the zipper representation is similarly not generic. Other work by Badouel *et al.* [82] also requires laziness and forces the programmer to be aware of a cyclic representation of zippers.

Yakushev *et al.* describe a fixed point method for defining operations on mutually recursive data types, with which they build a generic zipper [83]. Their approach is to translate data structures into a generic representation on which traversals and updates can be performed, and then to translate back. Even though their zipper is generic, the implementation is more complex than ours and includes the extra overhead of translation. It also uses advanced features of *Haskell* such as type families and rank-2 types.

Embedded Attribute Grammars: Regarding other notable embeddings of AGs in functional languages, they rely on laziness and use extensible records [84] or heterogeneous collections [85, 86]. The use of heterogeneous lists in the second of these approaches replaces the use in the first approach of extensible records, which are no longer supported by the main *Haskell* compilers. In our framework, attributes do not need to be collected in a data structure at all: they are regular functions upon which correctness checks are statically performed by the compiler.

The result is a simpler and more modular embedding. On the other hand, the use of these data structures ensures that an attribute is computed only once, being then updated to a data structure and later found there when necessary. In order to guarantee such a claim in our setting we need to rely on memorization strategies. Our embedding does not require the programmer to explicitly combine different attributes nor does it require combination of the semantic rules for a particular node in the syntax tree, as is the case in the work of Viera et al. [85, 86]. In this sense, our implementation requires less effort from the programmer.

Another embedding of AGs in Haskell [87] implements a staged approach. The idea is to separate the correctness checks in two phases, one at compile time and the other at runtime. In the first phase some invariants, like the typing of the attribute equations, are checked, while the second phase performs the global checks (e.g. that the used attributes are all defined). Once the checks succeed, the attribute evaluator function is produced, which does not require any more dynamic checks to execute. In contrast to this approach, ours does not involve any dynamic checks at all.

Recently, Norell and Gerdes have proposed an elegant embedding of AGs in the functional language Erlang [88]. While this embedding allows attributes to be used when generating data, and does not rely on lazy evaluation, it still does not include the extensions the zipper-based embedding provides [21].

Zipper-based Embedding of Attribute Grammars: In [42] we presented the zipper-based embedding of first-order AGs. Then, we extended this work to consider most modern attribute grammar extensions [21]. To provide a proper embedding of attribute grammars we incorporated memoization in our zipper-based embedding[89], as we briefly presented in this paper. The paper [22] extends this work by considering the higher-order extension of AG, and by presenting an in depth study of the performance of the memoized zipper-based AGs.

Strategic Term Rewriting: Strategic programming is not just a theoretical concept. It has actually been realized within several programming paradigms, namely in term rewriting based on the Stratego language [90], in functional programming based on *Haskell* [67, 91], in object-oriented programming based on Java [92, 93], and in attribute grammars [51]. Our Ztrategic library is inspired by Strafunski [67]. 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.

Combining Attribute Grammars and Strategic Term-Rewriting: Our joint embedding of AGs and strategies follows the pioneering work of Sloane who developed Kiama [51, 94]: 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 [95]. In our purely functional setting, we only compute attributes in the desired re-written tree (as is the case of the HOAG defining the **let** name analysis presented in section 7.1).

Influenced by Kiama, Kramer and Van Wyk [69] define *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 show in our work. They present several practical applications, 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 [44]. 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 [50].

JastAdd is a reference attribute grammar based system [55]. It supports most of AG extensions, including reference and circular AGs [72]. It also supports tree rewriting, with rewrite rules that can reference attributes. However, JastAdd 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 [21, 22]. Because strategies and AGs are first-class citizens, we can smoothly combine any such extensions with strategic term rewriting.

Memoization in Attribute Grammars: Incremental attribute evaluators memoize all attribute values in order to obtain an optimal re-evaluation time. After a change in the underlying tree, the dependency graph induced by the attribute equations is used to re-evaluate only the attributes affected by such a tree change [96]. After the seminal work of Reps and Teitelbaum [12, 74, 96] on (incremental) syntax-oriented editors, several AG systems were developed to produce evaluators which memoize all attribute values, either in the tree's nodes of an imperative evaluator [12, 55], or by memoizing function calls in a functional evaluator [13, 76, 77]. Our embedding of AG mimics the imperative evaluators in the sense that all attribute values are cached in tree nodes.

