Universidade do Minho

Escola de Engenharia

Ph.D. Thesis

Embedding Attribute Grammars and their Extensions using Functional Zippers

---

Author:
Pedro Martins

Supervisor:
Prof. Dr. João Saraiva
Co-Supervisor:
Prof. Dr. João Paulo Fernandes

Acknowledgments

I am writing this late at night. Every time I roll my eyes in either direction they hurt, and I have long ago forgot about back pain and maintaining a good posture. I am tired, stressed and slightly upset and worried. This is exactly how I thought it would be. I will miss every small part of these past four years.

I want to use this moment to show my gratitude to the people that in many ways contributed to the work I am presenting here. I am doing it without any particular order, as I honestly think I would not be capable of finishing this without any of them.

I start by showing my gratitude to my supervisors, João Saraiva and João Paulo. They have been with me for more than four years and yet, somehow, they always had the confidence and the right words when I much needed them. The moments I spent with them changed me in ways I simply cannot find the words to describe.

Eric Van Wyk kindly accepted me during my stay in Minneapolis. His commitment, knowledge and approach to research provided me with a very valuable environment on which I could idealize and develop this work. More than that, he and Peggy gave me such a warm welcome to the United States that I will always remember the great moments we spent together.

Luís Florêncio and Cátia Silva played a big part in providing moral support, through humor and mature advising. Ironically, I think I could handle myself without the latter, but never without the former.

I was lucky for having such an amazing environment as I worked in the best office with the best people. Rui, Jorge, Faria, Alpuim, Cláudio, Tiago, David, Sara, Jácome: you were the best lab and summer school mates I could

ask for.

My brother Luís was more than a brother, he was also my roommate during all this time. This means he was the one that had to deal with all the frustration and grumpyness of a grad student arriving home everyday. Not an easy task, but a crucial one, and something I will forever be in his debt for (when you read this man, this is NOT an excuse for not cleaning the kitchen every once in a while!!).

Finally, I would like to thank my parents José Amaro and Maria Isabel. Their education and support and love and hard work made everything possible, and no one deserves a bigger credit for the work presented here. This work is something they achieved before I did, this is something that is theirs before being mine, something they should feel proud of before I do. It is the result of their dedication and parenting before being anything else.

Several institutions also contributed to this thesis in important ways. This work was mainly supported by Fundação para a Ciência (FCT), by the European Regional Development Fund (ERDF) through the Programme Compete, by the MIT Portugal Program, a large-scale international collaboration involving MIT and government, academia, and industry in Portugal, by the Luso-American Foundation (FLAD) and by the National Science Foundation (NSF).

In particular, I received grants from the projects AMADEUS (PTDC/EIA/70271/2006), ref^a BI-1_PTDC/EIA/70271/2006; CROSS (FCOMP-01-0124-FEDER-010049), ref^a BI3-2011_PTDC/EIA-CCO/108995/2008; SAVED (MIT-PT/TS-ITS/0036/2008), FATBIT (FCOMP-01-0124-FEDER-020532) and BEST CASE (NORTE-01-0124-FEDER-000058), ref^a BIM-2013_BestCase_RL3.2_UMINHO.

Abstract

Embedding Attribute Grammars and their Extensions using Functional Zippers

Attribute grammars are a suitable formalism to express complex software language analysis and manipulation algorithms, which rely on multiple traversals of the underlying syntax tree. Attribute Grammars have been extended with mechanisms such as references, higher order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex computations.

In this work we defined 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.

We have also developed techniques to describe automatic bidirectional transformations between grammars. We define a method to define transformation specifications which, through our automatic mechanisms, are inverted and expanded and generate attribute grammars that specify a bidirectional environment.

We have implemented several real examples of language specification and processing in our setting, some of which are presented in this work. We have also developed and implemented a DSL using our technique for embedding attribute grammars, which we have deployed in a web portal for software analysis.

