Embedding Attribute Grammars and their Extensions using Functional Zippers

Pedro Martins[1⋆], João Paulo Fernandes[1,2], João Saraiva[1], Eric Van Wyk[3], and Anthony Sloane[4]

[1] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Braga, Portugal
[2] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Covilhã, Portugal
[3] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA
[4] Department of Computing, Macquarie University, Sydney, Australia
{prmartins,jpaulo,jas}@di.uminho.pt, evw@cs.umn.edu, anthony.sloane@mq.edu.au

Abstract. Attribute grammars are a suitable formalism to express complex software language analysis and manipulation algorithms, which rely on multiple traversals of the underlying syntax tree. Attribute Grammars have been extended with mechanisms such as references and higher-order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex computations. This paper defines an elegant and simple, zipper-based embedding of attribute grammars and their extensions as first class citizens. In this setting, language specifications are defined as a set of independent, off-the-shelf components that can easily be composed into a powerful, executable language processor. Techniques to describe automatic bidirectional transformations between grammars in this setting are also described. Several real examples of language specification and processing programs have been implemented.

Keywords: Atribute Grammars, Functional Programming, Functional Zippers, Bidirectional Transformations

1 Introduction

Attribute Grammars (AGs) [1] are a well-known and convenient formalism not only for specifying the semantic analysis phase of a compiler but also to model

⋆ This author is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ON.2 IC&DT Programa Integrado BEST CASE - Better Science Through Cooperative Advanced Synergetic Efforts, Ref. BIM-2013_BestCase_RL3.2_UMINHO and project FATBIT, Ref. FCOMP-01-0124-FEDER-020532.

complex multiple traversal algorithms. Indeed, AGs have been used not only to specify real programming languages, for example `Haskell` [2], but also to specify sophisticated pretty printing algorithms [3], deforestation techniques [4] and powerful type systems [5].

All these attribute grammars specify complex and large algorithms that rely on multiple traversals over large tree-like data structures. To express 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 data structure has to be used to glue together different traversal functions. In an imperative setting those values are stored in the tree nodes (which work as a gluing data structure), while in a declarative setting such data structures have to be defined and constructed. In an AG setting, the programmer does not have to concern himself or herself with scheduling traversals, nor on defining gluing data structures.

Recent research in attribute grammars has proceeded primarily in two directions. Firstly, AG-based systems have extended the standard AG formalism that improve the AG expressiveness. Higher-order AGs (HOAGs) [6, 7] provide a modular extension to AGs in which syntax trees can be stored as attribute values. Reference AGs (RAGs) [8] allow the definition of references to remote parts of the tree, and, thus, extend the traditional tree-based algorithms to graphs. Finally, Circular AGs (CAGs) allow the definition of fix-point based algorithms. AG systems like Silver [9], JastAdd [10], and Kiama [11] all support such extensions. Secondly, attribute grammars are embedded in regular programming languages with AG fragments as first-class values in the language: they can be analyzed, reused and compiled independently [12–14].

First class AGs provide:

i A full component-based approach to AGs where a language is specified/implemented as a set of reusable off-the-shelf components;
ii Semantic-based modularity, while some traditional AG systems use a (restricted) syntactic approach to modularity.

Moreover, by using an embedding approach there is no need to construct a large AG (software) system to process, analyse and execute AG specifications. First class AGs reuse for free the mechanisms provided by the host language as much as possible, while increasing abstraction in the host language. Although this option may also entail some disadvantages, e.g. error messages relating to complex features of the host language instead of specificities of the embedded language, the fact is that an entire infrastructure, including libraries and language extensions, is readily available at a minimum cost. Also, the support and evolution of such infrastructure is not a concern.

This paper presents a novel technique combining these two AG advances.

First, we propose a concise embedding of AGs in `Haskell`. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [15] for a purely functional environment and represent a tree together with a subtree that is the focus of attention, where that focus

may move within the tree. By providing access to any element of a tree, zippers are very convenient in our setting: attributes may be defined by accessing other attributes in other nodes. Moreover, they do not rely on any advanced feature of `Haskell` such as lazy evaluation or type classes. Thus, our embedding can be straightforwardly re-used in any other functional environment.

Second, we extend our embedding with the primary AG extensions proposed to the AG formalism and with novel techniques for AG-based bidirectionalization systems.

Attribute grammars, and their modern extensions, only provide support for specifying unidirectional transformations, despite bidirectional transformations being common in AG applications. Bidirectional transformations are especially common between abstract/concrete syntax. For example, when reporting errors discovered on the abstract syntax we want error messages to refer to the original code, not a possible de-sugared version of it. Or when refactoring source code, programmers should be able to evolve the refactored code, and have the change propagated back to the original source code.

In this work, we present the first embedding of HOAGs, RAGs and CAGs as first class attribute grammars together with a bidirectionalization system. By this we are able to express powerful algorithms as the composition of AG reusable components. We have used this approach in a number of applications, e.g., in developing techniques for a language processor to implement bidirectional AG specifications and to construct a software portal.

This paper is organized as follows: in Section 2 we motivate our approach with the introduction of both our running example and AGs. In Section 3 we introduce Zippers and explain how they can be used to embed AGs in a functional setting, and implement an AG in our setting.

Section 4 extends our running example and defines an AG implementing the scope rules for the newly defined language, with the aid of AG references. Section 5 describes the embedding of higher-order attributes as an extension to AGs and presents an example of an AG that uses this extension. In Section 6 we describe another AG extension, circularity, showing how it can be implemented with our technique, and give practical examples that build on the previous section.

In Section 7 a technique for defining a bidirectionalization system for AGs is presented, with an example providing automatic transformations between a concrete and an abstract version of our running example.

In Section 8 the reader is presented with works that relate to ours, either by having similar techniques or domains. Section 9 concludes this paper and Section 10 shows possible future research work.

2 Motivation

As a running example throughout this paper, we will describe and use the `LET` language, that could for example be used to define **let** expressions as incorporated in the functional languages `Haskell` [16] or `ML` [17].

While being a concise example, the `LET` language holds central characteristics of widely-used programming languages, such as a structured layout 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.

Below is an example of a program in the `LET` language, which corresponds to correct `Haskell` code[5].

$$\begin{aligned} \mathit{program} = \mathbf{let}\ & a = b + 3 \\ & c = 8 \\ & b = (c * 3) - c \\ \mathbf{in}\ & (a + 7) * c \end{aligned}$$

We observe that the value of *program* is $(a + 7) * c$, 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. Finally, the meaning of *program*, i.e. its value, is 208.

Our goal here is precisely to compute the semantics (i.e., the value) of a `LET` program. Implementing this computation introduces typical language processing challenges:

1. Name/scope analysis in order to verify whether or not all the variables that are used are indeed declared;
2. Semantic analysis in order to calculate the meaning of the program; this analysis incorporates name analysis through symbol table management and processing of the algebraic expressions that compose a program.

Since `LET` does not enforce a *declare-before-use* discipline, a straightforward definition of the scope analysis relies on a two traversals strategy over the abstract tree: first, to collect the declarations of variables, while at the same time searching for multiple declarations with the same variable; second, knowing the declared variables, to check whether all used identifiers have been declared.

We follow a top down strategy as we want to detect double variable declarations of the same variable during the first traversal. A top-down solution will identify the second time a variable is declared as the place where the error is located, whereas other strategies may regard other declaration to be faulty (for example, the first one).

In the following sections of the paper, we describe how the analysis of the `LET` language can be implemented in `Haskell` using our zipper-based AG embedding techniques together with the extensions we provide. For now, we start by demonstrating how to implement the scope analysis of a `LET` program as a regular AG.

The syntax of the `LET` language can be described by the following context-free grammar (CFG):

[5] To simplify our initial example, we do not consider nested *let* sub-expressions, but this extension to `LET` will be considered later in this paper.

```
(p1: Root)      Root → Let
(p2: Let)       Let  → Dcls Expr
(p3: Cons)      Dcls → Name Expr Dcls
(p4: Empty)     Dcls → ϵ
(p5: Plus)      Expr → Expr Expr
(p6: Minus)     Expr → Expr Expr
(p7: Times)     Expr → Expr Expr
(p8: Divide)    Expr → Expr Expr
(p9: Variable)  Expr → Name
(p10: Constant) Expr → Number
```

This grammar contains a `Root` non-terminal which is the start symbol of the grammar, a `Let` non-terminal that contains a list of declarations (`Dcls`) and an expression that corresponds to the meaning of the program (`Expr`). `Dcls` can have two forms: can be composed of a variable name (`Name`), an expression (`Expr`) and another declaration (`Dcls`) or can represent an empty list.

AGs themselves consist of an extension to CFGs, in the sense that they use the same formalism that is used to define the syntax of a language, CFGs, but semantics are introduced to define computations. Therefore, attributes are a set of intermingled computations implemented throughout the grammar.

There are two types of attributes in an AG: inherited and synthesized attributes. The difference between them is the way they traverse the tree: while the former perform computations from the top to the bottom of the tree, the semantics of the latter traverse a tree from the bottom to the top. As we will see, typically the meaning of an AG, i.e., its final result, is a combination of interconnected synthesized and inherited attributes.

The AG that we will construct in order to specify the name analysis task of the `LET` language can be split into the following semantic groups of operations, which are intermingled:

1. Capture all variable declarations before the current node is considered, which we will implement in the attribute `dcli` (declarations in). In the *program* above, if `dcli` was to be computed in the node for $b = (c * 3) - c$, it will be a list containing `a` and `c`, because these are the variables declared before this declaration. The attribute `dclo` (declarations out), returns all the declared variables in the program. Both these attributes are lists of identifiers.
2. Distribute all the declared variables in a program throughout the tree, which we will implement in the attribute `env` (environment). This will always produce the list of declared variables, regardless of the position on the tree where the attribute is accessed. `env` is a list of identifiers.
3. Calculate the list of invalid declarations, i.e., variables that have been declared twice and variables that are being used in an expression but have not been declared. For the AG that performs the scope/name analysis this attribute will constitute the meaning of the grammar, its final result, and will be called `errs` (errors). The returning type of `errs` will be a list of identifiers.

