Science of Computer Programming 132 (2016) 2–28

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Embedding attribute grammars and their extensions using functional zippers

Pedro Martins [a,1], João Paulo Fernandes [a,b], João Saraiva [a], Eric Van Wyk [c,2], Anthony Sloane [d]

[a] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Braga, Portugal
[b] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Covilhã, Portugal
[c] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA
[d] Department of Computing, Macquarie University, Sydney, Australia

ARTICLE INFO

Article history:
Received 17 April 2014
Received in revised form 20 February 2016
Accepted 15 March 2016
Available online 31 March 2016

Keywords:
Attribute grammars
Functional programming
Functional zippers
Bidirectional transformations

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 reference, higher-order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex computations. This paper studies 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.

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 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 [4,5] and powerful type systems [6].

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.

E-mail addresses: prmartins@di.uminho.pt (P. Martins), jpf@di.ubi.pt (J.P. Fernandes), jas@di.uminho.pt (J. Saraiva), evw@cs.umn.edu (E. Van Wyk), anthony.sloane@mq.edu.au (A. Sloane).

[1] This author is supported 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 – Foundations, Applications and Tools for Bidirectional Transformation (Ref. FCOMP-01-0124-FEDER-020532).

[2] This author is partially supported by NSF Award #1047961.

http://dx.doi.org/10.1016/j.scico.2016.03.005
0167-6423/

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, 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 [7–11]. 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, analyze 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.

Secondly, AG-based systems have extended the standard AG formalism which improves the expressiveness of AGs. Higher-order AGs (HOAGs) [12,13] provide a modular extension to AGs in which abstract trees can be stored as attribute values. Reference AGs (RAGs) [14,15] 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 [16,17], JastAdd [18], and Kiama [11] all support such extensions.

However, and even considering their modern extensions, attribute grammars 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, an embedding which is also powerful enough to express bidirectional transformations. Indeed, we revise the zipper-based AG embedding proposed in [9] to extend it with the bidirectional capabilities of [19]. We have used this embedding 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.

In the remainder of the paper, we start by revising the concise embedding of AGs in `Haskell` of [9]. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [20] 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.

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

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` [21] or `ML` [22].

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.[3]

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

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 and that a variable is not declared more than once;
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 arithmetic expressions that compose a program.

Since LET does not enforce a *declare-before-use* discipline, a straightforward definition of the scope analysis relies on two traversals over the abstract tree: first, to collect the declarations of variables, while at the same time searching for multiple declarations of 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 declarations 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 the zipper-based AG embedding techniques of [9] together with the extensions we provide. In particular, Sections 3, 4 and 5 focus on challenge 1) above while Section 6 focuses on challenge 2).

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):

```
(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 starting 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: they can be composed of a variable name (Name), an expression (Expr) and another declaration (Dcls) or they can represent an empty list.

AGs themselves consist of an extension to CFGs, in the sense that they use CFGs to define the syntax of a language, 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 would hold a list

[3] 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.

containing a and c, because these are the variables declared before b. The attribute `dclo` (declarations out) gives the declared variables including the current node. 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 complete 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 type of `errs` is a list of identifiers.

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

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

the AG that performs the name analysis will yield as result a list containing the variables a and z, in this order. The former because a is being declared twice and the latter because z 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 [23], 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).[4]

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:

- l_1 `++` l_2 concatenates lists l_1 and l_2
- `[]` represents an empty list
- `h : l` adds element `h` to the head of list `l`
- `mBIn x l` (read `x mustBeIn l`) returns the singleton list `[x]` in case `x` is not an element of `l`, and `[]` otherwise
- `mNBIn x l` (read `x mustNotBeIn l`), returns the singleton list `[x]` in case `x` is an element of `l`, and `[]` 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 a sentence. Such a function returns the accumulated list of declaration as its final result. This technique is known as accumulating parameters [24]. 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`, which both hold lists of variable names, that we present next.

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

```
(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.

[4] 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.

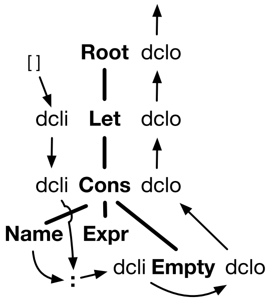

Fig. 1. The relationship between the inherited attribute `dcli` and the synthesized attribute `dclo`, implementing an accumulation pattern.

The attribute `dclo` works bottom-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. The implementation of `dclo` 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 specific semantics often depend only on specific parts of the tree/language. The full pattern of attribute calculation can be seen for a simple tree in Fig. 1.

2.2. Distributing variable declarations

One important part of the semantics of analyzing 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:

```
(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 = Dcls₁.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 relationship with `dclo` can be seen in Fig. 2. We use `A` to denote an `Expr` production such as `Plus` or `Minus`.

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.

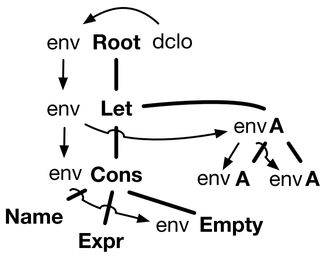

Fig. 2. The inherited attribute `env`, distributing the environment throughout the tree.

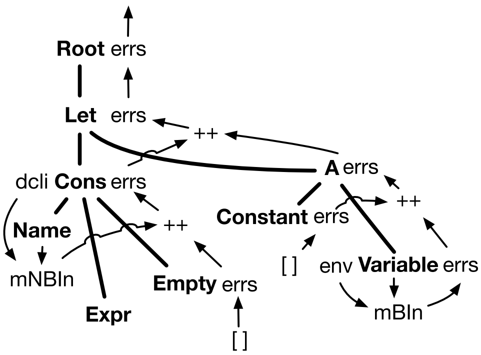

Fig. 3. The synthesized attribute `errs`.

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
      { 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 Fig. 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`.

Summarizing the AG formalism, attribute occurrences are calculated by invocations of small semantic functions that depend on the values of other attribute 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 [20] 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 languages 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 [25]. This library works for both homogeneous and heterogeneous data types. The library can traverse any data type that has an instance of the *Data* and *Typeable* type classes [26].

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 3))
         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.

On top of the zipper library we have implemented several simple abstractions that facilitate the embedding of attribute grammars. In particular, we have defined:

- $(.\$) :: Zipper\ a \to Int \to Zipper\ a$, for accessing any child of a structure given by its index starting at 1;
- $parent :: Zipper\ a \to Zipper\ a$, to move the focus to the parent of a concrete node;
- $(.|) :: Zipper\ a \to Int \to Bool$, to check whether the current location is a sibling of a tree node;
- $constructor :: Zipper\ a \to String$, 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.[5] For this, we may use the following algebraic expression which also 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,

$$zipper_{expr} = toZipper\ expr$$

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

$$constructor\ zipper_{expr} \equiv \text{“}Times\text{”}$$

move the focus from *expr* to its first child and check the constructor under focus afterwards,

$$child_1 = zipper_{expr}\ .\$\ 1$$
$$constructor\ child_1 \equiv \text{“}Plus\text{”}$$

and do the same with the parent,

$$constructor\ (parent\ child_1) \equiv \text{“}Times\text{”}$$

Finally, we can define functions such as $lexeme_{Constant_1} :: Zipper\ a \to Int$, where

$$lexeme_{Constant_1}\ (child_1\ .\$\ 2) \equiv 7$$

extracts information from the zipper. Throughout this paper, we will use lexeme functions to access child nodes at known zipper locations, not just those of terminal constructs. The name of these lexeme functions will always have the form $lexeme_{Constructor_i}$, where *Constructor* corresponds to the current data constructor and *i* corresponds to the number of the child 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

Having introduced the zipper data structure and illustrated its use in practice, we now show how that the AG presented in Section 2 can be implemented in the functional language `Haskell`.

We start by analyzing the implementation of attribute *dcli*. This is an inherited attribute that makes a top–down traversal over the tree, collecting declarations. We can define it in `Haskell` as:

```
dcli :: Zipper Root → [String]
dcli ag = case (constructor ag) of
  “Root” → [ ]
  _      → case (constructor (parent ag)) of
             “Cons” → lexeme_Cons1 (parent ag) : dcli (parent ag)
             _      → dcli (parent ag)
```

The value of *dcli* on the topmost node of a tree, *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 _ extends the declaration of *dcli* for all the tree nodes, whereas in the AG defined in Section 2 it is only declared for a few productions. This artificial extension of the semantics of *dcli* is not a problem, as this attribute will only be computed in contexts where other attributes are dependent upon it (i.e., where other attributes call *dcli*). This has the same semantic meaning as individually declaring *dcli* only for the tree nodes where it needs to be computed, but the usage of _ simplifies the implementation (and in some AG avoids the repetition of code for attributes which compute the same semantic actions in different contexts).

[5] Recall that we are using a generic zipper library, so no additional coding is necessary to accommodate a particular structure.

Next, we present the implementation of attribute *dclo*:

$$\begin{array}{l} dclo :: Zipper\ Root \rightarrow [String] \\ dclo\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\ \quad \text{“}Root\text{”} \quad \rightarrow dclo\ (ag\ .\$\ 1) \\ \quad \text{“}Let\text{”} \quad\ \ \rightarrow dclo\ (ag\ .\$\ 1) \\ \quad \text{“}Cons\text{”} \quad \rightarrow dclo\ (ag\ .\$\ 3) \\ \quad \text{“}Empty\text{”} \rightarrow dcli\ ag \end{array}$$

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`:

$$\begin{array}{l} env :: Zipper\ Root \rightarrow [String] \\ env\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\ \quad \text{“}Root\text{”} \rightarrow dclo\ ag \\ \quad \_ \qquad\ \ \rightarrow env\ (parent\ ag) \end{array}$$

where we define the attribute for the topmost 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 [16] and references to remote attributes in LRC [27]). In this embedding, we can elegantly implement this feature using standard primitives from the hosting language.

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

$$\begin{array}{l} errs :: Zipper\ Root \rightarrow [String] \\ errs\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\ \quad \text{“}Root\text{”} \qquad \rightarrow errs\ (ag\ .\$\ 1) \\ \quad \text{“}Let\text{”} \qquad\ \ \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\ \quad \text{“}Cons\text{”} \qquad \rightarrow (lexeme_{Cons_1}\ ag)\ \notin\ (dcli\ ag) \\ \qquad\qquad\qquad\quad +\!\!+\ errs\ (ag\ .\$\ 2) +\!\!+\ errs\ (ag\ .\$\ 3) \\ \quad \text{“}Empty\text{”} \quad\ \rightarrow [\,] \\ \quad \text{“}Plus\text{”} \qquad \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\ \quad \text{“}Divide\text{”} \quad \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\ \quad \text{“}Minus\text{”} \quad\ \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\ \quad \text{“}Times\text{”} \quad\ \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\ \quad \text{“}Variable\text{”} \rightarrow (lexeme_{Variable_1}\ ag)\ \in\ (env\ ag) \\ \quad \text{“}Constant\text{”} \rightarrow [\,] \end{array}$$

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

The semantic functions $\in$ and $\notin$ check whether or not a variable belongs to an environment and correspond to `mBIn` and `mNBIn`, respectively, from Section 2. They can easily be defined in `Haskell`:

$$\begin{array}{l} \in ::\ String \rightarrow [String] \rightarrow [String] \\ name\ \in\ [\,] \qquad\quad = [name] \\ name\ \in\ (n : es) = \textbf{if}\ (n \equiv name)\ \textbf{then}\ [\,]\ \textbf{else}\ name\ \in\ es \\ \notin ::\ String \rightarrow [String] \rightarrow [String] \\ a\ \notin\ [\,] = [\,] \\ a_1\ \notin\ (a_2 : es) = \textbf{if}\ (a_1 \equiv a_2)\ \textbf{then}\ [a_1]\ \textbf{else}\ a_1\ \notin\ es \end{array}$$

Recall that $\in$ and $\notin$ signal an error with a variable whose identifier is *id* by returning $[\,id\,]$. This means, for example, that if we expected variable x to be present in an environment *env* but it is not, then $x \in env$ produces $[\,x\,]$.

A difference between our embedding and the traditional definition of AGs 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 pre-defined. Also, regarding distribution of language features for dynamic loading and separate compilation, it is possible to divide an AG in modules that, e.g., may contain data types (representing the grammar) and functions (representing the attributes).

4. References in attribute grammars

In the last section we saw how zipper-based constructs can be used to implement AGs in a functional setting. Here we will show how one of the most widely used AG extensions, that allows us to create and use references to tree nodes, can be embedded as well.

Reference Attribute Grammars (RAGs) were first introduced by Magnusson and Hedin [28]. 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:

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

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*:

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

with *ConsLet* representing nested blocks of code. The remaining syntax tree keeps the exact same definition as presented in Section 3.

While syntactically this change is very simple to make, semantically it 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, variable 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 [29].

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*:

$$
\begin{aligned}
&lev :: Zipper\; a \to Int\\
&lev\; ag = \mathbf{case}\; (constructor\; ag)\; \mathbf{of}
\end{aligned}
$$

$$
\begin{array}{l}
\quad \text{"}Root\text{"} \rightarrow 0 \\
\quad \text{"}Let\text{"} \quad \rightarrow \textbf{case}\ (constructor\ (parent\ ag))\ \textbf{of} \\
\qquad\qquad\qquad \text{"}ConsLet\text{"} \rightarrow lev\ (parent\ ag) + 1 \\
\qquad\qquad\qquad \_ \qquad\qquad \rightarrow lev\ (parent\ ag) \\
\quad \_ \qquad\quad \rightarrow lev\ (parent\ ag)
\end{array}
$$

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:

$$
\begin{array}{l}
dcli :: Zipper\ Root \rightarrow [\,(String, Zipper\ Root)\,] \\
dcli\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
\quad \text{"}Root\text{"} \rightarrow [\,] \\
\quad \text{"}Let\text{"} \quad \rightarrow \textbf{case}\ (constructor\ (parent\ ag))\ \textbf{of} \\
\qquad\qquad\qquad \text{"}Root\text{"} \qquad \rightarrow dcli\ (parent\ ag) \\
\qquad\qquad\qquad \text{"}ConsLet\text{"} \rightarrow env\ (parent\ ag) \\
\quad \_ \qquad\quad \rightarrow \textbf{case}\ (constructor\ (parent\ ag))\ \textbf{of} \\
\qquad\qquad\qquad \text{"}Cons\text{"} \qquad \rightarrow (lexeme_{Cons_1}\ (parent\ ag), parent\ ag) \\
\qquad\qquad\qquad\qquad\qquad\qquad : dcli\ (parent\ ag) \\
\qquad\qquad\qquad \text{"}ConsLet\text{"} \rightarrow (lexeme_{ConsLet_1}\ (parent\ ag), parent\ ag) \\
\qquad\qquad\qquad\qquad\qquad\qquad : dcli\ (parent\ ag) \\
\qquad\qquad\qquad \text{"}Empty\text{"} \quad \rightarrow dcli\ (parent\ ag)
\end{array}
$$

The semantics are very similar to the previous version with two big differences: first, the return type of *dcli* is now $[\,(String, Zipper\ Root)\,]$[6] 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 $\notin$ as:

$$
\begin{array}{l}
\notin :: (String, Zipper\ Root) \rightarrow [\,(String, Zipper\ Root)\,] \rightarrow [String] \\
tuple\ \notin\ [\,] \qquad\qquad\qquad = [\,] \\
(a_1, r_1)\ \notin\ ((a_2, r_2) : es) = \textbf{if}\ (a_1 \equiv a_2) \wedge (lev\ r_1 \equiv lev\ r_2) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ [\,a_1\,] \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (a_1, r_1)\ \notin\ es
\end{array}
$$

Now $\notin$ checks if the variable name and 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:

$$
\begin{array}{l}
errs :: Zipper\ Root \rightarrow [String] \\
errs\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
\quad \text{"}Root\text{"} \qquad \rightarrow errs\ (ag\ .\$\ 1) \\
\quad \text{"}Let\text{"} \qquad\ \rightarrow errs\ (ag\ .\$\ 1) +\!\!+\ errs\ (ag\ .\$\ 2) \\
\quad \text{"}Cons\text{"} \qquad \rightarrow (lexeme_{Cons_1}\ ag, ag)\ \notin\ (dcli\ ag)
\end{array}
$$

[6] It could also have been defined as $[\,(String, Int)\,]$ by computing the level of each variable declaring node, but this would affect the extensibility properties that we refer to later.

$$
\begin{array}{ll}
 & \quad +\!\!+ \; errs\ (ag\ .\$\ 2) +\!\!+ errs\ (ag\ .\$\ 3) \\
\text{“ConsLet”} & \rightarrow (lexeme_{ConsLet_1}\ ag, ag) \notin (dcli\ ag) \\
 & \quad +\!\!+ \; errs\ (ag\ .\$\ 2) +\!\!+ errs\ (ag\ .\$\ 3) \\
\text{“Empty”} & \rightarrow [\,] \\
\text{“Plus”} & \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
\text{“Divide”} & \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
\text{“Minus”} & \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
\text{“Times”} & \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
\text{“Variable”} & \rightarrow (lexeme_{Variable_1}\ ag) \in (env\ ag) \\
\text{“Constant”} & \rightarrow [\,]
\end{array}
$$

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 referenced site. This capability together with attributes being first-class citizens in the host 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 that allows us to use higher-order attributes is implemented in our setting.

5. Higher-order attribute grammars

Higher-Order Attribute Grammars (HOAGs) were first introduced by Vogt et al. as a setting where the original tree structure used in an AG can be expanded and later decorated as the result of attribute computations [12]. This means that it is possible to associate semantics to the new parts of the tree.

HOAGs are commonly used for program transformations or translations to other languages. Consider a transformation of LET expressions that lifts nested declarations to the top level and renames variables as required in the process. Part of this transformation could be implemented using higher order synthesized attributes that construct a new syntax tree of LET expressions. Attributes on this new tree can also be evaluated. For example, we could translate this lifted expression to C using additional higher order attributes that construct C language syntax trees. Finally, a string *unparse* attribute on the C trees could be evaluated to compute a string representation of the C syntax tree.

Compared to traditional AGs, HOAGs provide a setting where:

- attributes define new trees whose semantics is defined as a new set of attribute 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 analyzing 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 a HOAG where added semantics will be defined as attributes.

Since we have already defined and implemented the scope rules for LET, it is simpler to define and implement the semantics of the symbol table (and to solve 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 safely search for a variable being used in an expression with the guarantee that it has been declared and so we will appropriately find it.

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

$$
\begin{array}{l}
\mathbf{data}\ Root_{HO} = Root_{HO}\ Dcls_{HO}\ Expr \\
\mathbf{data}\ Dcls_{HO} = Cons_{HO}\quad String\ IsSolved\ Expr\ Dcls_{HO} \\
\qquad\qquad\quad\ \mid\ ConsLet_{HO}\ String\ IsSolved\ Nested_{HO}\ Dcls_{HO} \\
\qquad\qquad\quad\ \mid\ Empty_{HO} \\
\mathbf{data}\ Nested_{HO} = NestedDcls_{HO}\ Dcls_{HO}\ Expr \\
\mathbf{data}\ IsSolved = IsSolved\ Int \mid NotSolved
\end{array}
$$

These four data types have the following functionality:

- $Root_{HO}$ contains the list of declarations and the final expression to be solved.
- $Dcls_{HO}$ has two data constructors, $Cons_{HO}$ and $ConsLet_{HO}$, for variable declarations and nested blocks respectively. These constructors both carry the variable name as a *String*, and both recursively define $Dcls_{HO}$. However, whereas the former has an expression, the latter carries nested information.
- $NestedDcls_{HO}$ 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 $Dcls_{HO}$, 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 → Root_HO
createSTRoot ag = case (constructor ag) of
  "Root" → Root_HO (createST ag) (lexeme_Let2 (ag .$ 1))
```