Resumo

Gramáticas de Atributos e as suas Extencões Embebidas em *"Zippers"* Funcionais

Gramáticas de atributos são um formalismo que permite exprimir algoritmos complexos de análise e transformação de programas, que tipicamente requerem varias travessias às árvores abstractas que os representam. As gramáticas de atributos foram estendidas com mecanismos que permitem referências, ordem superior e circularidade em atributos. Estas extensões permitem a implementação de mecanismos complexos e modulares de computações em linguagens.

Neste trabalho embebemos gramáticas de atributos e as suas extensões de forma elegante e simples, através de uma técnica chamada *"zippers"*. Na nossa técnica, especificações de linguagens são definidas com um conjunto de componentes independentes de primeira ordem, que podem ser facilmente compostos para formar poderosos ambientes de processamento de linguagens.

Também desenvolvemos técnicas que descrevem transformações bidirecionais entre gramáticas. Definimos métodos de especificar transformações que, através de mecanismos completamente automáticos, são invertidas e estendidas e geram gramáticas de atributos que especificam o nosso ambiente bidirecional.

Com esta técnica foram implementados vários exemplos de especificação e processamento de linguagens, alguns dos quais estão definidos e explicados neste documento. Da mesma forma, criamos e desenvolvemos uma linguagem de domínio especifico usando a nossa técnica; linguagem essa que integramos

num portal que permite a criação de análises de programas completamente configurada para servir os requisitos particulares de cada utilizador.

Contents

Acronyms

AG Attribute Grammar.

AST Abstract Syntax Tree.

BNF Backus Naur Form.

BX Bidirectional Transformation.

CAG Circular Attribute Grammar.

CFG Context-free Grammar.

CFL Context-free Language.

CST Concrete Syntax Tree.

DDSL Deep Domain-specific Language.

DSL Domain-specific Language.

EBNF Extended Backus Naur Form.

GHC Glasgow Haskell Compiler.

GPL General Purpose Language.

HOAG Higher Order Attribute Grammar.

OSS Open Source Software.

RAG Reference Attributed Grammar.

SDSL Shallow Domain-specific Language.

Chapter 1

Introduction

> *"Indeed, we are convinced that the style of programming with attribute grammars helps the programmer to construct better functional programs. Thus, the question that arises immediately is whether it would be possible to incorporate the elegant style of attribute grammars writing directly within a functional programming language"* - [Saraiva, 1999]

In a world so overwhelmed with machines and electronics, where computers evolved from simple apparatus to crucial and permanent components of our daily lives, we can not forget the way we use to communicate with them. Nowadays, this communication is achieved through sentences (programs) written in a high level programming language. Indeed, programming languages are artificial languages we have created to instruct a computer on how to behave.

As the usage of computers became increasingly ubiquitous, programming languages had to adapt to new domains. Nowadays, computer programs range from tiny scripts written as a hobby by people which need simplicity, to huge platforms written by hundreds of programmers that are comfortable with considerable complexity.

Computer programs must evenly balance speed and efficiency in micro controllers and in supercomputers. They can be written for a satellite where software updates are extremely difficult, or for open source projects, where

supporting communities patch and improve them on a daily basis.

This heterogeneity, both in the scope and in the domain programming languages were being used motivated the appearance of Domain-Specific Languages (DSL). Domain-specific languages are languages specialized in a particular application domain, which distinguishes them from General Purpose Languages (GPL) such as `Java` or `C` because general purpose languages are languages designed with broadly applicable domains and are used to solve a wide range of problems. This means they sometimes lack specialized features for a specific problem, or fail in providing an interface or a syntax related to the particular problem we are handling.

A mathematician uses `MATLAB` for running a statistic simulation, an hardware engineer uses `Verilog` to program an electronic chip, an accountant uses a spreadsheet for implementing formulas for financial calculus or a database manager uses `MySQL` to edit and retrieve information. Two of the most used languages by web programmares are domain-specific: `XML` and `HTML`. In short, each of them uses a domain-specific language because they are perfectly suited for their specific needs.