For example, for the program *faulty* that we can see below:

$$
\begin{aligned}
\mathit{faulty} = \textbf{let}\ & a = z + 3 \\
& c = 8 \\
& a = (c * 3) - c \\
\textbf{in}\ & (a + 7) * c
\end{aligned}
$$

the AG that performs the name analyis will yield as result a list containing the variables a and z, in this order. The former because is being declared twice and the latter because it is being used but was never declared. This result will be produced by the attribute *errs* with the aid of the other three.

In the definition of an AG, we use a syntax similar to the one in [18], where a definition `(p n) production {semantic rules}` is used to associate semantics with the syntax of a language. Syntax is defined by context-free grammar productions and semantics is defined by semantic rules that define attribute values. In a production, when the same non-terminal symbol occurs more than once, each occurrence is denoted by a subscript (starting from `1` and counting left to right)[6].

It is assumed that the value of the attribute `lexeme` is externally provided by a lexical analyzer to give values to terminal symbols. Also, we use the following constructions and auxiliary functions, whose syntax is taken directly from `Haskell` but have general constructions in most programming languages:

- `++` for lists concatenation
- `[]` represents an empty list
- `:` for the addition of an element to a list
- `mBIn` (`mustBeIn`), which returns a list with an element in case the element is not in a list, returns an empty list otherwise
- `mNBIn` (`mustNotBeIn`), which returns a list with an element in case the element is in the list, returns an empty list otherwise

2.1 Capturing Variable Declarations

In order to capture variable declarations, a typical solution in functional settings is to implement a recursive function that starts with an empty list and accumulates each declaration in a list while traversing it. Such function returns the accumulated list of declaration as its final result. This technique is known as accumulating parameters [19]. In AGs, accumulators are typically implemented as a pair of inherited and synthesized attributes, representing the usual argument/result pair in a functional setting. This pattern can be seen in the attributes `dcli` and `dclo` that we present next.

Capturing variable declarations is performed using a top-down strategy with the attribute `dcli`, as can be seen below:

[6] The traditional definition of AGs only permits semantic rules of the form `X.a = f(...)`, forcing the use of identity functions for constants. For clarity and simplicity, we allow their direct usage in attribute definitions.

```
(p1: Root) Root   → Let
      { Let.dcli = [] }
(p2: Let)  Let    → Dcls Expr
      { Dcls.dcli = Let.dcli }
(p3: Cons) Dcls1  → Name Expr Dcls2
      { Dcls2.dcli = Name.lexeme : Dcls1.dcli }
```

At the topmost node of a `LET` tree no variable declaration is visible. This is denoted by `dcli` being assigned the empty list on production `p1`. A `Cons` node inherits the same `dcli` that is computed for its `Let` parent, as can be seen in production `p2`. Finally, `p3` defines that when a variable is being declared, its name should be added/accumulated to the so far computed `dcli` attribute, and it is the resulting list that should be passed down.

Note that the value of the attribute `dcli` that is inherited by a `Cons` node excludes the declaration that is being made on it. As we will see below, this will help us detect double variable declarations of the same variable.

The attribute `dclo` works botton-up, and its function is to call `dcli` on the last element of the list of variable declarations. Since `dcli` returns a list of variables that are visible at the position where it is called, calling it at the bottom of the list will effectively produce the total list of variables. Similarly to `dcli`, this attribute is not present throughout the entire grammar, but only on the productions `p1-p5`. Its implementation is:

```
(p1: Root)  Root   → Let
      { Root.dclo = Let.dclo }
(p2: Let)   Let    → Dcls Expr
      { Let.dclo = Dcls.dclo }
(p3: Cons) Dcls1   → Name Expr Dcls2
      { Dcls1.dclo = Dcls2.dclo }
(p4: Empty) Dcls   → ϵ
      { Dcls.dclo = Dcls.dcli }
```

Another important remark about the attributes `dcli` and `dclo` is that they are only declared for the productions `p1-p3` (and `p4`, in the case of `dclo`), and not for the entire CFG. This is typical of AGs as sometimes, as is the case, specific semantics depend only on specific parts of the tree/language. The full pattern of attribute calculation can be seen for a simple tree in Figure 1.

2.2 Distributing Variable Declarations

One important part of the semantics of analysing the scope rules of a `LET` program is distributing the information regarding variable declarations throughout the entire tree. This is important because it will allow us, when searching for the usage of undeclared identifiers, to use an attribute that we are sure carries all the variable declarations in the entire program.

Distributing variable declarations is performed by the inherited attribute `env`, whose definition is:

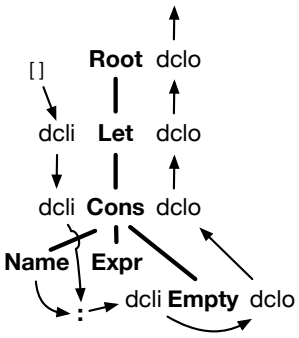


Fig. 1. The relation between the inherited attribute `dcli` and the synthesized attribute `dclo`, implementing an accumulation pattern.

```
(p1: Root)  Root   → Let
      { Let.env = Let.dclo }
(p2: Let)   Let    → Dcls Expr
      { Dcls.env = Let.env
      , Expr.env = Let.env }
(p3: Cons) Dcls1   → Name Expr Dcls2
      { Expr.env  = Dcls1.env
      , Dcls2.env = Dcls2.env }
(p5: Plus)   Expr1 → Expr2 Expr3       // Productions p5-p8
(p6: Minus)  Expr1 → Expr2 Expr3       // have the same
(p7: Times)  Expr1 → Expr2 Expr3       // semantic equations
(p8: Divide) Expr1 → Expr2 Expr3
      { Expr2.env = Expr1.env
      , Expr3.env = Expr1.env }
```

The attribute `env` is present everywhere in the tree with the same value. The equations go all the way up the tree to obtain the `dclo` attribute of the root. The inherited attribute `env` and its relation with `dclo` can be seen in Figure 2.

2.3 Calculating Invalid Identifiers

The meaning of an AG is typically given as the value of one of its synthesized attributes. When implementing scope analysis for the `LET` language, we want to derive a list of *invalid* identifiers, where by invalid we mean identifiers that are either declared twice, or are used but not declared.

This list represents the meaning of the grammar and is calculated by the attribute `errs` whose definition is:

```
(p1: Root) Root  → Let
      { Root.errs = Let.errs }
(p2: Let)  Let   → Dcls Expr
```

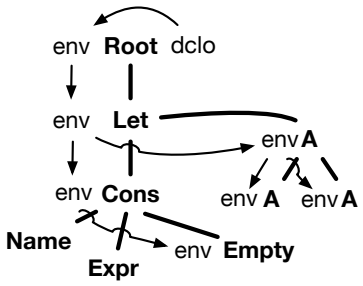


Fig. 2. The inherited attribute `env`, distributing the environment throughout the tree.

```
      { Let.errs = Dcls.errs ++ Expr.errs }
(p3: Cons) Dcls1 → Name Expr Dcls2
      { Dcls1.errs = (mNBIn Name.lexeme Dcls1.dcli)
                     ++ Expr.errs ++ Dcls2.errs }
(p4: Empty) Dcls → ϵ
      { Dcls.errs = [] }
(p5: Plus)   Expr1 → Expr2 Expr3       // Productions p5-p8
(p6: Minus)  Expr1 → Expr2 Expr3       // have the same
(p7: Times)  Expr1 → Expr2 Expr3       // semantic equations
(p8: Divide) Expr1 → Expr2 Expr3
      { Expr1.errs = (Expr2.errs) ++ (Expr3.errs) }
(p9: Variable)  Expr → Name
      { Expr.errs = mBIn Name.lexeme Expr.env }
(p10: Constant) Expr → Number
      { Expr.errs = [] }
```

This attribute is propagated up the tree and its semantics are only relevant for the productions `p3` and `p9` where the equations use the attributes `dcli` and `env` to check for double variable declarations and use of undeclared identifiers, respectively.

In the production `p3`, `errs` checks if a variable has been declared before. This is easily done with the attribute dcli. Recall that this attribute returns a list of variable declarations up to a certain tree node, which means that `errs` uses the auxiliary function `mNBIn` to see if the current variable is not present in the list produced by `dcli`.

Whenever variables are used inside expressions we have to see if they have been declared before. This means that the semantics for `errs` in the production `p9` checks the list produced by `env` (containing all the variables of the program) and to see if the variable is present.

In Figure 3 we can see how this attribute is defined throughout the abstract tree of `LET` and how it relates to the attributes `dcli` and `env`.

Sumarizing the AG formalism, attribute occurrences are calculated by invocations of small semantic functions that depend on the values of other attribute

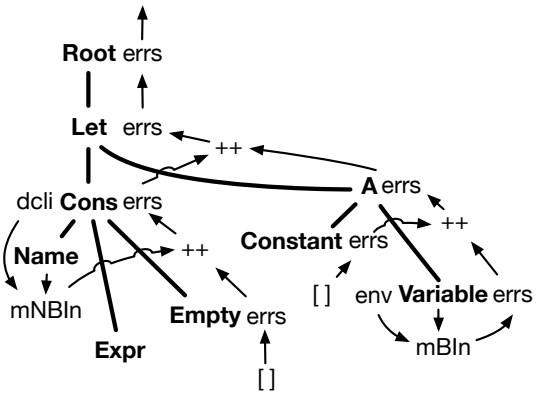


Fig. 3. The synthesized attribute `errs`.

occurrences. The calculations are specified by simple semantic equations associated with the grammar productions of the language. This approach makes the programmer's work easier as it decomposes complex computations into smaller parts that are easier to implement and to reason about than if the full computation was considered.

This is the kind of behavior we aim to add to a functional setting by embedding AGs. In the next section we will see how zippers can be used to embed this AG in the functional language `Haskell`.

3 Embedding Attribute Grammars

Our approach to the definition of attribute grammars envisions their implementation directly in `Haskell`. In this section we use the `LET` language in order to demonstrate how this embedding is achieved. Our approach relies on the concept of functional zippers, that we present next.

3.1 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.