The incremental evaluation of HOAGs has being extensively studied by Swierstra's group [13, 52, 76, 77] where AGs are compiled to purely functional attribute evaluators, and function memoization is used to achieve incremental evaluation. More recently, a new technique to compile HOAGs into incremental evaluators was developed: the evaluator maintains additional bookkeeping about which attribute values have changed [78, 97]. However, the overhead that comes with the bookkeeping greatly impacts performance.

To improve the performance of attribute evaluators, Söderberg and Hedin propose automate selective memoization in the context of reference AGs [98]. Our memoized, zipper-based embedding of AGs also supports selective memoization. In fact, in [22] the performance of the zipper-based AGs did improve by selecting which attributes to memoize.

11 Conclusions

In this document we presented the attribute grammar formalism as an elegant, intuitive and modular programming style to define complex multiple traversal algorithms. We revisited our simple embedding of the AG formalism as purely zipper-based *Haskell* functions, and we showed how to express such algorithms directly as an *Haskell* AG.

As we have carefully shown, when programming in this attribute grammar style, *Haskell* programmers do not have to define additional and intrusive data structures to glue traversal functions, nor to define counter-intuitive and hard to debug circular definitions. Thus, when using our embedding, functional programmers do not have to transform/optimize their original and clear multiple traversal algorithms, in order to avoid intrusive code or counter-intuitive solutions. In fact, the clear algorithm is naturally and elegantly expressed as a purely functional, zipper-based program.

Moreover, the presented AG embedding supports modular and extensible program design: different aspects of the algorithm are defined via zipper-based functions, and thus, clearly and conceptually separated from each other. In fact, there is no intrusive code to glue concerns, nor the tupling of all concerns in a single one! Furthermore, new extensions adding new functionalities/concerns can be added in a modular way, without having to change the existing solution. The new module/concern can be independently and incrementally compiled.

To avoid the re-evaluation of attributes we extended our embedding of AGs, with the memoization of the calls to the zipper-based functions. As a result, when using our memoized embedding of AGs, we get a proper and efficient attribute evaluator for free: It behaves as a linear function, while the non-memoized version behaves as a quadratic one.

Moreover, in order to express large-scale tree transformations, we combined AGs with strategic term-rewriting. Very much like AGs, strategies rely on a generic tree traversal mechanism. Thus, we expressed strategies via generic zippers which can be directly combined with our embedding of AGs. Such a joint embedding results in a multi-paradigm embedding of the two porwefull language engineering techniques.

11.1 Future Work

The work presented in this document shows two relevant directions for further research, namely:

Fusion in Higher-Order Attribute Grammars: HOAGs provide a component-based style of programming in language specification [60]. In fact, we used this AG extension when specifying the name analysis task of the **let** language: the *block* AG was reused as a name analysis off-the-shelf AG component (as shown in Section 7.1). When defining the **let** optimization rules, however, we needed to access attributes defined in the higher-order tree (*i.e.*, in the *block* tree/language) from a strategic function. Recall, for example, the use of attribute *env* on strategic function *expC* on page 53.

Unfortunately, accessing attributes in the higher-order tree from the original abstract syntax tree is not possible in HOAGs. Thus, in the strategic functions defined in Section 8.3 we are not considering the **let** HOAG defined in 7.1. In fact, when defining *expC* we used the equivalent first order AG, where the name analysis task is directly define in the **let** productions (exactly as defined in [21]).

In this case, we are loosing all the nice properties of the higher-order extension of AGs. Thus, the research question that we wish to address is: *how can we automatically transform a HOAG into an equivalent first order AG?* By looking at the evaluation of a higher-order AG we see that first a higher-order tree is constructed, which is then consumed when evaluating its attributes. Indeed, the higher-order tree is the intermediate data structure that glues the algorithms expressed in the original tree (the **let** in our example) and in the higher-order one (the *block* tree in our example). In functional programming, program fusion is a powerful technique to eliminate such intermediate data structures [27, 73]. As a consequence, we would like to extend program fusion to the higher-order attribute realm.

Fusion will not only improve the expressiveness of our combined embedding of strategies and AGs, but it will also produce more efficient evaluators since higher-order trees will be automatically deforested.

Memoized Strategies: In Section 9, we memoized attribute values to avoid attribute instances re-evaluation, thus achieving a proper embedding of AGs. By memoizing the calls to the zipper functions, we can significantly improve the performance of our zipper-based embedding: the non-memoized version behaves as a quadratic function, while the memoized behaves as a linear one. The embedding of strategies introduced in Section 8, however, is expressed on the non-memoized version of our embedding. In fact, we intend to extend and adapt our Ztrategic library so that it can be combined with memoized zipper-based AGs. In that way strategies will be able to access memoized attributes, and memoized attributes can be expressed by strategies. Moreover, we also wish to consider the memoization of the strategic combinator functions, so that the same transformation applied in equal subtrees of a AST is performed once, only.