A mathematician could use `C` instead of `MATLAB`, and so could an accountant. And a database manager could use `Java` to access a database. The problem is that the general purpose nature of these languages implies considerable complexity and abstraction by the user, whereas with a domain-specific language he has constructors and primitives whose sole purpose corresponds exactly to his needs. For example, `MATLAB` contains functions such as `annurate`, which calculate a periodic interest rate. This is possible in `C` or `Java`, but requires implementation and maintenance, whereas in `MATLAB` is simply available for free.

With software systems being use in a growing number of devices and application domains, such as hardware engineering, accounting or database management, it is desirable to design and implement special purpose languages that are tailored to the their specific characteristics and necessities.

However, the design and implementation of a new programming language from scratch can be costly, and there is the necessity of techniques that allow an easy definition and implementation of new domain-specific languages.

In the next section we will talk about different techniques used to put domain-specific languages into effect and make them usable.

1.1 Languages Design and Implementation

There are two methods for implementing a domain-specific language. On one side, there is the traditional option of defining a custom syntax and all the supporting machinery: parsers, interpreters, compilers, perhaps an editor. This option has the advantage of allowing fine-tuning of the syntax to closely resemble the primitives and constructs of the language, and editors oriented for the language provide environments that simplify and speed up implementations.

With a language-oriented environment, errors can be customized so they represent the exact domain we are working in. For example, if we consider the syntactic analysis of a language, parser generator systems like `yacc` [Brown et al., 1992] or `ANTLR` [Parr, 2013] will provide the user with errors directly related to the language being processed, such as badly defined language grammars.

The problem with defining a language processor from scratch is that usually a big effort is required for implementing and maintaining all this additional software. Moreover there will be a continuous necessity of these resources, as improvements on the language mean improvements on the software itself.

The other possibility for implementing a language is by embedding the domain-specific language into a general-purpose one [Hudak, 1996], where we try to retain as much as possible of the target language but raise the level of abstraction to a specific domain. This means all the features of the general-purpose language are still available, and the communities that research the GPL are inadvertently also improving the features available to the domain-specific language. Furthermore, a general-purpose language already has optimized compilers and editors that we can use and are available for free for our embedded domain-specific language.

Figure 1.1 shows the two different ways of implementing a DSL, with

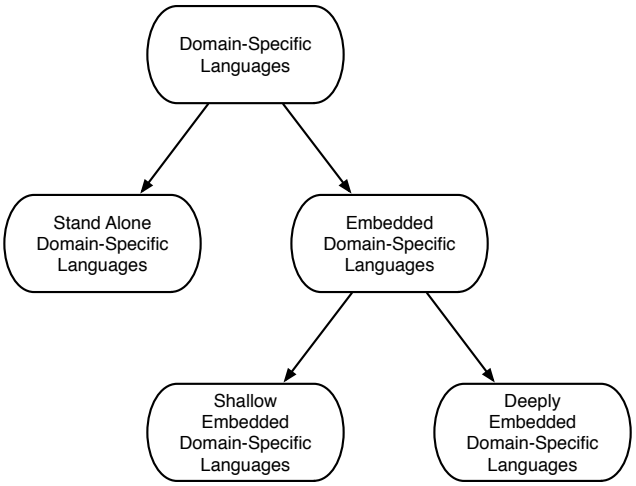


Figure 1.1: Implementing DSLs.

more traditional approach on the left and the embedding on the right. When embedding a language, two further techniques exist [Gill, 2014], which relate to the semantic meaning of the DSL in the host language.

A language can be implemented using a shallow embedding. Here, the shallow DSL (SDSL) is described in a way that represents a computation that a value, which is the meaning of the language.

On the other side we have deep embedded DSLs (DDSL), where the languages creates an abstract representation instead of a value. The resulting structure can be further analyzed and computed or be the subject of further transformations.