Moreover, conceptually, the idea of a functional zipper is applicable in (at least) other functional programming language besides `Haskell`, which means that our embedding can be achieved in other functional environments as well.

In our work we have used the generic zipper `Haskell` library of Adams [20]. This library works for both homogeneous and heterogeneous data ypes. The

library can traverse any data type that has an instance of the `Data` and `Typeable` type classes [21].

In order to illustrate how we may use zippers, we consider the following `Haskell` data type straightforwardly obtained from the abstract syntax of the `LET` language:

```
data Root = Root Let
data Let  = Let Dcls Expr
data Dcls = Cons String Expr Dcls
          | Empty
data Expr = Plus     Expr Expr
          | Minus    Expr Expr
          | Times    Expr Expr
          | Divide   Expr Expr
          | Variable String
          | Constant Int
```

A `LET` program can be expressed as an element of *Root*. For example, the `LET` *program* presented in the previous section is represented as:

```
Root (Let
        (Cons "a" (Plus (Variable "b") (Constant 5))
         Cons "c" (Constant 8)
         Cons "b" (Minus (Times (Variable "c") (Constant 3))
                         (Variable "c"))
         Empty)
        (Times (Plus (Variable "a") (Constant 7)) (Variable "c"))
     )
```

Typical of Zipper libraries, the one we use provides a set of functions, such as *up*, *down*, *left* and *right* that allow the programmer to easily navigate throughout a structure. The function *getHole* returns the subtree which is the current focus of attention.

In our setting, on top of the zipper library of Adams [20] we have implemented several simple abstractions that facilitate the embedding of attribute grammars. In particular, we have defined:

- `(.$) :: Zipper a → Int → Zipper a`, for accessing any child of a structure given by its index starting at `1`
- `parent :: Zipper a → Zipper a`, to move the focus to the parent of a concrete node
- `(.|) :: Zipper a → Int → Bool`, to check whether the current location is a sibling of a tree node
- `constructor :: Zipper a → String`, which returns a textual representation of the data constructor which is the current focus of the zipper.

With these functions defined, we can easily wrap a structure in a Zipper to navigate through it. Recall that we are using a generic zipper library, so no additional coding is necessary to accommodate a particular structure.

In order to see the Zippers in action, we may use the algebraic expression which represents the meaning of the previously defined *program*:

$$expr = Times\ (Plus\ (Variable\ \texttt{"a"})\ (Constant\ 7))\ (Constant\ 8)$$

and easily wrap it in a zipper,

$$expr\_zipper = toZipper\ expr$$

check if the constructor of the current node is `Times`,

$$constructor\ expr\_zipper \equiv \texttt{"Times"}$$

go to the first child and check the constructor name,

$$second\_child = expr\_zipper\ .\$\ 1$$
$$constructor\ second\_child \equiv \texttt{"Plus"}$$

and do the same with the parent,

$$constructor\ (parent\ second\_child) \equiv \texttt{"Times"}$$

Finally, we can define functions such as `lexeme_Constructor_1 :: Zipper a -> Int`, where

$$lexeme\_Constructor\_1\ (second\_child\ .\$\ 2) \equiv 7$$

extracts information from the zipper, simulating a standard lexer. Throughout this paper, the name of these lexeme functions will always have the form *lexeme_constructor_child*, where *constructor* corresponds to the current data constructor and *child* corresponds to the number of the child whose lexeme we want to obtain.

As we will see in the next section, despite their simplicity the mechanisms provided by the zippers to navigate through structures and the abstractions we have created on top of them are sufficiently expressive to embed AGs in a functional setting.

3.2 `LET` as an Embedded Attribute Grammar

With zippers introduced, we now show how the AG presented in Section 2 can be implemented in the functional language `Haskell`, starting with the attribute `dcli`. This is an inherited attribute that goes top-down in the tree, collecting declarations. We can define it in `Haskell` as:

$$dcli :: Zipper\ Root \rightarrow [String]$$
$$dcli\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of}$$

```
"Root" → []
_      → case (constructor (parent ag)) of
           "Cons"  → (dcli (parent ag))
                       ++ [lexeme_Cons_1 (parent ag)]
           "Empty" → dcli (parent ag)
```

The value of `dcli` in the top-most position, `Root`, corresponds to the empty list. For all the other positions of the tree, we have to test if the parent is a declaration, indicated by a `Cons` parent, in which case we add the value of the declared variable, or if it is anything else, in which case we just return whatever the value of `dcli` is in the parent node.

Note that the usage of the _ means that we are declaring `dcli` for all the tree nodes, whereas in the AG defined in Section 2 it is only declared for a few productions. This is not a problem, as even if `dcli` is only defined for a few production, which we can always do in our setting, it behaves exactly the same everywhere, so we are simplifying its definition.

Next, we present the attribute *dclo*:

```
dclo :: Zipper Root → [String]
dclo ag = case (constructor ag) of
  "Root"  → dclo (ag .$ 1)
  "Let"   → dclo (ag .$ 1)
  "Cons"  → dclo (ag .$ 3)
  "Empty" → dcli ag
```

This attribute collects the whole list of declared variables. Therefore, it goes down the tree until the bottom-most position where it is equal to the attribute `dcli`. Recall that `dcli` produces a list with all the declared identifiers up to the position where it is being called, which in the bottom-most position will equal the entire list of declared variables.

A similar approach is used when defining `env`:

```
env :: Zipper Root → [String]
env ag = case (constructor ag) of
  "Root" → dclo ag
  _      → env (parent ag)
```

where we defined the attribute for the top most production and then instruct it to go up as far as possible. These types of attributes are very common in AG specifications as a method of distributing information everywhere in the tree, with some AG systems providing specific constructs to allow this type of simpler implementations (such as `autocopy` in Silver [9] and references to remote attributes in LRC [22]). In this embedding, we can elegantly implement this feature using standard primitives from the hosting languague.

The last attribute we define is the one that represents the actual meaning of the AG, `errs`:

```
errs :: Zipper Root → [String]
errs ag = case (constructor ag) of
  "Root"     → errs (ag .$ 1)
  "Let"      → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Cons"     → mNBIn (lexeme_Cons_1 ag) (dcli ag)
                 ++ errs (ag .$ 2) ++ errs (ag .$ 3)
  "Empty"    → []
  "Plus"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Divide"   → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Minus"    → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Time"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Variable" → mBIn (lexeme_Variable_1 ag) (env ag)
  "Constant" → []
```

The only really interesting parts of the definition of this attribute are in `Cons`, where we test if a declaration is unique, i.e., if it has not been declared so far, and `Variable`, where we test if a variable that is currently being used has been declared somewhere in the program. The other parts of `errs` either go down the tree checking for errors, or immediatly say there are no errors in that specific position, as happens in `Empty` and `Constant`.

The semantic functions `mNBIn` and `mBIn` are easily defined in `Haskell`:

```
mBIn :: String → [String] → [String]
mBIn name []       = [name]
mBIn name (n : es) = if (n ≡ name) then [] else mBIn name es
mNBIn :: String → [String] → [String]
mNBIn tuple []       = []
mNBIn a1 (a2 : es) = if (a1 ≡ a2) then [a1] else mNBIn a1 es
```

A difference between our embedding and the traditional definition of AG is that in the former, an attribute is defined as a semantic function on tree nodes, while in the latter the programmer defines on one production exactly how many and how attributes are computed. Nevertheless, we argue that this difference does not impose increasing implementation costs as the main advantages of the attribute grammar setting still hold: attributes are modular, their implementation can be sectioned by sites in the tree and as we will see inter-attribute definitions work exactly the same way.

The structured nature of our embedding might provide an easier setting for debugging as the entire definition of one attribute is localized in one semantic function. Furthermore, we believe that the individual attribute definitions in our embedding can straightforwardly be understood and derived from their traditional definition on an attribute grammar system, as can be observed comparing the attribute definitions in the previous section with the ones in this section.

An advantage of the embedding of domain-specific languages in a host language is the use of target language features as native. In our case, this applies, e.g., to the `Haskell` functions `++` and `:` for list concatenation and addition,

whereas in specific AG systems the set of functions is usually limited and predefined. Also, regarding distribution of language features for dynamic loading and separate compilation, it is posible to divide an AG in modules that, e.g., may contain contain data types (representing the grammar) and functions (representing the attributes).

4 References in Attribute Grammars

In the last section we have seen how zipper-based constructs can be used to implement AG in a functional setting. Here we will show how an AG extension, References, can be embedded as well.

Reference Attribute Grammars (RAGs) were first introduced by Magnusson and Hedin [23]. They allow attribute values to be references to arbitrary nodes in the tree and attributes of the referenced nodes to be accessed via the references. Apart from providing access to non-local attribute occurrences, this extension is also important for adding extensibility to AGs and simplifying the implementation of future improvements to it.

We shall start by extending the `LET` programming language with nested expressions, allowing multiple-scoped declarations of all name entities that are used in a program. Having a hierarchy with multiple scopes is very common in real programming languages, such as in *try blocks* in *Java* and nested procedures in *Pascal*, and the example we present next is compilable `Haskell` code:

$$
\begin{aligned}
\mathit{program} = \mathbf{let}\ & a = b + 3 \\
& c = 8 \\
& w = \mathbf{let}\ c = a * b \\
& \quad\ \mathbf{in}\ \ c * b \\
& b = (c * 3) - c \\
\mathbf{in}\ & c * w - a
\end{aligned}
$$

This example works similarly to those in previous sections, but this time the variable w contrasts with the others as it is defined by a nested block. Because we have a nested definition, we have to be careful: as the variable b is not defined in this inner block, its value will come from the outer block expression $(c*3)-c$, but c is defined both in the inner and in the outer block. This means that we must use the inner c (defined to be $a * b$) when calculating $c * b$ but the outer c (defined to be 8) when calculating $(c * 3) - c$.