Here, the first argument of $Root_{HO}$ is the attribute that creates the symbol table and $lexeme_{Let_2}\ (ag\ .\$\ 1)$ accesses the expression that constitutes the meaning of the underlying LET program. Please recall that the abstract tree for LET has the form:

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

and therefore to access the top level expression we have to go to the first child of *Root*, a *Let*, and then get the expression in its 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 → Dcls_HO
createST ag = case (constructor ag) of
  "Root"    → createST (ag .$ 1)
  "Let"     → createST (ag .$ 1)
  "Cons"    → let var  = lexeme_Cons1 ag
                  expr = lexeme_Cons2 ag
              in Cons_HO var NotSolved expr (createST (ag .$ 3))
  "ConsLet" → let var = lexeme_ConsLet1 ag
                  nested = let nested = createST (ag .$ 2)
                               expr   = lexeme_Let2 (ag .$ 2)
                           in NestedDcls_HO nested expr
              in ConsLet_HO var NotSolved nested (createST (ag .$ 3))
  "Empty"   → Empty_HO
```

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.

Having defined *createSTRoot* and *createST*, we can now create a new tree on which new 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 Root_HO → Bool
isVarSolved name ag = case (constructor ag) of
  "Root_HO"       → isVarSolved_aux name ag
  "NestedDcls_HO" → isVarSolved_aux name ag
  _               → isVarSolved     name (parent ag)
```

Attribute *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 ($Root_{HO}$) or to the root of the nearest nested

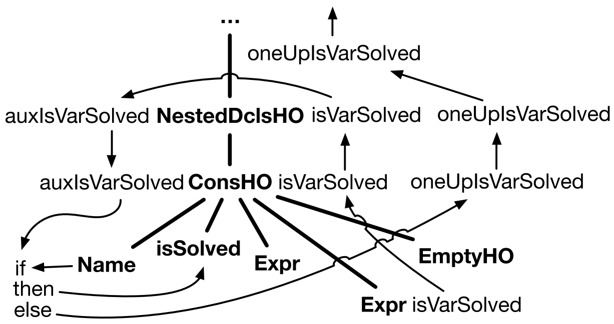

Fig. 4. Dependency between *isVarSolved*, *isVarSolved$_{aux}$* and *oneUpIsVarSolved*.

block (*NestedDcls$_{HO}$*). We do so to ensure that when the *isVarSolved$_{aux}$* attribute is called we are searching in the whole block, starting in its topmost position:

```
isVarSolved_aux :: String → Zipper Root_HO → Bool
isVarSolved_aux name ag = case (constructor ag) of
  "Root_HO"        → isVarSolved_aux name (ag .$ 1)
  "NestedDcls_HO"  → isVarSolved_aux name (ag .$ 1)
  "Cons_HO"        → if   lexeme_Cons_HO_1 ag ≡ name
                     then isVarSolved_aux name (ag .$ 2)
                     else isVarSolved_aux name (ag .$ 4)
  "ConsLet_HO"     → if   lexeme_ConsLet_HO_1 ag ≡ name
                     then isVarSolved_aux name (ag .$ 2)
                     else isVarSolved_aux name (ag .$ 4)
  "IsSolved"       → True
  "NotSolved"      → False
  "Empty_HO"       → oneUpIsVarSolved name ag
```

The synthesized *isVarSolved$_{aux}$* 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 *Empty$_{HO}$* and the *oneUpIsVarSolved* attribute is called:

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

The behavior 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 *Root$_{HO}$*, because we do not 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*, *isVarSolved$_{aux}$* and *oneUpIsVarSolved* is their interdependence. The first two, *isVarSolved* and *isVarSolved$_{aux}$*, 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 Fig. 4.