Regardless of the chosen strategy, processing a language traditionally involves a series of steps, which can be roughly split in:

1. Performing lexical analysis, which implies converting the textual format of the language into tokens, which are sequences of characters relevant as a group;

2. Formalizing the structure of the language, where we syntactically define the correctness of the language;

3. Formalizing the way the language behaves (the semantics), using information from the previous steps.

One example of a formalism that allows the definition of these steps are attribute grammars.

1.2 Attribute Grammars

Attribute Grammars (AGs) [Knuth, 1968] 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` [Dijkstra et al., 2009], but also to specify sophisticated pretty printing algorithms [Swierstra et al., 1999], deforestation techniques [Fernandes and Saraiva, 2007], powerful type systems [Middelkoop et al., 2010], syntax editors [Jourdan et al., 1990], programming environments [Kuiper and Saraiva, 1998], visual languages [Kastens and Schmidt, 2002] or program animations [Saraiva, 2002].

Attribute grammars have also proven to be a suitable formalism for the design and implementation of both domain specific and general purpose languages, with powerful systems based on attribute grammars [Reps and Teitelbaum, 1989; Jourdan et al., 1990; Kastens and Schmidt, 2002; Luković et al., 2011] being constructed.

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.

The original formulation of attribute grammars was improved through various extensions, that improve their expressiveness and the scope of problems they can deal with. Higher-order AGs (HOAGs) [Vogt et al., 1989; Saraiva and Swierstra, 2003] provide a modular extension to AGs in which syntax trees can be stored as attribute values. Reference AGs (RAGs) [Hedin, 1999] 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.

More recently, new extensions and features have been defined for attribute grammars, like forwarding attribute grammars [Van Wyk et al., 2002], multiple inheritance [Mernik et al., 2000, 2005], aspect oriented attribute grammars [de Moor et al., 2000b] or remote attributes [Boyland, 2005].

Research in attribute grammars has proceeded primarily in two directions. Firstly, there are AG-based systems such as `Silver` [Van Wyk et al., 2008], `JastAdd` [Ekman and Hedin, 2007], `LRC` [Kuiper and Saraiva, 1998] or `Eli` [Gray et al., 1992]. These systems followed the traditional approach of creating a stand-alone language, and they all have their own attribute evaluator and AG interpretation engines. Other systems, such as `UU-AG` [Swierstra et al., 2004] have a specific syntax but translate the code into a target language (in this case, `Haskell`), where computations are performed using its interpretors and compilers. They can be analyzed, reused and compiled independently.

Secondly, attribute grammars are embedded in regular programming languages with AG fragments as first-class values in the language. In the context of attribute grammars, this idea has already been explored [de Moor et al., 2000a; Sloane et al., 2010; Viera et al., 2009; Viera, 2013].

First class AGs provide a full component-based approach to AGs where a language is specified/implemented as a set of reusable off-the-shelf components. This means the implementations double as a typical program on the target language, which can be reused, compiled, refactored and be integrated in larger solutions.

Attribute grammars are a formalism with various notations. For example, `JastAdd` has a notation similar to `Java`, `LRC` uses `SSL` [Reps and Teitelbaum, 1989], `Silver` contains its own notations. All of these systems are themselves domain-specific languages, implemented and defined through embeddings and stand-alone mechanisms.

In this work, we will present attribute grammars as an embedded domain-specific language.

1.3 Embedding Attribute Grammars

In this work we will present a technique for embedding AGs on a functional setting. Our environment will support major AG extensions such as higher-order, references or circularity, which are unavailable in other functional AG embeddings.

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. Furthermore, with this option, 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.

Features such as parametric polymorphism, type inference, generalized algebraic data-types or pattern-matching, among many others, are extremely useful to the user but have to be designed and implemented, as was the case with `Silver` [Kaminski and Van Wyk, 2012]. With this technique, all these functional are available at no additional cost.