Syntactically, the language does not change much. We only need to add a new construct to the data type *Dcls*:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Dcls} = {} & \mathit{ConsLet}\ \mathit{String}\ \mathit{Let}\ \ \mathit{Dcls} \\
\mid {} & \mathit{Cons}\ \ \ \mathit{String}\ \mathit{Expr}\ \mathit{Dcls} \\
\mid {} & \mathit{Empty}
\end{aligned}
$$

with *ConsLet* representing nested blocks of code. The rest remains exactly as we presented it in Section 3.

Semantically however, this adds complications to defining the scope rules of a `LET` program. Nested blocks prioritize variable usage on declarations in the same block, only defaulting to outer blocks when no information is found. Furthermore, variables names are not exclusive throughout an entire `LET` program: they can be defined with the same name as long as they exist in different blocks.

Typical solutions to this problem involve a complex algorithm where each block is traversed twice. This implies that for each inner block, a full traversal of the outer block is necessary to capture variable declarations. These are then used in the inner block together with a first traversal of the inner block to capture the total number of variables that are needed to check for scope/name rules. Only after the inner block is checked can the second traversal of the outer block be performed and only then can wrong declarations and use of identifiers be detected. The idiosyncrasies of implementing the analysis for nested blocks is further explained in previous work [24].

In order to be able to detect multiple declarations, we will need to know the level in which a variable is declared. We will therefore start with a new attribute, *lev*:

```
lev :: Zipper a → Int
lev ag = case (constructor ag) of
  "Root" → 0
  "Let"  → case (constructor (parent ag)) of
             "ConsLet" → lev (parent ag) + 1
             _         → lev (parent ag)
  _      → lev (parent ag)
```

The top of the tree and the main block will be at level 0. For *Let*, we have to inspect the parent node. If it is a *ConsLet*, we are in a nested block and we have to increment the level value. For all the other cases, we use a strategy that we have seen before: we use the wildcard matching construct _ to define *lev* to be equal to its value in the parent node. Again, we could define *lev* independently for every tree node, but using this feature of the hosting language simplifies the implementation and makes our work easier.

Next we present the attribute *dcli* which has the same aim as the attribute with the same name presented in the previous section. Because we need to access the level of declarations to check for scope errors in a program, the new *dcli* holds a list with both the variable names and references to the declaration sites of those variables.

```
dcli :: Zipper Root → [(String, Zipper Root)]
dcli ag = case (constructor ag) of
  "Root" → []
  "Let"  → case (constructor (parent ag)) of
             "Root"    → dcli (parent ag)
             "ConsLet" → env (parent ag)
  _      → case (constructor (parent ag)) of
             "Cons"    → dcli (parent ag)
```

```
                    ++ [(lexeme_Cons_1 (parent ag), parent ag)]
"ConsLet" → dcli (parent ag)
                    ++ [(lexeme_ConsLet_1 (parent ag), parent ag)]
"Empty"   → dcli (parent ag)
```

The semantics are very similar to the previous version with two big differences: first, the return type of *dcli* is now $[(String, Zipper\ Root)]$ and second, the initial list of declarations in a nested block is the total environment of the outer one (see attribute *env* in the previous code).

Thus, references are implemented as zippers whose current focus is the site of the tree we want to reference. What this means in practice is that we can now use another characteristic of our embedding, which is attributes being first-class citizens in the target language, to re-define the semantic function *MNBIn* as:

```
mNBIn :: (String, Zipper Root) → [(String, Zipper Root)] → [String]
mNBIn tuple []                 = []
mNBIn (a1, r1) ((a2, r2) : es) = if (a1 ≡ a2) ∧ (lev r1 ≡ lev r2)
                                 then [a1]
                                 else mNBIn (a1, r1) es
```

Now *mNBIn* checks the both is the variable name and if the declaration level match, extending scope rules to check for declarations only in the same scope, as double declarations in different blocks are allowed.

In this example, references are also important to support extensibility of the AG. If all we wanted to do was check scope rules then it would be enough to carry declaration levels in the environment. However, carrying references makes it possible to easily extend to checking that the use of a variable conforms to other properties of its declaration. For example, if we were to extend `LET` to include type information, the declared type could be made available as an attribute of the declaration reference. Similarly, an interactive facility that displays the defining expression for a variable use could be implemented easily by following the reference.

The attribute *errs* follows the same semantics as we have seen in the previous sections, with the addition of a new case to support nested blocks.

```
errs :: Zipper Root → [String]
errs ag = case (constructor ag) of
  "Root"     → errs (ag .$ 1)
  "Let"      → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Cons"     → mNBIn (lexeme_Cons_1 ag, ag) (dcli ag)
                 ++ errs (ag .$ 2) ++ errs (ag .$ 3)
  "ConsLet"  → mNBIn (lexeme_ConsLet_1 ag, ag) (dcli ag)
                 ++ errs (ag .$ 2) ++ errs (ag .$ 3)
  "Empty"    → []
  "Plus"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Divide"   → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Minus"    → errs (ag .$ 1) ++ errs (ag .$ 2)
```

```
"Time"     → errs (ag .$ 1) ++ errs (ag .$ 2)
"Variable" → mBIn (lexeme_Variable_1 ag) (env ag)
"Constant" → []
```

The attributes *env* and *dclo* remain unchanged, with the former distributing the environment throughout the tree and the latter forcing *dcli* to compute the complete list of declared variables.

To summarize this section, references to non-local sites in the tree are represented by zippers whose focus is on the respective site. This capability together with attributes being first-class citizens in the embededing language provides the user with multiple ways to use AGs when developing programs in a functional setting. In the next section we will see how another important extension, Higher-order AGs (HOAGs), is implemented in our setting.

5 Higher-Order Attribute Grammars

Higher-Order Attribute Grammars (HOAGs) were first introduced by Vogt *et al.* and introduce a setting where the structure tree can be expanded as the result of attribute computations [6]. The new parts of the tree can themselves have semantics defined using attributes.

Comparing with traditional AGs, HOAGs provide a setting where:

- attributes define new trees whose semantics are defined as a new set of attributes occurrences, and
- computations in the original tree can depend on attributes from the new trees.

We have already defined the scope rules of `LET` in our setting with the help of references. These aid in analysing the level of variable declarations and detect errors in the declaration and use of these identifiers. In this section we will continue defining semantics for `LET`, but this time we will define an auxiliary structure, more precisely a symbol table, which will be an HOAG where certain semantics will be defined as attributes.

Since we have already defined and implemented the scope rules for `LET`, we can relax when defining and implementing the semantics of the symbol table (and when solving it, as we will see in the next section) and rely on the fact that our scope rules ensure the program is semantically correct. For example, we can freely search for a variable being used in an expression with the assurance that it is well declared and we will find it somewhere.

We choose to use nested symbol tables whose structure closely resembles the scoping structure of `LET` programs. The following data types define that structure:

```
data RootHO = RootHO DclsHO Expr
data DclsHO = ConsHO     String  IsSolved Expr    DclsHO
            | ConsLetHO  String  IsSolved DclsHO DclsHO
```

```
                | NestedDclsHO DclsHO Expr
                | EmptyHO
data IsSolved = IsSolved Int | NotSolved
```

These three data types have the following functionality:

- *RootHO* contains the list of declarations and the final expression to be solved.
- *DclsHO* has two data constructors, *ConsHO* and *ConsLetHO*, for variable declarations and nested blocks respectively. These constructors both carry the variable name as a *String*, and both recursively define *DclsHO*. However, whereas the former has an expression, the latter carries nested information. The data constructor *NestedDclsHO* carries information that corresponds to nested blocks: an expression which is the meaning of the block, and a list with the nested declarations.
- *IsSolved* is added to avoid continuous checks of completion of nested blocks and to facilitate accessing their meaning: once a nested block or an expression is solved we change this constructor from *NotSolved* to *IsSolved* and add its value.

Next, we present the attributes that create the higher order symbol table from an abstract tree of `LET`. We will need two attributes to do so: one that creates the whole list with type *DclsHO*, and another that creates the root of the higher order tree that constitutes the symbol table. We shall start by presenting the latter first:

```
createSTRoot :: Zipper Root → RootHO
createSTRoot ag = case (constructor ag) of
  "Root" → RootHO (createST ag) (lexeme_Let_2 (ag .$ 1))
```

Here, the first argument of *RootHO* is the attribute that creates the symbol table and *lexeme_Let_2* (*ag* .$ 1) lexes the expression that constitutes the meaning of this program. Please recall that the abstract tree for `LET` has the form:

```
    Root
     |
    Let
    / \
   /   \
Dcls   Expr
 |
...
```

and therefore to lexe top level expression we have to go to the first child of *Root*, a *Let*, and then we can apply the lexe function to the second child, *Expr*, which is why we write *lexeme_Let_2* (*ag* .$ 1).

The second attribute needed to construct the symbol table goes through the whole program and captures declarations and nested blocks:

```
createST :: Zipper Root → DclsHO
createST ag = case (constructor ag) of
  "Root"     → createST (ag .$ 1)
  "Let"      → createST (ag .$ 1)
  "Cons"     → let var  = lexeme_Cons_1 ag
                   expr = lexeme_Cons_2 ag
               in  ConsHO var NotSolved expr (createST (ag .$ 3))
  "ConsLet" → let var = lexeme_ConsLet_1 ag
                   nested = let nested = createST     (ag .$ 2)
                                expr   = lexeme_In_1 ((ag .$ 2) .$ 2)
                            in  NestedDclsHO nested expr
               in  ConsLetHO var NotSolved nested (createST (ag .$ 3))
  "Empty"    → EmptyHO
```

The most interesting parts of this attribute are the semantics for *Cons* and *ConsLet*. For these we extract the necessary information to construct the symbol table, declare all the elements as *NotSolved* and recursively call *createST* where needed, i.e., always in the tail of the program, following the recursive structure of the language, and when nested blocks are found.