Exercises

Exercise 1. Consider the problem of breath-first numbering a binary tree as proposed in [99]. Implement the strict, multiple traversal program; the circular, lazy program; and the zipper-based AG program.

(a zipper-based AG solution for this problem is presented in [100])

Exercise 2. Define a data type *Let* modeling let expressions in Haskell. Implement a zipper-based function *let2block* :: *Let* → *P*.

(a solution for this problem is presented in [43])

Exercise 3. Consider again the *block* processor that considers both the input and nesting structure of the input to produce the errors. Note that the zipper-based function *errorsAsBlockP* presented in Section 7 is inefficient since it is appending lists every time an error is detected.

1. Define the zipper-based AG module that uses an accumulator attribute so that list concatenations are avoided.
2. Implement the same extension in the three *Haskell* evaluators presented on pages 33, 34, and 37, respectively.

Exercise 4. Develop a program that recognizes and formats (possibly nested) HTML style tables. More precisely, we want our program to receive an abstract data type of an HTML table and to print a geometrically well defined table. An entry in the table can be a string or a nested table. Next we show an example of a possible input (left) and correspondent output (right), where we annotte each entry with its width/height (blue subscripts / superscripts) to make it easier for the reader to understand the necessary formatting process.

$\langle\text{TABLE}\rangle$

$\quad\langle\text{TR}\rangle\langle\text{TD}\rangle^{1}_{14}$ `The first line` $\langle/\text{TD}\rangle\langle\text{TD}\rangle^{1}_{4}$ `of a` $\langle/\text{TD}\rangle\langle/\text{TR}\rangle$

$\quad\langle\text{TR}\rangle\langle\text{TD}\rangle^{7}_{12}\langle\text{TABLE}\rangle$

$\qquad\qquad\langle\text{TR}\rangle\langle\text{TD}\rangle^{1}_{4}$ `This` $\langle/\text{TD}\rangle\langle\text{TD}\rangle^{1}_{2}$ `is` $\langle/\text{TD}\rangle\langle/\text{TR}\rangle$

$\qquad\qquad\langle\text{TR}\rangle\langle\text{TD}\rangle^{1}_{7}$ `another` $\langle/\text{TD}\rangle\langle\text{TD}/\rangle\langle/\text{TR}\rangle$

$\qquad\qquad\langle\text{TR}\rangle\langle\text{TD}\rangle^{1}_{5}$ `table` $\langle/\text{TD}\rangle\langle\text{TD}/\rangle\langle/\text{TR}\rangle$

$\qquad\quad\langle/\text{TABLE}\rangle$

$\qquad\quad\langle/\text{TD}\rangle\langle\text{TD}\rangle^{1}_{5}$ `table` $\langle/\text{TD}\rangle\langle/\text{TR}\rangle$

$\langle/\text{TABLE}\rangle$

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

Notice that in the output, all the lines have the same number of columns and the columns have the same length. None of these features are required in the HTML language. It should also be noticed that it is required to know the sizes of inner tables in order to resize the outer ones.

(an AG solution for this problem is presented in [8])

Exercise 5. Consider the advanced pretty printing algorithm described in Section 4 of [8]. Express the given attribute grammar as a zipper-based *Haskell* AG.

Exercise 6. Write the strategic version of the repmin function *replace* (presented in page 7), which is also expressed in the AG programming style, as function *nt*, in Section 4.3.