Together with the shallow embedding of attribute grammars in a functional setting, we will also present a practical application of our work with a deep embedded DSL. Throughout the next chapters, we will show how our shallow embedding can be used to implement various language-processing tasks.

We will present the embedding in the functional language `Haskell` [Jones et al., 1999] because its strong type system provides generic mechanisms that will be useful in the development of our embedding. Furthermore, this language is widely used and well known, and there are several good books [Hudak, 2000; Doets and van Eijck, 2004; O'Sullivan et al., 2008; Lipovaca, 2011; Mena, 2014] available for it.

We believe our approach provides a clear syntax which closely resembles the target domain. This means a domain-specialist programmer does not have to struggle to deal with the (potentially obscure) syntax of the functional target language.

We also tuned the syntax provided by our embedding to avoid having naive users invoking sophisticated and undesirable target language features without being aware of it, simplifying its usage. This is important because we have to use the target language interpreters and compilers, which means errors in our embedded DSL will relate to the target language (in this case, `Haskell`) and not to the DSL domain.

Our approach is favorable if we want to provide a DSL to a community which is already focused on a functional language that can be used as a host, as this option will provide them with extended and additional functionalities. It is also useful to users which want some of the functionalities of a functional setting but do not feel very comfortable with its syntax, as we provide a language-oriented DSL embedding.

Another advantage of our approach is that we gain for free the advanced characteristics of the target language. All of these features can be used together with the DSL we will embed, to create a powerful programming environment. It also means there is free access to development environments, compilers and there is a huge community available to help in development details and to continuously improve the existing features. We do not rely on the availability (which needs to be continuous) of the time and resources to to create a specific, stand-alone environment.

For the particular examples we will provide, which are written in `Haskell`, these advantages mean users have at their disposal features such as lazy evaluation, pattern matching, list comprehension, type classes or a strong type system, and environments such eclipse plug-ins to aid in development. These are specific advantages of the language we are using as target, and with our technique applied in other functional languages, some nonexistent features become available while others are lost.

Attribute grammars in our setting provide a method to implement complicated algorithms in a functional setting, but there are more alternatives to do so, as we will see in the next section.

1.4 Multiple Traversal Algorithms

As we mention in Section 1.2, AGs are a powerful formalism to express multiple traversal algorithms, like advanced pretty printing algorithms, type systems, programming environments, etc. In this thesis we will use a simple, but real example, to show the complex details involved when expressing such algorithms in a (functional) programming language. Let us consider the `LET` language, which represents the widely used let expressions in functional languages like `Haskell` [Jones et al., 1999], `ML` [Milner et al., 1997] or `Scala` [Odersky et al., 2008].

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.

Programs in this language consist of instruction blocks, where each instruction declares a variable, assigns some value to the variable, which can be constants or other variables, or defines a nested instruction block. A small example of a program in this language is:

$$
\begin{aligned}
\mathit{program} = \mathbf{let}\ & y = 4 \\
& a = \mathbf{let}\ w = 2 \\
& \quad\ \mathbf{in}\ \ x + y \\
\mathbf{in}\ \ & y + a
\end{aligned}
$$

In order to represent programs in the `LET` language, we define the following `Haskell` data-types:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Root} = \mathit{Root}\ \mathit{Let} \\
&\mathbf{data}\ \mathit{Let} = \mathit{Let}\ \mathit{Decls}\ \mathit{Expr} \\
&\mathbf{data}\ \mathit{Decls} = \mathit{Cons}\ \mathit{String}\ \mathit{Expr}\ \mathit{Decls} \\
&\qquad\qquad \mid \mathit{Empty} \\
&\qquad\qquad \mid \mathit{NestedLet}\ \mathit{String}\ \mathit{Let}\ \mathit{Decls} \\
&\mathbf{data}\ \mathit{Expr} = \mathit{Const}\ \ \mathit{Integer} \\
&\qquad\qquad \mid \mathit{Divide}\ \mathit{Expr}\ \mathit{Expr} \\
&\qquad\qquad \mid \mathit{Minus}\ \mathit{Expr}\ \mathit{Expr} \\
&\qquad\qquad \mid \mathit{Plus}\quad \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

```
  | Times Expr Expr
  | Var   String
```