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 Root_HO → Int
getVarValue name ag = case (constructor ag) of
  "Root_HO"       → getVarValue_aux name ag
  "NestedDcls_HO" → getVarValue_aux name ag
  _               → getVarValue     name (parent ag)
```

$$
\begin{array}{l}
getVarValue_{aux} :: String \to Zipper\ Root_{HO} \to Int \\
getVarValue_{aux}\ name\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
\quad \text{“}Root_{HO}\text{”} \quad\quad\ \to getVarValue_{aux}\ name\ (ag\ .\$\ 1) \\
\quad \text{“}NestedDcls_{HO}\text{”} \to getVarValue_{aux}\ name\ (ag\ .\$\ 1) \\
\quad \text{“}Cons_{HO}\text{”} \quad\quad \to \textbf{if} \quad lexeme_{Cons_{HO_1}}\ ag \equiv name \\
\quad\quad\quad\quad\quad\quad\quad\ \ \textbf{then}\ getVarValue_{aux}\ name\ (ag\ .\$\ 2) \\
\quad\quad\quad\quad\quad\quad\quad\ \ \textbf{else}\ \ getVarValue_{aux}\ name\ (ag\ .\$\ 4) \\
\quad \text{“}ConsLet_{HO}\text{”} \quad \to \textbf{if} \quad lexeme_{ConsLet_{HO_1}}\ ag \equiv name \\
\quad\quad\quad\quad\quad\quad\quad\ \ \textbf{then}\ getVarValue_{aux}\ name\ (ag\ .\$\ 2) \\
\quad\quad\quad\quad\quad\quad\quad\ \ \textbf{else}\ \ getVarValue_{aux}\ name\ (ag\ .\$\ 4) \\
\quad \text{“}IsSolved\text{”} \quad\quad \to lexeme_{IsSolved_1}\ ag \\
\quad \text{“}Empty_{HO}\text{”} \quad\quad \to oneUpGetVarValue\ name\ ag
\end{array}
$$

$$
\begin{array}{l}
oneUpGetVarValue :: String \to Zipper\ Root_{HO} \to Int \\
oneUpGetVarValue\ name\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
\quad \text{“}NestedDcls_{HO}\text{”} \to getVarValue \quad\quad\quad name\ (parent\ ag) \\
\quad \_ \quad\quad\quad\quad\quad\quad \to oneUpGetVarValue\ name\ (parent\ ag)
\end{array}
$$

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 cannot 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, i.e., the value it represents.

6. Circular attribute grammars

An Attribute Grammar is called circular (CAG) if it has an attribute that depends on itself, even if transitively. 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 [30–32]. Another example where CAGs are useful is analyzing variable declarations such as the ones in the following LET program.

$$
\begin{array}{l}
program = \textbf{let}\ x = y \\
\quad\quad\quad\quad\quad\quad z = 1 \\
\quad\quad\quad\quad\quad\quad y = z \\
\quad\quad\quad\quad\ \ \textbf{in}\ \ x + y + 1
\end{array}
$$

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 analyzes 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, ⊥, and compute approximations of the final result until it is not changed any more, 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 ⊥ 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:

[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 customization of the fixed point calculation to stop the circular attribute evaluation for cases like this.

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

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 fix_{point} function.

$$
\begin{aligned}
&\mathit{fix}_{point} :: \mathit{Zipper}\ a \to (\mathit{Zipper}\ a \to \mathit{Bool}) \to (\mathit{Zipper}\ a \to b) \\
&\qquad\qquad \to (\mathit{Zipper}\ a \to \mathit{Zipper}\ a) \to b \\
&\mathit{fix}_{point}\ \mathit{ag}\ \mathit{cond}\ \mathit{calc}\ \mathit{incr} = \mathbf{if}\quad \mathit{cond}\ \mathit{ag} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{calc}\ \ \mathit{ag} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathit{fix}_{point}\ (\mathit{incr}\ \mathit{ag})\ \mathit{cond}\ \mathit{calc}\ \mathit{incr}
\end{aligned}
$$

The arguments of this function are as follows:

- $\mathit{ag} :: \mathit{Zipper}\ a$, 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 $\mathit{Zipper}\ \mathit{Root}_{HO}$.
- $\mathit{cond} :: \mathit{Zipper}\ a \to \mathit{Bool}$, 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.
- $\mathit{calc} :: \mathit{Zipper}\ a \to b$, 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.
- $\mathit{incr} :: \mathit{Zipper}\ a \to \mathit{Zipper}\ a$, 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 fix_{point}, starting with the attribute that ends the circular computation by defining the fixed point (*cond*):

$$
\begin{aligned}
&\mathit{isSolved} :: \mathit{Zipper}\ \mathit{Root}_{HO} \to \mathit{Bool} \\
&\mathit{isSolved}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of} \\
&\quad \text{“}\mathit{Root}_{HO}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \vee \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \\
&\quad \text{“}\mathit{NestedDcls}_{HO}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \\
&\quad \text{“}\mathit{Cons}_{HO}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 4) \\
&\quad \text{“}\mathit{ConsLet}_{HO}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 4) \\
&\quad \text{“}\mathit{Empty}_{HO}\text{”} \to \mathit{True} \\
&\quad \text{“}\mathit{IsSolved}\text{”} \to \mathit{True} \\
&\quad \text{“}\mathit{NotSolved}\text{”} \to \mathit{False} \\
&\quad \text{“}\mathit{Plus}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \\
&\quad \text{“}\mathit{Divide}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \\
&\quad \text{“}\mathit{Minus}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \\
&\quad \text{“}\mathit{Times}\text{”} \to \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \\
&\quad \text{“}\mathit{Variable}\text{”} \to \mathit{isVarSolved}\ (\mathit{lexeme}_{\mathit{Variable}_1}\ \mathit{ag})\ \mathit{ag} \\
&\quad \text{“}\mathit{Constant}\text{”} \to \mathit{True}
\end{aligned}
$$

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 (Root_{HO} 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.

$$
\begin{array}{l}
\mathit{solveSTRoot} :: \mathit{Zipper}\ \mathit{Root}_{HO} \rightarrow \mathit{Zipper}\ \mathit{Root}_{HO} \\
\mathit{solveSTRoot}\ ag = \mathbf{let}\ \mathit{solved}_{dcls} = \mathit{solveST}\ (ag\ .\$\ 1) \\
\qquad\qquad\qquad\qquad\quad \mathit{top}_{expr} \quad = \mathit{lexeme}_{Root_{HO_2}}\ ag \\
\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{toZipper}\ (\mathit{Root}_{HO}\ \mathit{solved}_{dcls}\ \mathit{top}_{expr})
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{solveST} :: \mathit{Zipper}\ \mathit{Root}_{HO} \rightarrow \mathit{Dcls}_{HO} \\
\mathit{solveST}\ ag = \mathbf{case}\ (\mathit{constructor}\ ag)\ \mathbf{of} \\
\quad \text{“}\mathit{Cons}_{HO}\text{”} \rightarrow \\
\qquad \mathbf{if} \quad (\neg\ \mathit{isSolved}\ (ag\ .\$\ 2) \wedge \mathit{isSolved}\ (ag\ .\$\ 3)) \\
\qquad \mathbf{then\ let}\ \mathit{var} \ \ = \mathit{lexeme}_{Cons_{HO_1}}\ ag \\
\qquad\qquad\qquad \mathit{res} \ \ = \mathit{isSolved}\ (\mathit{calculate}\ (ag\ .\$\ 3)) \\
\qquad\qquad\qquad \mathit{expr} = \mathit{lexeme}_{Cons_{HO_3}}\ ag \\
\qquad\qquad \mathbf{in}\ \ \mathit{Cons}_{HO}\ \mathit{var}\ \mathit{res}\ \mathit{expr}\ (\mathit{solveST}\ (ag\ .\$\ 4)) \\
\qquad \mathbf{else}\ \ \mathbf{let}\ \mathit{var} \ \ = \mathit{lexeme}_{Cons_{HO_1}}\ ag \\
\qquad\qquad\qquad \mathit{res} \ \ = \mathit{lexeme}_{Cons_{HO_2}}\ ag \\
\qquad\qquad\qquad \mathit{expr} = \mathit{lexeme}_{Cons_{HO_3}}\ ag \\
\qquad\qquad \mathbf{in}\ \ \mathit{Cons}_{HO}\ \mathit{var}\ \mathit{res}\ \mathit{expr}\ (\mathit{solveST}\ (ag\ .\$\ 4) \\
\quad \text{“}\mathit{ConsLet}_{HO}\text{”} \rightarrow \\
\qquad \mathbf{if} \quad (\neg\ \mathit{isSolved}\ (ag\ .\$\ 2) \wedge \mathit{isSolved}\ (ag\ .\$\ 3)) \\
\qquad \mathbf{then\ let}\ \mathit{var} \ \ = \mathit{lexeme}_{ConsLet_{HO_1}}\ ag \\
\qquad\qquad\qquad \mathit{res} \ \ = \mathit{isSolved}\ (\mathit{calculate}\ (ag\ .\$\ 3)) \\
\qquad\qquad\qquad \mathit{expr} = \mathit{lexeme}_{ConsLet_{HO_3}}\ ag \\
\qquad\qquad \mathbf{in}\ \ \mathit{ConsLet}_{HO}\ \mathit{var}\ \mathit{res}\ \mathit{expr}\ (\mathit{solveST}\ (ag\ .\$\ 4)) \\
\qquad \mathbf{else}\ \ \mathbf{let}\ \mathit{var} \quad\ = \mathit{lexeme}_{ConsLet_{HO_1}}\ ag \\
\qquad\qquad\qquad \mathit{solved}\ \ = \mathit{lexeme}_{ConsLet_{HO_2}}\ ag \\
\qquad\qquad\qquad \mathit{newST} = \mathbf{let}\ \mathit{newST} = \mathit{solveST}\ (ag\ .\$\ 3) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{expr} \quad\ = \mathit{lexeme}_{NestedDcls_{HO_2}}\ (ag\ .\$\ 3) \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathit{NestedDcls}_{HO}\ \mathit{newST}\ \mathit{expr} \\
\qquad\qquad \mathbf{in}\ \ \mathit{ConsLet}_{HO}\ \mathit{var}\ \mathit{solved}\ \mathit{newST}\ (\mathit{solveST}\ (ag\ .\$\ 4)) \\
\quad \text{“}\mathit{Empty}_{HO}\text{”} \qquad\ \rightarrow \mathit{Empty}_{HO} \\
\quad \text{“}\mathit{NestedDcls}_{HO}\text{”} \rightarrow \mathit{solveST}\ (ag\ .\$\ 1)
\end{array}
$$

Attribute *solveST* uses the same idea to solve variables if they are defined as an expression or as a nested block (for the constructors $Cons_{HO}$ and $ConsLet_{HO}$, respectively). Recall the structure of part of the abstract tree for a LET program:

```
             ...
              |
           ConsHO
          / | \    \
      var   |   \  ...
           /     \
    iSolved     Expr
```

For the $ConsLet_{HO}$ the list has the same structure but instead of an expression it contains a nested block.

Attribute *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 $\neg\ \mathit{isSolved}\ (ag\ .\$\ 2) \wedge \mathit{isSolved}\ (ag\ .\$\ 3)$.
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 nested 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:

$$
\begin{array}{ll}
\multicolumn{2}{l}{calculate :: Zipper\ Root_{HO} \rightarrow Int} \\
\multicolumn{2}{l}{calculate\ ag = \mathbf{case}\ (constructor\ ag)\ \mathbf{of}} \\
\quad \text{"}Root_{HO}\text{"} & \rightarrow calculate\ (ag\ .\$\ 2) \\
\quad \text{"}NestedDcls_{HO}\text{"} & \rightarrow calculate\ (ag\ .\$\ 2) \\
\quad \text{"}Plus\text{"} & \rightarrow calculate\ (ag\ .\$\ 1) + calculate\ (ag\ .\$\ 2) \\
\quad \text{"}Divide\text{"} & \rightarrow calculate\ (ag\ .\$\ 1)\ /\ calculate\ (ag\ .\$\ 2) \\
\quad \text{"}Minus\text{"} & \rightarrow calculate\ (ag\ .\$\ 1) - calculate\ (ag\ .\$\ 2) \\
\quad \text{"}Times\text{"} & \rightarrow calculate\ (ag\ .\$\ 1) * calculate\ (ag\ .\$\ 2) \\
\quad \text{"}Variable\text{"} & \rightarrow getVarValue\ (lexeme_{Variable_1}\ ag)\ ag \\
\quad \text{"}Constant\text{"} & \rightarrow lexeme_{Constant_1}\ ag
\end{array}
$$

With these attributes defined, we are now in position to use the generic fix_{point} function to 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 fix_{point} as follows to successfully resolve the symbol table and provide a meaning for a valid `LET` program.

$$
\begin{array}{l}
solve :: Zipper\ Root \rightarrow Int \\
solve\ ag = \mathbf{let}\ ho_{st} = toZipper\ (createSTRoot\ ag) \\
\qquad\qquad \mathbf{in}\ \ fix_{point}\ ho_{st}\ isSolved\ calculate\ solveSTRoot
\end{array}
$$

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 [27,33,34]. Such systems include a manually implemented bidirectional transformation engine to synchronize 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 $A \rightarrow B$, a bidirectionalization system defines the $B \rightarrow A$ 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.

7.1. Background

In a previous paper [19] 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 theoretical background of our bidirectional transformation engine, while the full details can be found in the earlier paper.

We start by defining an operator scheme: $\Sigma = \langle S, F, \sigma \rangle$ where S is a set of sorts (sort names), F is a set of operator or function names and σ maps F to $S^* \times S$. For grammars, sorts correspond to nonterminals and terminals, operators correspond to production names, and signatures in σ correspond to productions. Constants are treated as nullary operators. A Σ-algebra: A_σ is defined as:

Source language: $\Sigma^E = \langle S^E, F^E, \sigma^E \rangle$ where:

- $S^E = \{E, T, F, \mathit{digits}, `+', `-', `*', `/', `(', `)', \mathit{String}\}$
- $F^E = \{\mathit{add}, \mathit{sub}, \mathit{et}, \mathit{mul}, \mathit{div}, \mathit{tf}, \mathit{nest}, \mathit{const}, \mathit{digits}, \mathit{neg}, `+', `-', `*', `/', `(', `)', \mathit{String}\}$
- $\sigma^E(\mathit{add}) = E\ `+'\ T \to E$,
  $\sigma^E(\mathit{sub}) = E\ `-'\ T \to E$,
  $\sigma^E(\mathit{et}) = T \to E$,
  $\sigma^E(\mathit{mul}) = T\ `*'\ F \to T$,
  $\sigma^E(\mathit{div}) = T\ `/'\ F \to T$,
  $\sigma^E(\mathit{tf}) = F \to T$,
  $\sigma^E(\mathit{nest}) = `('\ E\ `)' \to F$,
  $\sigma^E(\mathit{neg}) = `-'\ F \to F$,
  $\sigma^E(\mathit{const}) = \mathit{digits} \to F$,
  $\sigma^E(\mathit{digits}) = \mathit{String} \to \mathit{digits}$,
  $\sigma^E(`+') = \epsilon \to `+'$,
  $\sigma^E(`-') = \epsilon \to `-'$, ...

Target Language: $\Sigma^A = \langle S^A, F^A, \sigma^A \rangle$ where

- $S^A = \{A, \mathit{String}\}$
- $F^A = \{\mathit{plus}, \mathit{minus}, \mathit{times}, \mathit{divide}, \mathit{constant}\}$
- $\sigma^A(\mathit{plus}) = A\ A \to A$,
  $\sigma^A(\mathit{minus}) = A\ A \to A$,
  $\sigma^A(\mathit{times}) = A\ A \to A$,
  $\sigma^A(\mathit{divide}) = A\ A \to A$,
  $\sigma^A(\mathit{constant}) = \mathit{String} \to A$

Fig. 5. Concrete and abstract syntax of arithmetic expressions.

- $\{A_s\}_{s \in S}$ – an S-indexed family of sets, called *carrier sets*
- $\{f^A : A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \to A_s \mid f \in F, \sigma(f) = S_1 \times S_2 \times \ldots \times S_n \to S\}$. For each function name $f \in F$ there is a function f^A over the appropriate carrier sets in $\{A_s\}_{s \in S}$ as indicated by the signature of f, $\sigma(f)$.

Word algebras, over variables, specify (ground) terms or patterns, if variables are used, and are denoted $W_\Sigma(V)$ for variables V. An example of a word is $\mathit{plus}(\mathit{mul}(a, b), c)$, where a, b and c are variables. We will find a need to order patterns (words with variables) from least specific to most specific. We use a standard notion of specificity in that one word is more specific than another if the set of ground terms created from all instantiations is a subset of such ground terms for another pattern.

Our first example is given in Fig. 5 and shows the operator scheme Σ^E for the source language E and Σ^A for the target language A. Some readers will be more familiar with the BNF notation for context free grammars, which is similar to algebras and can be easily translated into this algebraic setting. Nonterminal and terminal symbols become sorts. The sorts E, T, F in S^E correspond to the nonterminals commonly used in this example. The sort *digits* represents an integer literal terminal, and the operator and punctuation symbols are given sort names by quoting the symbol. For example, $`+'$ corresponds to the terminal symbol for the addition symbol. Strings are also used and play the role of lexemes on scanned tokens; thus we have the sort *String*.

The productions in a grammar correspond to operators; in this example: $\mathit{add}, \mathit{sub}, \mathit{et}, \mathit{mul}, \mathit{div}, \mathit{tf}, \mathit{nest}, \mathit{neg}, \mathit{const} \in F^E$ for the concrete syntax. The signature of each of these operators is given by σ^E and is written "backwards" from how they appear in BNF. For example, the operator *add* is a ternary operator taking values of sort E, $`+'$, and $T$ and creating values of type $E$, as denoted by $\sigma^E(\mathit{add}) = E\ `+'\ T \to E$.

There is also a single operator for each terminal symbol. If the regular expression that would be associated with a terminal in its scanner specification is constant, then this operator is nullary. If it is not constant but identifies a pattern for, say, variable names or integer constants, then we make the signature unary with *String* being the single argument. We will refer to nullary terminal operators as constant, and unary terminal operators as non-constant. To avoid too much notational clutter we will overload sort and operator names for those corresponding to terminal symbols.

Trees in this language are written as terms or words from the corresponding word algebra, parametrized by a set of strings representing lexemes. This algebra is technically denoted $W_\Sigma(\mathit{String})$ but we omit *String* below. We overload *String* to denote the sort, as in Fig. 5, and here to denote the carrier set of strings.

7.1.1. Specifying the forward transformation

As in most approaches to bidirectional transformation, the forward transformation is provided and used to generate the backward transformation. Here we describe the structure of the forward specifications used in our approach and provide the forward transformation specification from Σ^E to Σ^A, which is shown in Fig. 6.

In defining the forward transformation, the first part of the specification is the *sort-map*, which in our approach will be generalized so that the range of the sort map is a set of sorts in the target. In our first example, this maps all of the source sorts of expressions (E, T, and F) to the single sort for expressions in the target/abstract scheme A.

– The sort map: $sm^{get} :: F^E \to 2^{F^A}$
$sm^{get}(E) = \{A\}, \quad sm^{get}(T) = \{A\}, \quad sm^{get}(F) = \{A\}$
– The rewrite rules rw^{get}:

$$
\begin{array}{l}
get_A^E(add(l, \text{'}+\text{'}, r)) \to plus(get_A^E(l), get_A^T(r)) \\
get_A^E(sub(l, op, r)) \to minus(get_A^E(l), get_A^T(r)) \\
get_A^E(et(t)) \to get_A^T(t) \\
get_A^T(mul(l, \text{'}*\text{'}, r)) \to times(get_A^T(l), get_A^F(r)) \\
get_A^T(div(l, \text{'}/\text{'}, r)) \to divide(get_A^T(l), get_A^F(r)) \\
get_A^T(tf(f)) \to get_A^F(f) \\
get_A^F(neg(\text{'}-\text{'}, r)) \to minus(constant(\text{“}0\text{”}), get_A^F(r)) \\
get_A^F(nest(\text{'('}, e, \text{')'})) \to get_A^E(e) \\
get_A^F(const(digits(d)) \to constant(d)
\end{array}
$$

Fig. 6. Forward transformation specification.

The patterns used in the rewrite rules to specify the translation from the source to the target are not merely terms from the (word algebra of the) source or target language (extended with variables). We create additional operators, based on the sort map, whose signatures include sorts from both the source language and the target language. From a sort map sm, we create additional operators:

$$\{get_Y^X \mid sm(X) = Y\}$$

indicating that the forward (get) transformation maps an X in the source to a Y in the target. The signatures for such operators are as expected:

$$\sigma^{get}(get_Y^X) = Y \to X, \forall X \in S^S, Y \in sm(X) .$$

The left and right hand sides of the rewrite rules are then words in a word algebra for the operator scheme that includes both the source and target operator schemes and these attribute-like operators. Left hand side and right hand side patterns are words in $W_{\Sigma^{get}}(V)$ for a sort-indexed set of variable names V. Both the left and right hand side are terms of the same target language sort. Note that we do not have rules for sorts corresponding to terminal symbols; they have no translation in the target.[8]

7.2. Generating the backward transformation

In this section we describe the process for inverting the forward transformation to generate the backward one.

Inverting the sort map and rewrite rules The first step is to invert the sort map. In our example this inversion leads to a sort map sm^{put} that maps the abstract sort A back to three concrete sorts E, T, and F. The inverted sort map now maps target sorts to multiple source sorts. Thus, we really are defining 3 put transformations: putting an A back to an E, back to a T, and back to an F. This is the basis for the *put* operators that are analogous to the *get* operators seen above.

The second step is to invert the rewrite rules. The result of this process for the rules in Fig. 6 produces the rules in Fig. 7. Given the restriction on the forward transformation, this process is relatively straightforward. A rule of the form

$$get_Y^X(w(v_1, \ldots, v_n)) \to w'(get_{Y_1}^{X_1}(v_1), \ldots, get_{Y_1}^{X_1}(v_n))$$

where v_i is of sort X_i and $sm(X_i) = \{Y_i\}$ is inverted to form the rule

$$put_X^Y(w'(v'_1, \ldots v'_n)) \to w(put_{X_1}^{Y_1}(v'_1), \ldots, put_{X_n}^{Y_n}(v'_n))$$

in which the variables v'_i are of sort Y_i. This can be seen in the inverted rules in Fig. 7.

Extending the rules Consider a transformation in an abstract syntax that creates the subtree $times(\_, plus(\_, \_))$. While the second argument of *times* is of sort *F*, *plus* maps most directly back to an *E*. Thus the backward transformation must create a source term of type F from term $plus(\_, \_)$.

The key to solve this problem lies in the rules with a right hand side of the form $put_X^Y(v) \to w(put_{X'}^Y(v))$, where w is a word containing the sub-word $put_{X'}^Y(v)$ which holds the only variable, namely v. Such a rule shows how to transform any term of type X' in the source language to one of type X in the source language. For example, the rule $put_F^A(e) \to$

[8] In [19] we explain how attribute grammar equations are derived from this transformations. Here, it suffices to understand the semantic transformation involved.

– The sort map: $sm^{put} :: F^A \to 2^{F^E}$
$sm^{put}(A) = \{E, T, F\}$
– The rewrite rules rw^{put}

$$
\begin{aligned}
&put_E^A(plus(l,r)) \to add(put_E^A(l), \text{`}+\text{'}, put_T^A(r))\\
&put_E^A(minus(l,r)) \to sub(put_E^A(r), \text{`}-\text{'}, put_T^A(r))\\
&put_E^A(minus(constant(\text{“}0\text{”}), r)) \to neg(\text{`}-\text{'}, put_F^A(r))\\
&put_E^A(t) \to et(put_T^A(t))\\
&put_T^A(times(l,r)) \to mul(put_T^A(l), \text{`}*\text{'}, put_F^A(r))\\
&put_T^A(divide(l,r)) \to div(put_T^A(l), \text{`}/\text{'}, put_F^A(r))\\
&put_T^A(f) \to tf(put_F^A(f))\\
&put_F^A(e) \to nest(\text{`}(\text{'}, put_E^A(e), \text{`})\text{'})\\
&put_F^A(constant(d)) \to const(digits(d))
\end{aligned}
$$

Fig. 7. Direct inversion of forward transformation specification.

$nest(\text{`}(\text{'}, put_E^A(e), \text{`})\text{'})$ in Fig. 7 shows that a term of type E can be converted to one of type F by wrapping it in parentheses, that is, in the term $nest(\text{`}(\text{'}, \_, \text{`})\text{'})$.

We specialize the inverted rewrite rules of this form so that their left hand sides are of the form $put_Y^X(p(v_1, \ldots, v_n))$ for $p \in F^A$ in which $v_1, \ldots, v_n$ are variables of the appropriate type:

1. If
   (a) $\exists X \in S^S$ and $Y \in sm(X)$ such that there does not exist a rewrite rule whose left hand side has the form $put_Y^X(p(v_1, \ldots, v_n))$ for some $p \in F^S$ such that return type of p is X (for some α, $\sigma^\Sigma(p) = \alpha \to X$) and for some variables $v_1, \ldots, v_n$, and
   (b) there exists a rule of the form $put_Y^X(t) \to w(put_{Y'}^X(t))$
   then add the rule $put_Y^X(p(v_1, \ldots, v_n)) \to w(put_{Y'}^X(p(v_1, \ldots, v_n)))$
2. Repeat step 1 until no more rules can be added.

For example, as noted above, the rule $put_F^A(e) \to nest(\text{`}(\text{'}, put_E^A(e), \text{`})\text{'})$ in Fig. 7 allows parenthesization of an E to form an F. The steps above would then add the following rule:

$$put_F^A(plus(l,r)) \to nest(\text{`}(\text{'}, put_E^A(plus(l,r)), \text{`})\text{'}) ,$$

and then repeat this process until we have as many extended rules as possible.

This mechanism is aided by maintaining information regarding the original tree, which allows transformations to be contextual, i.e., have a notion of the original tree so information that goes through the bidirectional system holds its integrity and structure as much as possible. For example in some cases, where no change was applied to the information (or to parts of it), it implies only linking to the original source.

This technique is also powerful enough to support non-linear, compound rules, but there are cases were aggressive transformations put the structure of the tree out of the domain of the transformations, forcing the programmer to write “tree repair”, and re-shape the tree so it can be accepted as input by the transformations.

7.3. Bidirectionalization with zipper-based AGs

Our previous work relied on the AG system Silver, with its distinctive characteristics, to develop a bidirectional environment. In this section, we show how our new zipper-based setting has enough expressiveness to support the same semantics as we have previously defined using Silver-specific features.

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:

```haskell
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
```

$$
\begin{array}{lll}
 & | & \mathit{Sub}\ E\ T \\
 & | & \mathit{Et}\quad T \\
\textbf{data}\ T = & & \mathit{Mul}\ T\ F \\
 & | & \mathit{Div}\ \ T\ F \\
 & | & \mathit{Tf}\quad F \\
\textbf{data}\ F = & & \mathit{Nest}\quad E \\
 & | & \mathit{Neg}\quad F \\
 & | & \mathit{Var}\quad \mathit{String} \\
 & | & \mathit{Const}\ \mathit{Int}
\end{array}
$$

Note this representation is directly mapped from the one presented in Fig. 5. It 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[9]:

```
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 parentheses, but this is far from ideal as it unnecessarily produces a complicated concrete representation.

Another problem in defining a bidirectional system is illustrated 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 did not write `0-1` on the abstract tree on purpose).

The differences between the concrete and abstract representations of `LET` add difficulties when writing the transformations. In previous work [19] we have developed an automatic bidirectionalization system that can use two context free

9 The production *Neg* creates additional difficulties, therefore it is omitted on purpose and will be discussed later.

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 [16].

Despite our previous work generating code as an AG Domain Specific Language (DSL) in Silver, our zipper-based 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:

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

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 all the AGs we have seen in the previous sections are still semantically valid. Recall that in the embedding presented in this paper we address children by their ordering number, which means that adding more children to the end of a tree node does not change the addresses of the existing ones.

In our setting, the transformations are represented by a set of synthesized attributes *get* that are named $get_{From \to To}$, 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*:

$$\begin{array}{l} get_{RootC \to Root} :: \mathit{Zipper\ RootC} \to \mathit{Root} \\ get_{RootC \to Root}\ ag = \textbf{case}\ (\mathit{constructor\ ag})\ \textbf{of} \\ \quad \text{“}\mathit{RootC}\text{”} \to \mathit{Root}\ (get_{LetC \to Let}\ (ag\ .\$\ 1))\ (\mathit{createLink\ ag}) \end{array}$$

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 previously: now, abstract nonterminals (*Expr*) can map to more than one in the concrete side (*E*, *T* or *F*).