References

1. D. E. Knuth, Semantics of Context-free Languages, Mathematical Systems Theory 2 (2) (1968) 127–145, correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
URL `https://doi.org/10.1007/BF01692511`
2. A. Dijkstra, J. Fokker, S. D. Swierstra, The architecture of the Utrecht Haskell compiler, in: S. Weirich (Ed.), Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009, ACM, 2009, pp. 93–104.
URL `https://doi.org/10.1145/1596638.1596650`
3. J. Öqvist, G. Hedin, Extending the JastAdd extensible Java compiler to Java 7, in: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, ACM, New York, NY, USA, 2013, p. 147–152.
URL `https://doi.org/10.1145/2500828.2500843`
4. T. Kaminski, E. Van Wyk, A modular specification of Oberon0 using the Silver attribute grammar system, Science of Computer Programming 114 (2015) 33–44, LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.
URL `https://doi.org/10.1007/BF00264249`
5. A. M. Sloane, M. Roberts, Oberon0 in Kiama, Science of Computer Programming 114 (2015) 20–32.
URL `https://doi.org/10.1016/j.scico.2015.10.010`
6. J. Åkesson, T. Ekman, G. Hedin, Implementation of a Modelica compiler using JastAdd attribute grammars, Science of Computer Programming 75 (1) (2010) 21 – 38.
URL `https://doi.org/10.1016/j.scico.2009.07.003`
7. I. F. Jr., M. Mernik, I. Fister, D. Hrncic, Implementation of a domain-specific language EasyTime using LISA compiler generator, in: M. Ganzha, L. A. Maciaszek, M. Paprzycki (Eds.), Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings, 2011, pp. 801–808.
URL `http://ieeexplore.ieee.org/document/6078287/`
8. S. D. Swierstra, P. R. Azero Alcocer, J. Saraiva, Designing and implementing combinator languages, in: S. D. Swierstra, J. N. Oliveira, P. R. Henriques (Eds.), Advanced Functional Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 150–206.
URL `https://doi.org/10.1007/10704973_4`
9. J. Saraiva, D. Swierstra, Data Structure Free Compilation, in: Stefan Jähnichen (Ed.), 8th International Conference on Compiler Construction, CC/ETAPS'99, Vol. 1575 of LNCS, Springer-Verlag, 1999, pp. 1–16.
URL `https://doi.org/10.1007/978-3-540-49051-7_1`
10. J. P. Fernandes, J. Saraiva, Tools and Libraries to Model and Manipulate Circular Programs, in: PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation, ACM Press, 2007, pp. 102–111.
URL `https://doi.org/10.1145/1244381.1244399`
11. A. Middelkoop, A. Dijkstra, S. D. Swierstra, Iterative type inference with attribute grammars, in: E. Visser, J. Järvi (Eds.), Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven,

The Netherlands, October 10-13, 2010, ACM, 2010, pp. 43–52.
URL `https://doi.org/10.1145/1868294.1868302`

12. T. Reps, T. Teitelbaum, The synthesizer generator, SIGPLAN Not. 19 (5) (1984) 42–48.
URL `http://doi.acm.org/10.1145/390011.808247`
13. M. Kuiper, J. Saraiva, Lrc - A Generator for Incremental Language-Oriented Tools, in: K. Koskimies (Ed.), 7th International Conference on Compiler Construction, CC/ETAPS'98, Vol. 1383 of LNCS, Springer-Verlag, 1998, pp. 298–301.
URL `https://doi.org/10.1007/BFb0026440`
14. M. Kuiper, D. Swierstra, Using attribute grammars to derive efficient functional programs, in: Computing Science in the Netherlands CSN'87, 1987.
15. G. Huet, The Zipper, Journal of Functional Programming 7 (5) (1997) 549–554.
URL `https://doi.org/10.1017/S0956796897002864`
16. R. Bird, Introduction to Functional Programming using Haskell, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1998.
17. S. Jones, Haskell 98 Language and Libraries: The Revised Report, Journal of functional programming: Special issue, Cambridge University Press, 2003.
18. S. P. Luttik, E. Visser, 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, 1997, p. 9.
19. U. Kastens, P. Pfahler, M. T. Jung, The Eli System, in: Int. Conference on Compiler Construction, Springer-Verlag, 1998, pp. 294–297.
URL `https://doi.org/10.1007/BFb0026439`
20. A. Dijkstra, S. D. Swierstra, Typing Haskell with an attribute grammar, in: V. Vene, T. Uustalu (Eds.), Advanced Functional Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 1–72.
URL `https://doi.org/10.1007/11546382_1`
21. P. Martins, J. P. Fernandes, J. Saraiva, E. Van Wyk, A. Sloane, Embedding attribute grammars and their extensions using functional zippers, Science of Computer Programming 132 (P1) (2016) 2–28.
URL `https://doi.org/10.1016/j.scico.2016.03.005`
22. J. P. Fernandes, P. Martins, A. Pardo, J. Saraiva, M. Viera, Memoized zipper-based attribute grammars and their higher order extension, Science of Computer Programming 173 (2019) 71–94.
URL `https://doi.org/10.1016/j.scico.2018.10.006`
23. Z. Hu, H. Iwasaki, M. Takeichi, A. Takano, Tupling calculation eliminates multiple data traversals, in: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97, Association for Computing Machinery, New York, NY, USA, 1997, p. 164–175.
URL `https://doi.org/10.1145/258948.258964`
24. A. Gill, J. Launchbury, S. L. Peyton Jones, A short cut to deforestation, in: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, FPCA '93, Association for Computing Machinery, New York, NY, USA, 1993, p. 223–232.
URL `https://doi.org/10.1145/165180.165214`
25. A. Takano, E. Meijer, Shortcut deforestation in calculational form, in: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95, Association for Computing Machinery, New York, NY, USA, 1995, p. 306–313.
URL `https://doi.org/10.1145/224164.224221`