In this representation, *program* is defined as:

```
program = Let (Cons "y" (Const 4)
              (NestedLet "a" (Let (Cons "w" (Const 2) (Empty))
                                  (Plus (Var "x") (Var "y")))
                 Empty)
              )
              (Plus (Var "y") (Var "a"))
```

We wish to construct a program to deal with the scope rules of the `LET` structured language. In a let expression an identifier may be declared at most once, and identifier declaration is necessary.

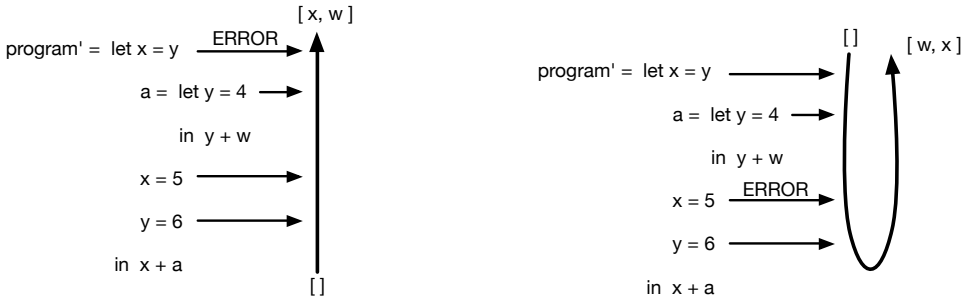
In nested expressions, using identifiers declared in an outer expression is allowed, and the definition of an identifier in a local scope hides the definition of the same identifier in a global one.

In *program* we saw a simple example with a single error: the invalid use of the not declared identifier x. Below, *program*$'$ illustrates a more complex situation where an inner declaration of y hides an outer one.

$$
\begin{array}{rl}
\textit{program}' = \textbf{let}\ x = y & \\
\qquad\qquad a = \textbf{let}\ y = 4 & \\
\qquad\qquad\quad \textbf{in}\ \ y + w & \\
\qquad\qquad x = 5 & \\
\qquad\qquad y = 6 & \\
\qquad \textbf{in}\ \ x + a &
\end{array}
$$

Programs such as *program* or *program*$'$ describe the basic block-structure found in many languages, with the peculiarity that no order is enforced about where identifiers can be declared and used. This means that declarations of identifiers may also occur after their first use.

According to the scope rules of the `LET` language, *program*$'$ contains two errors: a) at the outer block, the variable x has been declared twice, and b) the use of the variable w, at the inner block, has no binding occurrence at all.



(a) Bottom-up strategy. (b) Top-down strategy.

Figure 1.2: Bottom-up vs top-down strategy. Observe that only the second option detects duplicated declarations in the sequential order of the program.

We aim to develop a program that analyses `LET` programs and computes a list containing the identifiers which do not obey the scope rules. In order to make it easier to detect which identifiers are being incorrectly used in a program, we require that the list of invalid identifiers follows the sequential structure of the program. Thus, the semantic meaning of processing *program′* is $[w, x]$ (see Figure 1.2 where this result is shown).

Because we allow use before declaration, a conventional implementation of the required analysis leads to a program which traverses the abstract syntax tree twice: once to accumulate the declarations of identifiers and construct an environment, and again to check the uses of identifiers using the computed environment. The uniqueness of names is detected in the first traversal: for each newly encountered declaration we check whether the identifier has already been declared. In this case an error is computed.

An algorithm for processing this language has to be designed in two traversals:

1. On a first traversal, the algorithm has to collect the list of local definitions and, secondly, detect duplicate definitions from the collected ones. Because we want to detect duplicate declarations only in the moment they are declared twice, we have to follow a top-down strategy, as can be seen in Figure 1.2. A top-down strategy is usually implemented by accumulating parameters in a functional programming [Bird, 1998]. This would be implemented with the function:

$duplicate\_decls :: Let \to Env$

that takes a `LET` program and creates an environment.