With *createSTRoot* and *createST* defined we have now created a new tree on which attributes can be defined. The new higher order tree can be easily transformed into an HOAG in our setting by wrapping it inside a zipper, after which we can define attribute computations such as the ones we have seen in the previous sections. For example, we can define semantics that check if a variable is solved in the symbol tree, starting with the attribute *isVarSolved*.

```
isVarSolved :: String → Zipper RootHO → Bool
isVarSolved name ag = case (constructor ag) of
  "RootHO"       → auxIsVarSolved name ag
  "NestedDclsHO" → auxIsVarSolved name ag
  _              → isVarSolved     name (parent ag)
```

isVarSolved is an inherited attribute that takes as argument the variable name as a string and a zipper for the current focus. The equations search upwards either to the root of the tree (*RootHO*) or to the root of the nearest nested block (*NestedDclsHO*). We do so to ensure that when the *auxIsVarSolved* attribute is called we are searching in the whole block, starting in its top most position:

```
auxIsVarSolved :: String → Zipper RootHO → Bool
auxIsVarSolved name ag = case (constructor ag) of
  "RootHO"       → auxIsVarSolved name (ag .$ 1)
  "NestedDclsHO" → auxIsVarSolved name (ag .$ 1)
  "ConsHO"       → if   lexeme_ConsHO_Var_1 ag ≡ name
                   then auxIsVarSolved name (ag .$ 2)
                   else auxIsVarSolved name (ag .$ 4)
  "ConsLetHO"    → if   lexeme_ConsLetHO_Var_1 ag ≡ name
                   then auxIsVarSolved name (ag .$ 2)
```

```
                  else  auxIsVarSolved name (ag .$ 4)
"IsSolved"      → True
"NotSolved"     → False
"EmptyHO"       → oneUpIsVarSolved name ag
```

The synthesized *auxIsVarSolved* attribute goes down the tree and searches for the declaration of the specified variable. Here, the fact that the variables are defined as a nested block or as an expression is not important, as in either cases we can use the constructor *isSolved*. At the bottom we encounter the production *EmptyHO* and the *oneUpIsVarSolved* attribute is called.

```
oneUpIsVarSolved :: String → Zipper RootHO → Bool
oneUpIsVarSolved name ag = case (constructor ag) of
  "NestedDclsHO" → isVarSolved        name (parent ag)
  _              → oneUpIsVarSolved name (parent ag)
```

The only function of *oneUpIsVarSolved* is to go up as far as possible, jump to a parent block, and restart the whole process again with *isVarSolved*.

Because we have already defined the scope rules analysis for `LET` in Section 4, the semantics for the symbol table can be simplified because we know we are dealing with a valid program. For example, the attribute *oneUpIsVarSolved* is never defined for *RootHO*, because we don't know how many nested blocks we have to search to find a variable declaration but we are sure that we will find one at least in the main block of the program.

One important note about the three attributes *isVarSolved*, *auxIsVarSolved* and *oneUpIsVarSolved* is their interdependence. The first two, *isVarSolved* and *auxIsVarSolved*, search for a variable in a block, with the former going to the topmost position of a block and the latter going top-down in search of the variable. In case nothing is found, *oneUpIsVarSolved* goes up one block. The relation between these three attributes is illustrated in Figure 4.

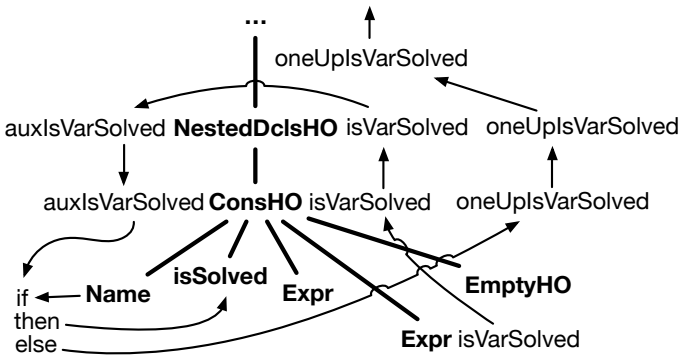


Fig. 4. Dependency between *isVarSolved*, *auxIsVarSolved* and *oneUpIsVarSolved*.

We have defined these attributes for a tree created by an AG in the first place, thereby creating an HOAG. In a traditional approach we would define computations on the symbol table using semantic functions that sit outside the AG. By using an HOAG we make it possible to define those computations themselves using attributes. For example, the following attributes calculate the value of a solved variable given a resolved symbol table.

```
getVarValue :: String → Zipper RootHO → Int
getVarValue name ag = case (constructor ag) of
  "RootHO"        → auxGetVarValue name ag
  "NestedDclsHO" → auxGetVarValue name ag
  _               → getVarValue name (parent ag)

auxGetVarValue :: String → Zipper RootHO → Int
auxGetVarValue name ag = case (constructor ag) of
  "RootHO"        → auxGetVarValue name (ag .$ 1)
  "NestedDclsHO" → auxGetVarValue name (ag .$ 1)
  "ConsHO"        → if   lexeme_ConsHO_Var_1 ag ≡ name
                    then auxGetVarValue name (ag .$ 2)
                    else auxGetVarValue name (ag .$ 4)
  "ConsLetHO"     → if   lexeme_ConsLetHO_Var_1 ag ≡ name
                    then auxGetVarValue name (ag .$ 2)
                    else auxGetVarValue name (ag .$ 4)
  "IsSolved"      → lexeme_IsSolved_1 ag
  "EmptyHO"       → oneUpGetVarValue name ag

oneUpGetVarValue :: String → Zipper RootHO → Int
oneUpGetVarValue name ag = case (constructor ag) of
  "NestedDclsHO" → getVarValue      name (parent ag)
  _               → oneUpGetVarValue name (parent ag)
```

These definitions operate in a similar manner to the attributes we have already seen to check is a variable is solved, with the same type of interdependence and semantics between the three.

We have shown that we can create an HOAG representing a symbol table of a `LET` program, and how semantics can be defined for it. However, from this symbol table we can not directly calculate the meaning of a `LET` program. We still need to resolve the symbol table and find the exact meaning of each variable.

In the next section we will see how an extension that allows circular computations of attributes can be used to gracefully implement the resolution of the symbol table and finally calculate the meaning of a program.

6 Circular Attribute Grammars

An Attribute Grammar is called circular (CAG) if it has an attribute that depends transitively on itself. CAGs allow circular dependencies between attributes

on the condition that a fixed-point can necessarily be reached for all possible attribute trees. This is guaranteed if the circular dependencies between the attribute(s) are defined by a monotonic computation that necessarily reaches a stopping condition.

Previous work has shown practical, well-known applications of AGs with circular definitions of attributes, including applications in different domains such as data-flow analysis or code optimizations [25–27]. Another example where CAGs are useful is analysing variable declarations such as the ones in the following `LET` program.

$$\begin{aligned} \mathit{program} = \mathbf{let}\ & x = y \\ & z = 1 \\ & y = z \\ \mathbf{in}\ & x + y + 1 \end{aligned}$$

In *program*, the textual order in which variables are declared is different to the order implied by their data dependence and by which variable evaluation is defined. If these orders were the same then evaluation of a `LET` program would be much simpler and would require only an algorithm that analyses the variables in textual order.

One way to solve this "out of order" problem is to first construct a symbol table as in Section 5 and then define attributes that circularly iterate over that structure. In other words, repeatedly calculate attributes on the symbol table until all of the variables are solved.[7]

Therefore, to process this `LET` program we need a circular, fixed-point evaluation strategy. The general idea is to start with a bottom value, $\perp$, and compute approximations of the final result until it is not changed anymore, that is, the least fixed point: `x = ⊥; x = f(x); x = f(f(x)); ...` is reached. To guarantee the termination of this computation, it must be possible to test the equality of the result (with $\perp$ being its smallest value). All in all, the computation will return a final result of the form `f(f(...f(⊥)...))`.

Of course, this solution might produce an infinite loop in cases where circular variable declarations are present, such as in this program:

$$\begin{aligned} \mathit{program} = \mathbf{let}\ & x = y \\ & y = x \\ \mathbf{in}\ & x + y + 1 \end{aligned}$$

There is no fixed point in this case. Fortunately, this kind of program is invalid so our computations do not take them into account.

In order to implement fixed point computations in our embedding we use the following *fixed_point* function.

[7] In this particular case, we do not necessarily need to reach a state where all the variables are solved. We can stop it, for example, whenever the top expression of a program has only values and no variables. As we will see, our setting allows custumisation of the fixed point calculation to stop the circular attribute evaluation for cases like this.

```
fixed_point :: Zipper a → (Zipper a → Bool) → (Zipper a → b)
                 → (Zipper a → Zipper a) → b
fixed_point ag cond calc incre = if   cond ag
                                 then calc  ag
                                 else fixed_point (incre ag) cond calc incre
```

The arguments of this function are as follows:

- *ag* :: *Zipper a* : the tree on which we want to compute the fixed point computation. In the case of the symbol table HOAG presented in the previous section *ag* would be a value of type *Zipper RootHO*.

- *cond* :: *Zipper a* → *Bool* : a function that takes a tree and returns a Boolean value that signals when the fixed point has been achieved.

  As we have seen above, the traditional definition of the fixed point states that computation stops when equality is achieved, i.e., when the result of a computation is equal to its input. Here we have extended this definition to use any user-defined Boolean-value attribute to define the stopping condition as it can allow more powerful and/or efficient computations to be defined.

  In the case of `LET`, the user can define an attribute that not only checks for the total resolution of all variables but also checks if the expression that represents the meaning of a program does not require symbol table resolution, for example, because it only contains values.

- *calc* :: *Zipper a* → *b* : a computation that is performed after the fixed point has been reached. For example, the computation might calculate the value of the top expression of the symbol table after all declarations have been resolved. The identity function can be passed as *calc* if the user does not want an additional computation to be applied after the circular computation.

- *incre* :: *Zipper a* → *Zipper a* : an attribute that performs an iteration of the circular computation. It returns a new structure that is checked using *cond* and, if a fixed point is not reached, is used as the input for the next iteration.

The type *b* is the type of the final result of the circular computation, provided by *calc*. If the identity function is used, *b* will be *Zipper a*.