26. J. Voigtländer, Semantics and pragmatics of new shortcut fusion rules, in: J. Garrigue, M. V. Hermenegildo (Eds.), Functional and Logic Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 163–179.
URL `https://doi.org/10.1007/978-3-540-78969-7_13`
27. P. Wadler, Deforestation: transforming programs to eliminate trees, Theoretical Computer Science 73 (1990) 231–248.
URL `https://doi.org/10.1016/0304-3975(90)90147-A`
28. Z. Hu, T. Yokoyama, M. Takeichi, Program optimizations and transformations in calculation form, in: Lämmel et al. [101], pp. 144–168.
URL `https://doi.org/10.1007/11877028_5`
29. J. P. Fernandes, J. Cunha, J. Saraiva, A. Pardo, Watch out for that tree! A tutorial on shortcut deforestation, in: V. Zsók, Z. Porkoláb, Z. Horváth (Eds.), Central European Functional Programming School: 6th Summer School, CEFP 2015, Budapest, Hungary, July 6–10, 2015, Revised Selected Papers, Springer International Publishing, 2019, pp. 1–41.
URL `https://doi.org/10.1007/978-3-030-28346-9_1`
30. R. S. Bird, Using Circular Programs to Eliminate Multiple Traversals of Data, Acta Informatica (21) (1984) 239–250.
URL `https://doi.org/10.1007/BF00264249`
31. J. P. Fernandes, A. Pardo, J. Saraiva, A shortcut fusion rule for circular program calculation, in: Haskell'07: Proceedings of the ACM SIGPLAN Haskell Workshop, ACM Press, New York, NY, USA, 2007, pp. 95–106.
URL `https://doi.org/10.1145/1291201.1291216`
32. A. Pardo, J. P. Fernandes, J. Saraiva, Shortcut fusion rules for the derivation of circular and higher-order monadic programs, in: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 81–90.
URL `https://doi.org/10.1145/1480945.1480958`
33. A. Pardo, J. P. Fernandes, J. Saraiva, Shortcut fusion rules for the derivation of circular and higher-order programs, High. Order Symb. Comput. 24 (1-2) (2011) 115–149.
URL `https://doi.org/10.1007/s10990-011-9076-x`
34. J. P. Fernandes, Design, implementation and calculation of circular programs, Ph.D. thesis, Deparment of Informatics, University of Minho, Portugal (2009).
35. J. P. Fernandes, J. Saraiva, D. Seidel, J. Voigtländer, Strictification of circular programs, in: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 131–140.
URL `https://doi.org/10.1145/1929501.1929526`
36. J. Paakki, Attribute grammar paradigms - A high-level methodology in language implementation, ACM Computing Surveys 27 (2) (1995) 196–255.
URL `https://doi.org/10.1145/210376.197409`
37. T. Reps, T. Teitelbaum, The Synthesizer Generator Reference Manual, 3rd Edition, Springer, 1989.
URL `https://doi.org/10.1007/978-1-4613-9633-8`
38. D. Swierstra, A. Baars, A. Löh, The UU-AG Attribute Grammar System, `http://www.cs.uu.nl/groups/ST` (2004).
39. H. Alblas, Attribute evaluation methods, in: H. Alblas, B. Melichar (Eds.), International Summer School on Attribute Grammars, Applications and Systems, Vol. 545 of LNCS, 1991, pp. 48–113.
URL `https://doi.org/10.1007/3-540-54572-7_3`