2. On a second traversal, the algorithm has to use the list of definitions from the previous step as the global environment, detect the use of non-defined variables and finally combine the errors from both traversals. This would be implemented with the function:

   $missing\_decls :: Let \to Env \to Errors$

   that takes a program and an environment and returns a list of errors.

A straightforward solution to implement name analysis on `LET` would be as defined here by *semantics*:

$semantics :: Let \to Errors$
$semantics\ p = missing\_decls\ p\ (duplicate\_decls\ p)$

The problem with this solution is that the errors computed on the first traversal, with *duplicate_decls*, are never carried to the second traversal and will not appear in the final list of errors found.

To be able to compute the duplicated declarations of a block, the implementation also has to explicitly pass the errors detected between the two traversals of the program. As a consequence, a (intermediate) gluing data structure has to be defined to convey information computed in the first traversal to the second one. The need for explicitly defining and constructing intermediate data structures is not specific of a functional implementation, and in other programming paradigms such intermediate values are stored, as side effects, in the abstract syntax trees. In this case, the abstract syntax tree is the gluing data structure.

In all programming paradigms these gluing data structures make programs more complex to write, less concise and more difficult to be reused, but note that is not only the computation of the list of errors or the additional data types that require additional work. The scheduling of the two traversals is not straightforward either.

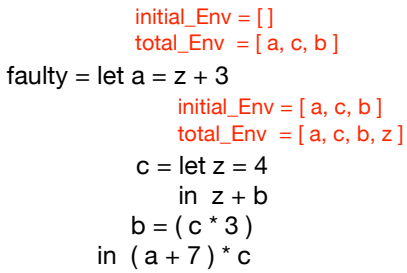


Figure 1.3: The value of the environment in different parts of a program.

As we can see in Figure 1.3, the initial environment of the nested block is the total environment of its parent. This means that an outer block of a program has to be completely traversed before starting the calculation of the environment of an inner block. This is one characteristic of the language that makes defining one algorithm for name analysis harder to implement.

Thus, the traversals of outer and inner let expressions are intermingled, making it very complex to determine how to schedule the different traversals in larger languages. An example of a very complex algorithms that schedules a four traversal pretty printing algorithm is presented in [Swierstra et al., 1999].

In the next section we present the full implementation of an strict program, written in `Haskell`, showing these problems. After that we present a solution in a lazy programming paradigm where no gluing data structure has to be defined. However, such solution has other disadvantages that we will discuss in detail later.

1.4.1 Strict Algorithms

One solution to implement name analysis on an abstract tree of `LET` is to do so through a strict program. The idea is to use a set of functions to perform one traverse on the tree, capturing all the information necessary and storing it in an intermediate format. Afterwards, a second set of functions uses this intermediate "gluing" data type and performs a second traverse in order to completely compute the list of errors.

In this implementation, a "gluing" data structure, of type Let_2, has to

be defined by the programmer and is constructed to pass the detected errors explicitly from the first to the second traversal, in order to compute the final list of errors in the desired order. To be able to compute the missing declarations of a block, the implementation also has to explicitly pass the environment a block between the two traversals of the block. This information must therefore also be in the Let_2 intermediate structure:

$$\begin{aligned}
&\textbf{data}\ Let_2 \quad = Let_2\ Decls_2\ Expr \\
&\textbf{data}\ Decls_2 = Cons_2 \qquad Error\ Expr\ Decls_2 \\
&\qquad\qquad\quad \mid NestedLet_2\ Error\ Lev \quad Let\ Decls_2 \\
&\qquad\qquad\quad \mid Empty_2
\end{aligned}$$

The data constructor $NestedLet_2$ has to carry the errors (*Error*) and the level (*Lev*) between traversals. We have already seen why the errors have to be carried from the first to the second traversal. The value of the level must also be carried around because it is on the second traversal that the first traversal on nested expressions starts (recall Figure 1.3, where it is visible that the initial environment of an inner block is composed by the total environment of an outer one).