Returning to the running example of the previous section, we have created a symbol table as an HOAG and defined a semantics for it that can be applied to obtain a value assuming that all symbols have been solved. We now show how a symbol table can be resolved using circular, fixed-point based computation. To do so, we have to define the attributes that will be used as arguments of *fixed_point*, starting with the attribute that ends the circular computation by defining the fixed point (*cond*):

```
isSolved :: Zipper RootHO → Bool
isSolved ag = case (constructor ag) of
  "RootHO"        → isSolved (ag .$ 1) ∨ isSolved (ag .$ 2)
```

```
"NestedDclsHO" → isSolved (ag .$ 1)
"ConsHO"       → isSolved (ag .$ 2) ∧ isSolved (ag .$ 4)
"ConsLetHO"    → isSolved (ag .$ 2) ∧ isSolved (ag .$ 4)
"EmptyHO"      → True
"IsSolved"     → True
"NotSolved"    → False
"Plus"         → isSolved (ag .$ 1) ∧ isSolved (ag .$ 2)
"Divide"       → isSolved (ag .$ 1) ∧ isSolved (ag .$ 2)
"Minus"        → isSolved (ag .$ 1) ∧ isSolved (ag .$ 2)
"Time"         → isSolved (ag .$ 1) ∧ isSolved (ag .$ 2)
"Variable"     → isVarSolved (lexeme_Variable_1 ag) ag
"Constant"     → True
```

This attribute has very simple semantics: it just goes through the tree and checks if either all variables are solved, or the topmost expression representing the meaning of the program is already solved without reference to variables (*RootHO* case). From this point on, the attribute tries to check if all variables are solved, through the constructor *IsSolved*, or if an expression contains only constants or solved variables.

The next attribute to be defined is *solveSTRoot*, which together with *solveST* performs one iteration of the fixed point computation, solving as many variables as possible.

```
solveSTRoot :: Zipper RootHO → Zipper RootHO
solveSTRoot ag = let solved_decl = solveST (ag .$ 1)
                     top_expr    = lexeme_RootHO ag
                 in toZipper (RootHO solved_decl top_expr)
```

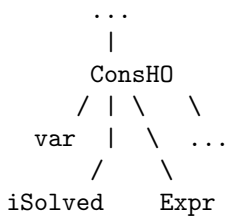
In this definition the topmost expression is ignored and *solveST* tries to solve the declarations. (If the meaning expression only contains constants *isSolved* will notice and terminate the fixed point computation before *solveSTRoot* is called.)

The attribute *solveST* considers a list of declarations and solves as many as can be solved in a single pass.

```
solveST :: Zipper RootHO → DclsHO
solveST ag = case (constructor ag) of
  "ConsHO" →
    if     (¬ isSolved (ag .$ 2) ∧ isSolved (ag .$ 3))
    then let var  = lexeme_ConsHO_1 ag
             res  = IsSolved (calculate (ag .$ 3))
             expr = lexeme_ConsHO_A ag
         in  ConsHO var res expr (solveST (ag .$ 4))
    else let var  = lexeme_ConsHO_1 ag
             res  = lexeme_ConsHO_2 ag
             expr = lexeme_ConsHO_3 ag
         in  ConsHO var res expr (solveST (ag .$ 4)
  "ConsLetHO" →
```

```
if     (¬ isSolved (ag .$ 2) ∧ isSolved (ag .$ 3))
then let var  = lexeme_ConsLetHO_1 ag
         res  = IsSolved (calculate (ag .$ 3))
         expr = lexeme_ConsLetHO_3 ag
     in  ConsLetHO var res expr (solveST (ag .$ 4))
else let var      = lexeme_ConsLetHO_1 ag
         solved   = lexeme_ConsLetHO_2 ag
         newST = let newST = solveST (ag .$ 3)
                     expr  = lexeme_NestedDclsHO_2 (ag .$ 3)
                 in  NestedDclsHO newST expr
     in  ConsLetHO var solved newST (solveST (ag .$ 4))
"EmptyHO"       → EmptyHO
"NestedDclsHO" → solveST (ag .$ 1)
```

solveST attribute uses the same idea to solve variables if they are defined as an expression or as a nested block (for the constructors *ConsHO* and *ConsLetHO*, respectively). Recall the structure of part of the abstract tree for a `LET` program:



For the *ConsLetHO* the list has the same structure but instead of an expression it contains a nested block.

solveST works as follows:

1. First check if the variable is not solved but if its expression/nested block is solved (all the variables it uses are solved). This is performed with the line ¬ *isSolved* (*ag* .$ 2) ∧ *isSolved* (*ag* .$ 3).
2. If the condition holds, we can solve the variable, which means we *calculate* (defined below) the value of either the expression or the nested block and update the constructor to *isSolved*.
3. If the condition does not hold, we cannot do anything yet, so we will reconstruct this part of tree exactly as we read it.
   - If we are dealing with a variable defined by a nest block, we will try to see if any nested definitions can be solved, by calling *solveST* in the nested block: *solveST* (*ag* .$ 3)
4. The attribute always ends by going to the next declaration, which corresponds to the fourth child: *solveST* (*ag* .$ 4)

With the attributes *isSolved* and *solveSTRoot* defined, we only have to define an attribute that calculates both the meaning of the program through the symbol

tree and of the expressions that define the value of variables throughout each iteration:

```
calculate :: Zipper RootHO → Int
calculate ag = case (constructor ag) of
  "RootHO"       → calculate (ag .$ 2)
  "NestedDclsHO" → calculate (ag .$ 2)
  "Plus"         → calculate (ag .$ 1) + calculate (ag .$ 2)
  "Divide"       → calculate (ag .$ 1) / calculate (ag .$ 2)
  "Minus"        → calculate (ag .$ 1) − calculate (ag .$ 2)
  "Time"         → calculate (ag .$ 1) ∗ calculate (ag .$ 2)
  "Variable"     → getVarValue (lexeme_Variable_1 ag) ag
  "Constant"     → lexeme_Constant_1 ag
```

With these attributes defined, we are now in position to use the generic *fixed_point* function and solve the symbol table. Please recall that this function takes four arguments: our AG in the form of a zipper, a function that checks for termination, a function that is applied whenever the fixed point is reached, and a function that performs one iteration.

In our case, we use *fixed_point* as follows to successfully resolves the symbol table and provides a meaning for a valid `LET` program.

```
solve :: Zipper Root → Int
solve ag = let ho_st = toZipper (createSTRoot ag)
           in  fixed_point ho_st isSolved calculate solveSTRoot
```

As well as illustrating how circular computations can be defined to iterate over a structure, this example also shows that circularity can easily be combined with other AG extensions, in this case higher-order attributes as used for the *ho_st* value.

7 Bidirectional Transformations

Bidirectional transformations are programs which express a transformation from one input to an output together with the reverse transformation, carrying any changes or modifications to the output, in a single specification.

In the context of grammars, a bidirectional transformation represents a transformation from a phrase in one grammar to a phrase in the other, with the opposite direction automatically derived from the first transformation specification.

AGs, and their modern extensions, only provide support for specifying unidirectional transformations, despite bidirectional transformations being common in AG applications, especially between abstract/concrete syntax. For example, when reporting errors discovered on the abstract syntax we want error messages to refer to the original program's concrete syntax, not a possible de-sugared version of it. Or when refactoring source code, programmers should be able to evolve the refactored code, and have the change propagated back to the original source code.

Another application is in semantic editors generated by AGs [28, 22, 29]. Such systems include a manually implemented bidirectional transformation engine to synchronise the abstract tree and its pretty printed representation displayed to users. This is a complex and specific bidirectional transformation that is implemented as two hand-written unidirectional transformations that must be manually synchronized when one of the transformations changes. This makes maintenance complex and error prone. For example, in a transformation $A \to B$, a bidirectionalization system defines the $B \to A$ transformation, which has to carry any upgrades applied to B back to a new A' which is as close as possible to the original A.

In a previous paper [30] we describe a system for generating attribute grammar implementations of bidirectional transformations given only a specification of the forward transformation. This approach is applied here to the embedding of AGs using zippers. Here we sketch the structure of the generated bidirectional transformation while the full details can be found in the earlier paper.

Returning to our running example of the `LET` language in the previous sections, we have worked with its abstract syntax representation (AST) since the abstract syntax is easier to handle and to reason about. However, a concrete syntax representation (CST) is as important. If we want to construct a parser for `LET`, and if we want to provide the programmer with a nice syntax for the language, we will inevitably need a concrete syntax representation, which is what we present next in the form of an `Haskell` data type:

```
data RootC = RootC LetC
data LetC = LetC DclsC InC
data DclsC = ConsLetC      String LetC DclsC
           | ConsAssignC String E    DclsC
           | EmptyDclsC
data E = Add E T
       | Sub E T
       | Et  T
data T = Mul T F
       | Div T F
       | Tf  F
data F = Nest  E
       | Neg   F
       | Var   String
       | Const Int
```

This representation is more complex than the one we have presented in Section 2 (and extended with nested blocks in Section 3), because it has more non-terminal symbols and more productions.

Nonterminals `RootC`, `LetC` and `DclsC` have a single corresponding nonterminal in the abstract representation, `Root`, `Let` and `Dcls` respectively. The same is true for their constructors/productions:

```
RootC       → Root
LetC        → Let
ConsLetC    → ConsLet
ConsAssignC → ConsAssign
EmptyDclsC  → EmptyDcls
```

Since these mappings represent a bijective relation between these constructors, it is very easy have the backward transformation represented just by the inversion of these mappings:

```
RootC       ← Root
LetC        ← Let
ConsLetC    ← ConsLet
ConsAssignC ← ConsAssign
EmptyDclsC  ← EmptyDcls
```

The expressions, on the other hand, are not so simple. In this concrete representation we have three data types for expressions, E, T and F, whereas we have only one in the abstract, *Expr*. An example of the possible mappings between concrete and abstract types, with the former on the left side, is presented next[8]:

```
Add   → Plus
Sub   → Minus
Et    → -
Mul   → Times
Div   → Divide
Tf    → -
Var   → Variable
Const → Constant
```

The constructors `Et` and `Tf` do not have corresponding constructors in the abstract syntax. However, deriving the backward transformation from these mappings presents new challenges. Some decision must be made to determine if an *Expr* on the abstract side is mapped to an E, T or F and this decision should be made for each node in the AST. The simple, naive solution is to map every *Expr* back to F and wrap everything in parenthesis, but this is far from ideal as it unnecessarily produces a complicated concrete representation.