40. M. D. Adams, Scrap your zippers: A generic zipper for heterogeneous types, in: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming, WGP '10, New York, NY, USA, 2010, pp. 13–24.
URL `https://doi.org/10.1145/1863495.1863499`
41. R. Lämmel, S. P. Jones, Scrap your boilerplate: A practical design pattern for generic programming, in: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI '03, New York, NY, USA, 2003, pp. 26–37.
URL `https://doi.org/10.1145/640136.604179`
42. P. Martins, J. P. Fernandes, J. Saraiva, Zipper-based attribute grammars and their extensions, in: A. R. Du Bois, P. Trinder (Eds.), Programming Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 135–149.
URL `https://doi.org/10.1007/978-3-642-40922-6_10`
43. P. Martins, Embedding attribute grammars and their extensions using functional zippers, Ph.D. thesis, Deparment of Informatics, University of Minho, Portugal (July 2014).
44. T. Sheard, S. P. Jones, 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, 2002, p. 1–16.
URL `https://doi.org/10.1145/581690.581691`
45. T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Lämmel et al. [101], pp. 422–436.
URL `https://doi.org/10.1007/11877028_18`
46. U. Kastens, W. M. Waite, Name analysis for modern languages: a general solution, Softw. Pract. Exp. 47 (11) (2017) 1597–1631.
URL `https://doi.org/10.1002/spe.2489`
47. D. E. Knuth, The genesis of attribute grammars, in: P. Deransart, M. Jourdan (Eds.), Attribute Grammars and their Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 1990, pp. 1–12.
URL `https://doi.org/10.1007/3-540-53101-7_1`
48. A. van Wijngaarcien, B. J. Mailloux, J. E. L. Peck, C. H. A. Kostcr, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, Revised Report on the Algorithmic Language Algol 68, SIGPLAN Not. 12 (5) (1977) 1–70.
URL `https://doi.org/10.1145/954652.1781176`
49. J. Saraiva, Purely functional implementation of attribute grammars, Ph.D. thesis, Utrecht University, The Netherlands (December 1999).
URL `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`
50. E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an Extensible Attribute Grammar System, Electronic Notes in Theoretical Computer Science 203 (2) (2008) 103–116.
URL `http://dx.doi.org/10.1016/j.entcs.2008.03.047`
51. A. M. Sloane, L. C. L. Kats, E. Visser, A pure object-oriented embedding of attribute grammars, Electronic Notes in Theoretical Computer Science 253 (7) (2010) 205–219.
URL `http://dx.doi.org/10.1016/j.entcs.2010.08.043`
52. J. Saraiva, S. D. Swierstra, Generating spreadsheet-like tools from strong attribute grammars, in: Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings, 2003, pp. 307–323.
URL `https://doi.org/10.1007/978-3-540-39815-8_19`

53. L. Krishnan, E. V. Wyk, Monolithic and modular termination analyses for higher-order attribute grammars, Science of Computer Programmaing 96 (2014) 511–526. URL https://doi.org/10.1016/j.scico.2014.05.016
54. R. Farrow, Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars, in: ACM SIGPLAN '86 Symp. on Compiler Construction, ACM press, 1986, pp. 85–98.
URL https://doi.org/10.1145/12276.13320
55. T. Ekman, G. Hedin, The JastAdd extensible Java compiler, SIGPLAN Not. 42 (10) (2007) 1–18.
URL http://doi.acm.org/10.1145/1297105.1297029
56. U. Kastens, Ordered attribute grammars, Acta Informatica 13 (1980) 229–256. URL https://doi.org/10.1007/BF00288644
57. U. Kastens, Implementation of Visit-Oriented Attribute Evaluators, in: H. Alblas, B. Melichar (Eds.), International Summer School on Attribute Grammars, Applications and Systems, Vol. 545 of LNCS, 1991, pp. 114–139.
URL https://doi.org/10.1007/3-540-54572-7_4
58. A. Pardo, J. P. Fernandes, J. Saraiva, Multiple intermediate structure deforestation by shortcut fusion, Science of Computer Programming 132 (2016) 77–95. URL https://doi.org/10.1016/j.scico.2016.07.004
59. U. Kastens, W. M. Waite, Modularity and reusability in attribute grammars, Acta Inf. 31 (7) (1994) 601–627.
URL https://doi.org/10.1007/BF01177548
60. J. Saraiva, 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, 2002, pp. 268–282.
URL https://doi.org/10.1007/3-540-45821-2_17
61. P. Hudak, Building domain-specific embedded languages, ACM Comput. Surv. 28 (4es) (Dec. 1996).
URL http://doi.acm.org/10.1145/242224.242477
62. J. Saraiva, D. Swierstra, Generic Attribute Grammars, in: D. Parigot, M. Mernik (Eds.), Second Workshop on Attribute Grammars and their Applications, WAGA'99, INRIA Rocquencourt, Amsterdam, The Netherlands, 1999, pp. 185–204.
63. J. Gibbons, N. Wu, Folding domain-specific languages: Deep and shallow embeddings (functional pearl), SIGPLAN Not. 49 (9) (2014) 339–347.
URL https://doi.org/10.1145/2692915.2628138
64. H. H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher order attribute grammars, in: Proc. of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, PLDI '89, ACM, New York, NY, USA, 1989, p. 131–145.
URL https://doi.org/10.1145/73141.74830
65. H. H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher order attribute grammars, SIGPLAN Not. 24 (7) (1989) 131–145.
URL https://doi.org/10.1145/74818.74830
66. E. Visser, Z.-e.-A. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98, Association for Computing Machinery, New York, NY, USA, 1998, p. 13–26.
URL https://doi.org/10.1145/289423.289425