The *put* attribute (defining the backward transformation) for *Add*, for example, will ask for $put_{Expr \to E}$ of its left child and $put_{Expr \to T}$ 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 avoid naively wrapping every sub-expression in parentheses, 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*:

$$\begin{array}{l}
put_{Expr \to F} :: \mathit{Zipper\ Root} \to F \\
put_{Expr \to F}\ ag = \textbf{case}\ (\mathit{getLink\ ag})\ \textbf{of} \\
\quad \mathit{IsE\ e} \quad \to \mathit{Nest\ e} \\
\quad \mathit{IsT\ t} \quad \to \mathit{Nest}\ (\mathit{Et\ t}) \\
\quad \mathit{IsF\ f} \quad \to f \\
\quad \mathit{Empty} \to \textbf{case}\ (\mathit{constructor\ ag})\ \textbf{of} \\
\qquad \text{“}\mathit{Plus}\text{”} \quad \to \textbf{let}\ \mathit{left} \ \ = put_{Expr \to E}\ (ag\ .\$\ 1) \\
\qquad\qquad\qquad\qquad \mathit{right} = put_{Expr \to T}\ (ag\ .\$\ 2) \\
\qquad\qquad\qquad \textbf{in}\ \ \mathit{Nest}\ (\mathit{Add\ left\ right}) \\
\qquad \text{“}\mathit{Minus}\text{”} \quad \to \\
\qquad\quad \textbf{case}\ (\mathit{getHole\ ag} :: \mathit{Maybe\ Expr})\ \textbf{of} \\
\qquad\qquad \mathit{Just}\ (\mathit{Minus}\ (\mathit{Constant}\ 0\ _)\ _\ _) \to \mathit{Neg}\ (put_{Expr \to F}\ (ag\ .\$\ 2)) \\
\qquad\qquad \mathit{otherwise} \to \textbf{let}\ \mathit{left} \ \ = put_{Expr \to E}\ (ag\ .\$\ 1) \\
\qquad\qquad\qquad\qquad\qquad \mathit{right} = put_{Expr \to T}\ (ag\ .\$\ 2) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \ \mathit{Nest}\ (\mathit{Sub\ left\ right}) \\
\qquad \text{“}\mathit{Times}\text{”} \quad \to \textbf{let}\ \mathit{left} \ \ = put_{Expr \to T}\ (ag\ .\$\ 1) \\
\qquad\qquad\qquad\qquad \mathit{right} = put_{Expr \to F}\ (ag\ .\$\ 2) \\
\qquad\qquad\qquad \textbf{in}\ \ \mathit{Nest}\ (\mathit{Et}\ (\mathit{Mul\ left\ right})) \\
\qquad \text{“}\mathit{Divide}\text{”} \quad \to \textbf{let}\ \mathit{left} \ \ = put_{Expr \to T}\ (ag\ .\$\ 1) \\
\qquad\qquad\qquad\qquad \mathit{right} = put_{Expr \to F}\ (ag\ .\$\ 2) \\
\qquad\qquad\qquad \textbf{in}\ \ \mathit{Nest}\ (\mathit{Et}\ (\mathit{Div\ left\ right})) \\
\qquad \text{“}\mathit{Constant}\text{”} \to \mathit{Const}\ (\mathit{lexeme}_{\mathit{Constant}_1}\ ag) \\
\qquad \text{“}\mathit{Variable}\text{”} \ \to \mathit{Var} \quad (\mathit{lexeme}_{\mathit{Variable}_1}\ ag)
\end{array}$$

There are a couple of important remarks regarding the implementation of $put_{Expr \to F}$:

- 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* $\to$ *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 [19], 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 $put_{Expr \to E}$ and $put_{Expr \to T}$ and their definitions are very similar to the one of $put_{Expr \to F}$.

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 parentheses. 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 [19].

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, automatically making the boilerplate code that was until now implemented by the user.

8. Related work

In this paper, we have proposed a zipper-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 functional languages without extensions. 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* [35], *HTML table formatting* [13], and *smart parentheses*, an illustrative example of [16], that are available through the cabal package `ZipperAG`.[10]

Moreover, the navigation via a generic zipper that we envision here has applications in other domains: *i*) our setting is being used to create combinator languages for process management [36] which themselves are fundamental to a platform for open source software analysis and certification [37,38]; and *ii*), the setting that we propose was applied in 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 [39]. 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.

[10] http://hackage.haskell.org/package/ZipperAG

Badouel et al. define attribute evaluation as zipper transformers [40]. While their encoding is simpler than that of Uustalu and Vene, they also use laziness as a key aspect and the zipper representation is similarly not generic. Other work by Badouel et al. [41] 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 [42]. 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 advanced features of `Haskell` such as type families and rank-2 types.

Finally, none of the above-mentioned zipper-based approaches to AGs has shown how to deal with standard AG extensions or how to support bidirectional transformations.

8.2. Non-zipper-based approaches

Circular programs have been used in the past to straightforwardly implement AGs in a lazy functional language [43,44]. These works, in contrast to our own, rely on the target language being 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 [7,8,10], they do not offer the modern AG extensions that we provide, with the exception of [10] 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 [7] or heterogeneous collections [8,10]. The use of heterogeneous lists in the second of these approaches replaces the use in the first approach of extensible records, which are no longer supported by the main `Haskell` compilers. In our framework, attributes do not need to be collected in a data structure at all: they are regular functions upon which correctness checks are statically performed by the compiler. The result is a simpler and more modular embedding. On the other hand, the use of these data structures ensures that an attribute is computed only once, being then updated to a data structure and later found there when necessary. In order to guarantee such a claim in our setting we need to rely on memorization strategies, 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. [8,10]. 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.

The work presented in [45,46] has very similar goals to our work: it discusses the combination of circular, higher-order and reference AGs. This work, however, expresses such combination of AG extensions on the generative JastAdd AG system [18], and not as a pure embedding as we propose in our zipper-based setting.

Recently, Norell and Gerdes have proposed an elegant embedding of AGs in `Erlang` [47]. While this embedding allows attributes to be used when generating data and does not rely on lazy evaluation, it still does not include the extensions we provide.

Furthermore, we should stress that none of the mentioned approaches has shown how to support bidirectional transformations.

8.3. Bidirectional transformations

Data transformations are an active research topic with multiple strategies applied in various fields, some with a particular emphasis on rule-based approaches. Czarnecki and Helsen [48] 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 [49], programmable structure editors [50], model-driven development in software engineering [51], among others. Czarnecki and colleagues have written a detailed review [52] 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 [53]. BOTL [54], 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 in either direction.

A well-regarded approach to bidirectionalization systems is through lens combinators [49,55]. 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 by Matsuda et al. uses a language for specifying transformations very similar to the one presented in this work, with automatic derivation of the backward transformation [56]. 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 in our setting, where decisions between mapping different sets of nonterminals are completely automated.

In the context of attribute grammars, Yellin's early work on bidirectional transformations in AGs defined attribute grammar inversion [57]. 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 implementations 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 as modern, large and more complex attribute grammar 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 our 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 from having to write complex attribute equations that have to perform multiple pattern matching, manage both the links back and their types, prioritize 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:

- Improve attribute definition by referencing non-terminals instead of (numeric) positions on the right-hand side of productions.
- Wherever possible, benchmark our embedding against other AG embeddings and systems.
- We would like to evaluate this embedding together with all the extensions presented on a number of mainstream syntactically rich languages.