Another problem in defining a bidirectional system appears in cases as the ones presented by the production *Neg*. This production is transformed according to the mapping:

```
Neg (r) → Minus (Constant(0),r)
```

where `r` represents the only child of `Neg`, which is carried out to a subtraction in the abstract view. However, we want to map it back to a negation on the CST, particularly if a negation was there in the first place (i.e., the user didn't right `0-1` on the abstract tree on purpose).

[8] The production *Neg* creates additional dificulties, therefore it is ommitted on purpose and will be discussed below.

The differences between the concrete and abstract representations of `LET` add difficulties when writing the transformations. In previous works [30] we have developed an automatic bidirectionalization system that can use two context free grammars, for the target and for the source, and a representation of a forward transformation and automatically derive AGs that implement such transformations. This system is capable of generating these transformations as AGs, making use of the powerful features of the AG system Silver [9].

Despite our previous work generating code as an AG Domain Specific Language (DSL) in Silver, our embedding provides sufficient expressiveness to support such transformations, as we shall show next.

When applying the backward transformation to a modified tree, it is helpful to have access to the original tree to which the forward transformation was applied so that, at least, the unmodified parts map back to their original representation. We begin by presenting the following data type:

$$\begin{aligned}\textbf{data}\ \mathit{Link} = {} & \mathit{IsRootC}\ \mathit{RootC} \mid \mathit{IsLetC}\ \mathit{LetC} \mid \mathit{IsInC}\ \mathit{InC} \\ & \mid \mathit{IsDclsC}\ \mathit{DclsC} \mid \mathit{IsE}\ E \mid \mathit{IsT}\ T \mid \mathit{IsF}\ F \mid \mathit{Empty}\end{aligned}$$

which represents a link to the original node in the CST for which the AST node was created. All the constructors of the abstract representation are upgraded to have this link as their last child. This process changes the abstract data type, but maintains all the AGs we have seen in the previous sections semantically valid. Recall that in the embedding presented in this paper we call attributes by their ordering number, which means that adding more children to the end of a tree node does not change the order of the existing ones.

In our setting, the transformations are represented by a set of synthesized attributes *get* that is named *getFrom_To*, with *From* representing the type that is being mapped to *To*. Next, we present an example of an attribute that implements the mapping from *RootC* to *Root*:

$$\begin{aligned}&\mathit{getRootC\_Root} :: \mathit{Zipper}\ \mathit{RootC} \to \mathit{Root} \\ &\mathit{getRootC\_Root}\ \mathit{ag} = \textbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \textbf{of} \\ &\quad \texttt{"RootC"} \to \mathit{Root}\ (\mathit{getLetC\_Let}\ (\mathit{ag}\ .\$\ 1))\ (\mathit{createLink}\ \mathit{ag})\end{aligned}$$

where *createLink* is defined as the function that takes a zipper and creates an instance of *Link*. This is the basis of our transformation: go through the concrete tree and create nodes of the AST in an AG-fashion until we have gone through all the nodes in the CST.

When defining the backward transformation we have to be more careful with the problems we have seen previosly: now, abstract nonterminals (*Expr*) can map to more than one in the concrete (E, T or F).

The put attribute (defining the backward transformation) for *Add*, for example, will ask for *putExpr_E* of its left child and *putExpr_T* of its right since these are the correct types for its left and right children, and in our system each *Expr* knows how to translate itself back to any of the E, T, or F non-terminals. By doing so we to avoid naively wrapping every sub-expression in parenthesis, although our transformation still does this if it is required.

Next we present the attribute that transforms parts of an abstract tree whose node is of type *Expr* into nodes of the concrete tree whose type is F:

```
putExpr_F :: Zipper Root → F
putExpr_F ag = case (getLink ag) of
  IsE e   → Nest e
  IsT t   → Nest (Et t)
  IsF f   → f
  Empty → case (constructor ag) of
    "Plus"      → let left  = putExpr_E (ag .$ 1)
                      right = putExpr_T (ag .$ 2)
                  in  Nest (Add left right)
    "Minus"     →
      case (getHole ag :: Maybe Expr) of
        Just (Minus (Constant 0 _) _ _) → Neg (putExpr_F (ag .$ 2))
        otherwise → let left  = putExpr_E (ag .$ 1)
                        right = putExpr_T (ag .$ 2)
                    in  Nest (Sub left right)
    "Times"     → let left  = putExpr_T (ag .$ 1)
                      right = putExpr_F (ag .$ 2)
                  in  Nest (Et (Mul left right))
    "Divide"    → let left  = putExpr_T (ag .$ 1)
                      right = putExpr_F (ag .$ 2)
                  in  Nest (Et (Div left right))
    "Constant" → Const (lexeme_Constant_1 ag)
    "Variable" → Var   (lexeme_Variable_1 ag)
```

There are a couple of important remarks regarding the implementation of *putExpr_F*:

- The first thing the attribute computation does is extract the link back in the node. This is done with the function *getLink*. If this link exists, we can use this information right away, and no other analysis or computations need to be performed. This ensures that the transformation always transforms back to a tree which is as similar as possible to the original one.

  These links back satisfy the invariant that if a node has a link back then all of its children have a link back and there were no transformations on that AST from its original construction from the CST.
- Whenever the types do not match, the system automatically detects if any special constructs can be used. Take for example the line *IsE e → Nest e*. The attribute detects there is a link to something of type E that can be used, but the attribute itself must generate something of type F. Through a completely automatic mechanism described in [30], the system finds that the constructor *Nest* can be used to transform the link into a valid type, and does so.

- If there is no link back (i.e., the link is *Empty*), the attribute will transform it into its equivalent in the concrete representation. *Variable*, for example, is transformed into a *Var*.
- For the constructor *Minus*, the system is capable of detecting that this constructor came either from a *Sub* or from a *Neg*, specializing the transformation whenever possible, i.e., finding if the *Minus* has a zero on the left side, in which case it maps to *Neg*.

The full implementation of the backwards transformation also has the functions *putExpr_E* and *putExpr_T* and their definitions are very similar to *putExpr_F*.

In this setting we create an environment where rewrite rules represent the mappings shown above to specify the forward transformation. From these, we generate the forward and backward AGs implementations, with links back to the original CST. These are used in the transformation but, when not present, we get back a tree without excessive use of parenthesis. In the end we end up with AGs that implement the transformation in both ways and whose semantics are much more complicated that the ones we would usually write by hand.

There are certain conditions on which transformations on the AST create an input where a transformation cannot be applied. Our setting provides mechanisms that detect such trees and advise the user to change the tree so it matches the domain of the transformation. We call these *tree repairs*, and their semantics are further explained in [30].

One last important remark about the bidirectionalization system is that we are generating all these attributes that implement transformations automatically from specific data types for the source, the view and rewrite rules for the forward transformation. This code generation means we can also generate types in `Haskell` directly from the source and view specifications, as well as the functions *constructor* and *lexeme* that we have been using so far, making the the boilerplate code that was until now implemented by the user into an automatic process.

8 Related Work

In this paper, we have proposed a zippers-based embedding of attribute grammars in a functional language. The implementations we obtain are modular and do not rely on laziness. We believe that our approach is the first that deals with arbitrary tree structures while being applicable in both lazy and strict settings. Furthermore, we have been able to implement in our environment all the standard examples that have been proposed in the attribute grammar literature. This is the case of *repmin* [31], *HTML table formatting* [7], and *smart parentesis*, an illustrative example of [9], that are available through the cabal package `ZipperAG`[9].

Moreover, the navigation via a generic zipper that we envison here has applications in other domains: *i*) our setting is being used to create combinator languages for process management [32] which themselves are fundamental to a

[9] `http://hackage.haskell.org/package/ZipperAG`

platform for open source software analysis and certification [33, 34]; and *ii*), the setting that we propose was applied on a prototype for bidirectional transformations applied to programming environments for scientific computing.

Below we survey only works most closely related to ours: works in the realm of functional languages and attribute grammar embeddings.

8.1 Zipper-based approaches

Uustalu and Vene have shown how to embed attribute computations using comonadic structures, where each tree node is paired with its attribute values [35]. This approach is notable for its use of a zipper as in our work. However, it appears that this zipper is not generic and must be instantiated for each tree structure. Laziness is used to avoid static scheduling. Moreover, their example is restricted to a grammar with a single non-terminal and extension to arbitrary grammars is speculative.

Badouel *et al.* define attribute evaluation as zipper transformers [36, 37]. 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. This is also the case of [38], that 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 [39]. Their approach is to translate data structures into a generic representation on which traversals and updates can be performed, then to translate back. Even though their zipper is generic, the implementation is more complex than ours and incurs the extra overhead of translation. It also uses more advanced features of `Haskell` such as type families and rank-2 types.

8.2 Non-zipper-based approaches

Circular programs have been used in the past to straightforwardly implement AGs in a lazy functional language [40, 41]. These works, in contrast to our own, rely on the target language to be lazy, and their goal is not to embed AGs: instead they show that there exists a direct correspondence between an attribute grammar and a circular program.

Regarding other notable embeddings of AGs in functional languages [12–14], they do not offer the modern AG extensions that we provide, with the exception of [14] that uses macros to allow the definition of higher-order attributes. Also, these embeddings are not based on zippers, they rely on laziness and use extensible records [12] or heterogeneous collections [13, 14]. 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 memoization strategies, often costly in terms of performance.

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.* [13, 14]. In this sense, our implementation requires less effort from the programmer.

The Kiama library embeds attribute grammars in Scala and supports extensions such as higher-order attributes and circularity [11]. Kiama's embedding is not purely functional since the host language is not, but it is pure in the sense that it adds no constructs to the Scala language like our Haskell embedding. The role of the zippers in our approach is played by object references in Kiama. In Kiama there is no need to maintain a zipper since a reference to a node is sufficient to identify it, an approach that is not available in a value-based functional language. Kiama uses in-structure references such as "parent" to access the surrounding context of a node, instead of having more traditional inherited attribute definitions.