67. R. Lämmel, J. Visser, Typed combinators for generic traversal, in: S. Krishnamurthi, C. R. Ramakrishnan (Eds.), Practical Aspects of Declarative Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 137–154.
URL `https://doi.org/10.1007/3-540-45587-6_10`
68. R. Lämmel, J. Visser, A strafunski application letter, in: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages, PADL '03, Springer-Verlag, Berlin, Heidelberg, 2003, p. 357–375.
URL `https://doi.org/10.1007/3-540-36388-2_24`
69. L. Kramer, E. Van Wyk, Strategic tree rewriting in attribute grammars, in: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 210–229.
URL `https://doi.org/10.1145/3426425.3426943`
70. J. N. Macedo, M. Viera, J. Saraiva, Zipping strategies and attribute grammars, in: 16th International Symposium on Functional and Logic Programming, FLOPS 2022, (to appear), 2022.
71. G. Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000).
72. E. Söderberg, G. Hedin, Circular higher-order reference attribute grammars, in: M. Erwig, R. F. Paige, E. Van Wyk (Eds.), Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings, Vol. 8225 of Lecture Notes in Computer Science, Springer International Publishing, Cham, 2013, pp. 302–321.
URL `https://doi.org/10.1007/978-3-319-02654-1_17`
73. J. Launchbury, T. Sheard, Warm fusion: Deriving build-catas from recursive definitions, in: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95, ACM, New York, NY, USA, 1995, p. 314–323.
URL `https://doi.org/10.1145/224164.224223`
74. T. Reps, T. Teitelbaum, A. Demers, Incremental context-dependent analysis for language-based editors, ACM Transactions on Programming Languages and Systems 5 (3) (1983) 449–477.
URL `https://doi.org/10.1145/2166.357218`
75. J. Hughes, Lazy memo-functions, in: J.-P. Jouannaud (Ed.), Functional Programming Languages and Computer Architecture, Vol. 201 of LNCS, 1985, pp. 129–146.
URL `https://doi.org/10.1007/3-540-15975-4_34`
76. M. Pennings, D. Swierstra, H. Vogt, Using cached functions and constructors for incremental attribute evaluation, in: M. Bruynooghe, M. Wirsing (Eds.), Programming Language Implementation and Logic Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 130–144.
URL `https://doi.org/10.1007/3-540-55844-6_132`
77. J. Saraiva, S. D. Swierstra, M. F. Kuiper, Functional incremental attribute evaluation, in: Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings, 2000, pp. 279–294.
URL `http://dx.doi.org/10.1007/3-540-46423-9_19`
78. J. Bransen, A. Dijkstra, S. D. Swierstra, Incremental evaluation of higher-order attributes, Science of Computer Programming 137 (2017) 98 – 124, selected and extended papers from Partial Evaluation and Program Manipulation 2015