References

[1] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (1968) 127–145, corrections in Math. Syst. Theory 5 (1971) 95–96.
[2] A. Dijkstra, J. Fokker, S.D. Swierstra, The architecture of the Utrecht Haskell compiler, in: S. Weirich (Ed.), Haskell, ACM, 2009, pp. 93–104.
[3] D. Swierstra, P. Azero, J. Saraiva, Designing and implementing combinator languages, in: Advanced Functional Programming, in: LNCS, vol. 1608, Springer-Verlag, 1999, pp. 150–206.
[4] J. Saraiva, D. Swierstra, Data structure free compilation, in: Stefan Jähnichen (Ed.), 8th International Conference on Compiler Construction, CC/ETAPS'99, in: LNCS, vol. 1575, Springer-Verlag, 1999, pp. 1–16.
[5] J.P. Fernandes, J. Saraiva, Tools and libraries to model and manipulate circular programs, in: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'07, ACM, 2007, pp. 102–111.
[6] A. Middelkoop, A. Dijkstra, S.D. Swierstra, Iterative type inference with attribute grammars, in: E. Visser, J. Järvi (Eds.), GPCE, ACM, 2010, pp. 43–52.
[7] O. de Moor, K. Backhouse, S.D. Swierstra, First-class attribute grammars, Informatica (Slovenia) 24 (3) (2000).
[8] M. Viera, D. Swierstra, W. Swierstra, Attribute grammars fly first-class: how to do aspect oriented programming in Haskell, in: Proceedings of the 14th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'09, 2009, pp. 245–256.
[9] P. Martins, J.P. Fernandes, J. Saraiva, Zipper-based attribute grammars and their extensions, in: Proceedings of the 17th Brazilian Symposium on Programming Languages, SBLP'13, in: LNCS, vol. 8129, Springer, 2013, pp. 135–149.
[10] M. Viera, First class syntax, semantics, and their composition, PhD thesis, Utrecht University, The Netherlands, 2013.
[11] A.M. Sloane, L.C.L. Kats, E. Visser, A pure embedding of attribute grammars, Sci. Comput. Program. 78 (2013) 1752–1769.
[12] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, SIGPLAN Not. 24 (7) (1989) 131–145.
[13] J. Saraiva, S.D. Swierstra, Generating spreadsheet-like tools from strong attribute grammars, in: F. Pfenning, Y. Smaragdakis (Eds.), GPCE, in: LNCS, vol. 2830, Springer, 2003, pp. 307–323.
[14] J.T. Boyland, Remote attribute grammars, J. ACM 52 (4) (2005) 627–687.
[15] E. Magnusson, G. Hedin, Circular reference attributed grammars – their evaluation and applications, Sci. Comput. Program. 68 (1) (2007) 21–37.

[16] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Sci. Comput. Program. 75 (1–2) (2010) 39–54.
[17] T. Kaminski, E. Van Wyk, Integrating attribute grammar and functional programming language features, in: Proc. of the 4th Intl. Conf. on Software Language Engineering, SLE 2011, in: Lecture Notes in Computer Science, vol. 6940, Springer-Verlag, 2011, pp. 263–282.
[18] T. Ekman, G. Hedin, The jastadd extensible Java compiler, SIGPLAN Not. 42 (10) (2007) 1–18.
[19] P. Martins, J. Saraiva, J.P. Fernandes, E. Van Wyk, Generating attribute grammar-based bidirectional transformations from rewrite rules, in: Proceedings of the 2014 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'14, ACM, 2014, pp. 63–70.
[20] G. Huet, The zipper, J. Funct. Program. 7 (5) (1997) 549–554.
[21] S. Peyton Jones, J. Hughes, L. Augustsson, et al., Report on the programming language Haskell 98, Technical report, 1999.
[22] R. Milner, M. Tofte, D. Macqueen, The Definition of Standard ML, MIT Press, Cambridge, MA, USA, 1997.
[23] J. Paakki, Attribute grammar paradigms — a high-level methodology in language implementation, ACM Comput. Surv. 27 (2) (1995) 196–255.
[24] R. Bird, Introduction to Functional Programming Using Haskell, 2nd edn., Prentice Hall PTR, 1998.
[25] M.D. Adams, Scrap your zippers: a generic zipper for heterogeneous types, in: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming, WGP'10, ACM, New York, NY, USA, 2010, pp. 13–24.
[26] R. Lämmel, S.P. Jones, 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, pp. 26–37.
[27] M. Kuiper, J. Saraiva, LRC – a generator for incremental language-oriented tools, in: Procs. of Compiler Construction (CC), in: LNCS, vol. 1383, Springer-Verlag, 1998, pp. 298–301.
[28] E. Magnusson, G. Hedin, Circular reference attributed grammars: their evaluation and applications, Sci. Comput. Program. 68 (1) (2007) 21–37.
[29] J. Saraiva, Purely functional implementation of attribute grammars, PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1999.
[30] R. Farrow, Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars, SIGPLAN Not. 21 (7) (1986) 85–98.
[31] L.G. Jones, Efficient evaluation of circular attribute grammars, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 429–462.
[32] A. Sasaki, M. Sassa, Circular attribute grammars with remote attribute references and their evaluators, New Gener. Comput. 22 (1) (2004) 37–60.
[33] T. Reps, T. Teitelbaum, The Synthesizer Generator, Springer-Verlag, 1989.
[34] E. Söderberg, Contributions to the construction of extensible semantic editors, PhD thesis, Lund University, Sweden, 2012.
[35] R. Bird, Using circular programs to eliminate multiple traversals of data, Acta Inform. 21 (1984) 239–250.
[36] P. Martins, J.P. Fernandes, J. Saraiva, A purely functional combinator language for software quality assessment, in: Symposium on Languages, Applications and Technologies, SLATE'12, in: OASICS, vol. 21, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012, pp. 51–69.
[37] P. Martins, J.P. Fernandes, J. Saraiva, A web portal for the certification of open source software, in: A. Cerone, D. Persico, S. Fernandes, A. Garcia-Perez, P. Katsaros, S.A. Shaikh, I. Stamelos (Eds.), SEFM Satellite Events, in: Lecture Notes in Computer Science, vol. 7991, Springer, 2012, pp. 244–260.
[38] P. Martins, N. Carvalho, J.P. Fernandes, J.J. Almeida, J. Saraiva, A framework for modular and customizable software analysis, in: 13th Int. Conf. on Computational Science and Its Applications, ICCSA 2013, in: LNCS, vol. 7972, 2012, pp. 443–458.
[39] T. Uustalu, V. Vene, Comonadic functional attribute evaluation, in: Trends in Functional Programming, vol. 10, Intellect Books, 2005, pp. 145–162.
[40] E. Badouel, R. Tchougong, C. Nkuimi-Jugnia, B. Fotsing, Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition, Theor. Comput. Sci. 480 (0) (2013) 1–25.
[41] E. Badouel, B. Fotsing, R. Tchougong, Attribute grammars as recursion schemes over cyclic representations of zippers, Electron. Notes Theor. Comput. Sci. 229 (5) (2011) 39–56.
[42] A.R. Yakushev, S. Holdermans, A. Löh, J. Jeuring, Generic programming with fixed points for mutually recursive datatypes, in: Procs. of the 14th ACM SIGPLAN International Conference on Functional Programming, 2009, pp. 233–244.
[43] T. Johnsson, Attribute grammars as a functional programming paradigm, in: Functional Programming Languages and Computer Architecture, 1987.
[44] M. Kuiper, D. Swierstra, Using attribute grammars to derive efficient functional programs, in: Computing Science in the Netherlands, 1987.
[45] E. Söderberg, G. Hedin, Circular higher-order reference attribute grammars, in: M. Erwig, R. Paige, E. Wyk (Eds.), Software Language Engineering, in: Lecture Notes in Computer Science, vol. 8225, Springer International Publishing, 2013, pp. 302–321.
[46] E. Söderberg, G. Hedin, Declarative rewriting through circular nonterminal attributes, Comput. Lang. Syst. Struct., Part A 44 (2015) 3–23, Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).
[47] U. Norell, A. Gerdes, Attribute grammars in Erlang, in: Proceedings of the 14th ACM SIGPLAN Workshop on Erlang, Erlang 2015, ACM, 2015, pp. 1–12.
[48] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (3) (2006) 621–645.
[49] A. Bohannon, B.C. Pierce, J.A. Vaughan, Relational lenses: a language for updatable views, in: Procs. of ACM Principles of Database Systems, PODS, ACM, 2006, pp. 338–347.
[50] Z. Hu, S.C. Mu, M. Takeichi, A programmable editor for developing structured documents based on bidirectional transformations, in: Procs. of Partial Evaluation and Program Manipulation, PEPM, ACM, 2004, pp. 178–189.
[51] P. Stevens, A landscape of bidirectional model transformations, in: Generative and Transformational Techniques in Software Engineering II, in: LNCS, vol. 5235, Springer-Verlag, 2008, pp. 408–424.
[52] K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, J.F. Terwilliger, Bidirectional transformations: a cross-discipline perspective, in: Procs. of Theory and Practice of Model Transformations, ICMT, in: LNCS, vol. 5563, Springer-Verlag, 2009, pp. 260–283.
[53] F. Jouault, I. Kurtev, Transforming models with ATL, in: Procs. of Satellite Events at the MoDELS, in: LNCS, vol. 3844, Springer-Verlag, 2006, pp. 128–138.
[54] M. Hibberd, M. Lawley, K. Raymond, Forensic debugging of model transformations, in: Procs. of Model Driven Engineering Languages and Systems, in: LNCS, vol. 4735, Springer-Verlag, 2007, pp. 589–604.
[55] J. Foster, M. Greenwald, J. Moore, B. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem, ACM Trans. Program. Lang. Syst. 29 (3) (2007) 17.
[56] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, M. Takeichi, Bidirectionalization transformation based on automatic derivation of view complement functions, in: Procs. of ACM SIGPLAN International Conference on Functional Programming, ICFP, ACM, 2007, pp. 47–58.
[57] D.M. Yellin, Attribute Grammar Inversion and Source-to-Source Translation, LNCS, vol. 302, Springer-Verlag, 1988.