In [42] the authors introduce "Reference AGs ", and show how these are a special form of circular higher-order AGs, which are available in our setting and are part of the running example on this paper.

8.3 Bidirectional Transformations

Data transformations are an active research topic with multiple strategies applied on various fields, some with a particular emphasis on rule-based approaches. Czarnecki and Helsen [43] present a survey of such techniques, but while they mention bidirectionality, they do not focus on it.

Bidirectional data transformations have been studied in different computing disciplines, such as updatable views in relational databases [44], programmable structure editors [45], model-driven development in software engineering [46], among others. In [47] a detailed discussion and extensive citations on bidirectional transformations are included.

The ATLAS Transformation Language is widely used and has good tool support, but bidirectional transformations must be manually written as a pair of unidirectional transformations [48]. BOTL [49], an object-oriented transformation language, defines a relational approach to transformation of models conforming to metamodels. Despite discussing non bijective transformations, no specification is given regarding how consistency should be restored when there are multiple choices on either direction.

A well-regarded approach to bidirectionalization systems is through lens combinators [44, 50]. These define the semantic foundation and a core programming language, for bidirectional transformations on tree-structured data, but it only works well for surjective (information decreasing) transformations, our system can cope with rather heterogeneous source and target data types.

The approach followed in [51] uses a language for specifying transformations very similar to the one presented in this work, with automatic derivation of

the backward transformation. Similar to our approach, this system statically checks whether changes in views are valid without performing the backward transformation, but they do not provide type-solving techniques such as the one available on our setting, where decisions between mapping different sets of non terminals are completely automated.

In the context of attribute grammars, Yellin's early work on bidirectional transformations in AGs defined attribute grammar inversion [52]. In attribute grammars inversion, an inverse attribute grammar computes an input merely from an output, but in our bidirectional definition of attribute grammars, a backward transformation can use links to the original source to perform better transformations. Thus, our approach can produce more realistic source trees after a change to the target.

9 Conclusion

In this paper we have presented an embedding of modern AG extensions using a concise and elegant zipper-based implementation. We have shown how reference attributes, higher-order attributes and circular attributes can be expressed as first class values in this setting. As a result, complex multiple traversal algorithms can be expressed using an off-the-shelf set of reusable components.

In the particular case of circular attributes, we have presented a generalized fixed_point computation that provides the programmer with easy, AG-based implemetations of complex circular attribute definitions.

We have presented our embedding in the `Haskell` programming language, despite not relying on any advanced feature of `Haskell` such as lazy evaluation. Thus, similar concise embeddings could be defined in other functional languages.

As we have shown both by the examples presented and by the ones available online, our simple embedding provides the same expressiveness of modern, large and more complex attribute grammar based systems.

We have also shown how rewrite rules can be used to specify forward transformations, and be automatically inverted to specify backward transformations, and then be implemented in ou zipper-based embedding of attribute grammars with enforced quality on the transformation.

The features our bidirectionalization system supports are completely automatic for many applications, freeing the programmer of having to write complex attribute equations that have to perform multiple pattern matching, manage both the links back and their types, prioritizing transformations, etc. As far as we are aware, this is the first integration of a bidirectional transformation system in a pure embedded AG framework.

10 Future Work

As part of our future research for our embedding, we plan to:

1. Improve attribute definition by referencing non-terminals instead of (numeric) positions on the right-hand side of productions.

2. Wherever possible, benchmark our embedding against other AG embeddings and systems.
3. We would like to evaluate this embedding together with all the extensions presented on a number of mainstream syntactically rich languages.

References

1. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145 Corrections in **5**(1971) pp. 95–96.
2. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the utrecht haskell compiler. In Weirich, S., ed.: Haskell, ACM (2009) 93–104
3. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In: Advanced Functional Programming. Number 1608 in LNCS, Springer-Verlag (1999) 150–206
4. Fernandes, J.P., Saraiva, J.: 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) 102–111
5. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In Visser, E., Järvi, J., eds.: GPCE, ACM (2010) 43–52
6. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. SIGPLAN Not. **24**(7) (June 1989) 131–145
7. Saraiva, J., Swierstra, S.D.: Generating spreadsheet-like tools from strong attribute grammars. In Pfenning, F., Smaragdakis, Y., eds.: GPCE. Volume 2830 of LNCS., Springer (2003) 307–323
8. Magnusson, E., Hedin, G.: Circular reference attributed grammars - their evaluation and applications. Sci. Comput. Program. **68**(1) (2007) 21–37
9. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming **75**(1–2) (January 2010) 39–54
10. Ekman, T., Hedin, G.: The jastadd extensible java compiler. SIGPLAN Not. **42**(10) (October 2007) 1–18
11. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure embedding of attribute grammars. Science of Computer Programming **78** (2013) 1752–1769
12. de Moor, O., Backhouse, K., Swierstra, S.D.: First-class attribute grammars. Informatica (Slovenia) **24**(3) (2000)
13. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: Procs. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'09). (2009) 245–256
14. Viera, M.: First Class Syntax, Semantics, and Their Composition. PhD thesis, Utrecht University, The Netherlands (2013)
15. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
16. Peyton Jones, S., Hughes, J., Augustsson, L., et al.: Report on the programming language Haskell 98. Technical report (February 1999)
17. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
18. Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Surv. **27**(2) (June 1995) 196–255
19. Bird, R.: Introduction to Functional Programming using Haskell. 2 edn. Prentice Hall PTR (May 1998)

20. Adams, M.D.: 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, ACM (2010) 13–24
21. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Procs. of the 2003 ACM SIGPLAN Inter. WorkShop on Types in Language Design and Implementation. (TLDI '03), ACM (2003) 26–37
22. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In: Procs. of Compiler Construction (CC). Number 1383 in LNCS, Springer-Verlag (1998) 298–301
23. Magnusson, E., Hedin, G.: Circular reference attributed grammars: Their evaluation and applications. Sci. Comput. Program. **68**(1) (August 2007) 21–37
24. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands (December 1999)
25. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. SIGPLAN Not. **21**(7) (July 1986) 85–98
26. Jones, L.G.: Efficient evaluation of circular attribute grammars. ACM Trans. Program. Lang. Syst. **12**(3) (July 1990) 429–462
27. Sasaki, A., Sassa, M.: Circular attribute grammars with remote attribute references and their evaluators. New Generation Computing **22**(1) (2004) 37–60
28. Reps, T., Teitelbaum, T.: The Synthesizer Generator. Springer-Verlag (1989)
29. Söderberg, E.: Contributions to the Construction of Extensible Semantic Editors. PhD thesis, Lund University, Sweden (2012)
30. Martins, P., Saraiva, J.a., Fernandes, J.a.P., Van Wyk, E.: Generating attribute grammar-based bidirectional transformations from rewrite rules. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. PEPM '14, New York, NY, USA, ACM (2014) 63–70
31. Bird, R.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
32. Martins, P., Fernandes, J.P., Saraiva, J.: A purely functional combinator language for software quality assessment. In: Symposium on Languages, Applications and Technologies (SLATE '12). Volume 21 of OASICS., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012) 51–69
33. Martins, P., Fernandes, J.P., Saraiva, J.: A Web Portal for the Certification of Open Source Software. In Cerone, A., Persico, D., Fernandes, S., Garcia-Perez, A., Katsaros, P., Shaikh, S.A., Stamelos, I., eds.: SEFM Satellite Events. Volume 7991 of Lecture Notes in Computer Science., Springer (2012) 244–260
34. Martins, P., Carvalho, N., Fernandes, J.P., Almeida, J.J., Saraiva, J.: A framework for modular and customizable software analysis. In: 13th Int. Conf. on Computational Science and Its Applications (ICCSA 2013). LNCS (7972) (2012) 443–458
35. Uustalu, T., Vene, V.: Comonadic functional attribute evaluation. Trends in Functional Programming, Intellect Books (10) (2005) 145–162
36. Badouel, E., Fotsing, B., Tchougong, R.: Yet another implementation of attribute evaluation. Research Report RR-6315, INRIA (2007)
37. Badouel, E., Tchougong, R., Nkuimi-Jugnia, C., Fotsing, B.: Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition. Theoretical Computer Science **480**(0) (2013) 1 – 25
38. Badouel, E., Fotsing, B., Tchougong, R.: Attribute grammars as recursion schemes over cyclic representations of zippers. Electronic Notes Theory Computer Science **229**(5) (2011) 39–56

39. Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Procs. of the 14th ACM SIGPLAN International Conference on Functional programming. (2009) 233–244
40. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Functional Programming Languages and Computer Architecture. (1987)
41. Kuiper, M., Swierstra, D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands. (November 1987)
42. Sderberg, E., Hedin, G.: Circular higher-order reference attribute grammars. In Erwig, M., Paige, R., Wyk, E., eds.: Software Language Engineering. Volume 8225 of Lecture Notes in Computer Science. Springer International Publishing (2013) 302–321
43. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–645
44. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: A language for updatable views. In: Procs. of ACM Principles of Database Systems (PODS), ACM (2006) 338–347
45. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Procs. of Partial Evaluation and Program Manipulation (PEPM), ACM (2004) 178–189
46. Stevens, P.: A landscape of bidirectional model transformations. In: Generative and Transformational Techniques in Software Engineering II. Number 5235 in LNCS. Springer-Verlag (2008) 408–424
47. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: Procs. of Theory and Practice of Model Transformations (ICMT). Number 5563 in LNCS, Springer-Verlag (2009) 260–283
48. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Procs. of Satellite Events at the MoDELS. Number 3844 in LNCS, Springer-Verlag (2006) 128–138
49. Hibberd, M., Lawley, M., Raymond, K.: Forensic debugging of model transformations. In: Procs. of Model Driven Engineering Languages and Systems. Number 4735 in LNCS, Springer-Verlag (2007) 589–604
50. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems **29**(3) (2007) 17
51. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Procs. of ACM SIGPLAN International Conference on Functional Programming (ICFP), ACM (2007) 47–58
52. Yellin, D.M.: Attribute Grammar Inversion and Source-to-source Translation. Number 302 in LNCS. Springer-Verlag (1988)