(PEPM'15).
URL `http://dx.doi.org/10.1016/j.scico.2016.06.001`

79. T. Johnsson, Attribute grammars as a functional programming paradigm, in: Functional Programming Languages and Computer Architecture, 1987, pp. 154–173.
URL `https://doi.org/10.1007/3-540-18317-5_10`
80. T. Uustalu, V. Vene, Comonadic functional attribute evaluation, Trends in Functional Programming, Intellect Books (10), 2005, pp. 145–162.
81. E. Badouel, R. Tchougong, C. Nkuimi-Jugnia, B. Fotsing, Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition, Theoretical Computer Science 480 (0) (2013) 1 – 25.
URL `https://doi.org/10.1016/j.tcs.2013.02.007`
82. E. Badouel, B. Fotsing, R. Tchougong, Attribute grammars as recursion schemes over cyclic representations of zippers, Electronic Notes Theory Computer Science 229 (5) (2011) 39–56.
URL `https://doi.org/10.1016/j.entcs.2011.02.015`
83. A. R. Yakushev, S. Holdermans, A. Löh, J. Jeuring, Generic programming with fixed points for mutually recursive datatypes, in: Procs. of the 14th ACM SIGPLAN International Conference on Functional programming, 2009, pp. 233–244.
URL `https://doi.org/10.1145/1596550.1596585`
84. O. de Moor, K. Backhouse, D. Swierstra, First-class attribute grammars, Informatica (Slovenia) 24 (3) (2000).
URL `citeseer.ist.psu.edu/demoor00firstclass.html`
85. M. Viera, S. D. Swierstra, W. Swierstra, Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell, SIGPLAN Not. 44 (9) (2009) 245–256.
URL `http://doi.acm.org/10.1145/1631687.1596586`
86. M. Viera, First class syntax, semantics, and their composition, Ph.D. thesis, Utrecht University, The Netherlands (2013).
87. M. Viera, F. Balestrieri, A. Pardo, A staged embedding of attribute grammars in haskell, in: Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 95–106.
URL `https://doi.org/10.1145/3310232.3310235`
88. U. Norell, A. Gerdes, Attribute Grammars in Erlang, in: Workshop on Erlang, 2015, ACM, 2015, pp. 1–12.
URL `http://doi.acm.org/10.1145/2804295.2804296`
89. J. P. Fernandes, P. Martins, A. Pardo, J. Saraiva, M. Viera, Memoized zipper-based attribute grammars, in: 20th Brazilian Symposium on Programming Languages (SBLP 2016), Vol. 9889 of LNCS, Springer, 2016, pp. 46–61.
URL `https://doi.org/10.1007/978-3-319-45279-1_4`
90. E. Visser, Stratego: A language for program transformation based on rewriting strategies, in: Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01), LNCS, Springer-Verlag, Berlin, Heidelberg, 2001, p. 357–362.
URL `https://doi.org/10.1007/3-540-45127-7_27`
91. E. Visser, Z.-e.-A. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, SIGPLAN Not. 34 (1) (1998) 13–26.
URL `https://doi.org/10.1145/291251.289425`

92. J. Visser, Visitor combination and traversal control, SIGPLAN Not. 36 (11) (2001) 270–282.
URL `https://doi.org/10.1145/504311.504302`

93. E. Balland, P. Brauner, R. Kopetz, P. Moreau, A. Reilles, Tom: Piggybacking rewriting on java, in: F. Baader (Ed.), Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings, Vol. 4533 of Lecture Notes in Computer Science, Springer, 2007, pp. 36–47.
URL `https://doi.org/10.1007/978-3-540-73449-9\_5`

94. L. C. L. Kats, A. M. Sloane, E. Visser, 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, CC '09, Springer-Verlag, Berlin, Heidelberg, 2009, p. 142–157.
URL `https://doi.org/10.1007/978-3-642-00722-4_11`

95. A. M. Sloane, M. Roberts, L. G. C. Hamey, Respect your parents: How attribution and rewriting can get along, in: B. Combemale, D. J. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings, Vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, Cham, 2014, pp. 191–210.
URL `https://doi.org/10.1007/978-3-319-11245-9_11`

96. T. Reps, Optimal-time incremental semantic analysis for syntax-directed editors, in: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82, ACM, New York, NY, USA, 1982, pp. 169–176.
URL `http://doi.acm.org/10.1145/582153.582172`

97. J. Bransen, A. Dijkstra, S. D. Swierstra, Incremental evaluation of higher order attributes, in: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015, 2015, pp. 39–48.
URL `http://doi.acm.org/10.1145/2678015.2682541`

98. E. Söderberg, G. Hedin, Automated selective caching for reference attribute grammars, in: Proceedings of the Third International Conference on Software Language Engineering, SLE 10, Vol. 6563 of LNCS, Springer-Verlag, 2010, pp. 2–21.
URL `https://doi.org/10.1007/978-3-642-19440-5\_2`

99. C. Okasaki, Breadth-first numbering: Lessons from a small exercise in algorithm design, SIGPLAN Not. 35 (9) (2000) 131–136.
URL `https://doi.org/10.1145/357766.351253`

100. P. Martins, J. P. Fernandes, J. Saraiva, Zipper-based modular and deforested computations, in: V. Zsók, Z. Horváth, L. Csató (Eds.), Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers, Springer International Publishing, Cham, 2015, pp. 407–427.
URL `https://doi.org/10.1007/978-3-319-15940-9_10`

101. R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Vol. 4143 of Lecture Notes in Computer Science, Springer, 2006.
URL `https://doi.org/10.1007/11877028`