For every block we compute three things: its environment, its lexical level and its invalid identifiers. The environment defines the context where the block occurs. It consists of all the identifiers that are visible in the block. The lexical level indicates the nesting level of a block. Observe that we have to distinguish between the same identifier declared at different levels, which is a valid declaration (for example, y in *program′*), and the same identifier declared at the same level, which is an invalid declaration (for example, x in *program′*). Finally, we have to compute the list of identifiers that are incorrectly used, i.e., the list of errors.

We will need auxiliary functions that check the rule that a variable must not be in (*mNBIn*) the environment (to check for duplicated declarations) and the rule that a variable must be in (*mBIn*) the environment (to check for the usage of undeclared identifiers). These two functions can be easily defined in `Haskell` as:

$$mBIn :: String \rightarrow Env \rightarrow Error$$

$$
\begin{array}{l}
mBIn\ name\ e = \\
\textbf{case}\ e\ \textbf{of} \\
\quad [\,] \rightarrow [name] \\
\quad ((Tuple\ n\ l) : es) \rightarrow \textbf{if}\ (n \equiv name)\ \textbf{then}\ [\,] \\
\qquad\qquad\qquad\qquad \textbf{else}\ (mBIn\ name\ es)
\end{array}
$$

$$
\begin{array}{l}
mNBIn :: String \rightarrow Int \rightarrow Env \rightarrow Error \\
mNBIn\ name\ lev\ e = \\
\textbf{case}\ e\ \textbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad ((Tuple\ n\ l) : es) \rightarrow \textbf{if}\ ((n \equiv name) \wedge (l \equiv lev)) \\
\qquad\qquad\qquad\qquad \textbf{then}\ [n] \\
\qquad\qquad\qquad\qquad \textbf{else}\ (mNBIn\ name\ lev\ es)
\end{array}
$$

Next, we present scope analysis for `LET` defined with a strict strategy in `Haskell`:

```
  -- Scheduling the two traversals while also starting
  -- the intermediate structure
semantics_Strict :: Root → Error
semantics_Strict (Root program) = errors
  where
    (let_2, dclo) = duplicateDecls_Let program [] 0
    errors        = missingDecls_Let let_2 dclo

duplicateDecls_Let :: Let → Dcli → Lev → (Let_2, Dclo)
duplicateDecls_Let (Let decls expr) dcli lev = (Let_2 decls_2 expr, dclo)
  where
    (decls_2, dclo) = duplicateDecls_Decls decls dcli lev

  -- Constructing the intermediate structure while checking
  -- for duplicated declarations
duplicateDecls_Decls :: Decls → Dcli → Lev → (Decls_2, Dclo)
duplicateDecls_Decls (Cons name expr decls) dcli lev =
  (Cons_2 error expr decls_2, dclo)
  where
    dcli_2 = (Tuple name lev) : dcli
    error          = mNBIn name lev dcli
```

$$
\begin{aligned}
&\quad (decls_2, dclo) = duplicateDecls_{Decls}\ decls\ dcli_2\ lev \\
&\quad \text{-- Scheduling the traversal on nested expressions} \\
&duplicateDecls_{Decls}\ (NestedLet\ name\ nested\_let\ decls)\ dcli\ lev = \\
&\quad (NestedLet_2\ error\ lev_2\ nested\_let\ decls_2, dclo) \\
&\quad \textbf{where} \\
&\qquad lev_2 \qquad\quad = lev + 1 \\
&\qquad dcli_2 \qquad\quad = (Tuple\ name\ lev) : dcli \\
&\qquad error \qquad\quad = mNBIn\ name\ lev\ dcli \\
&\qquad (decls_2, dclo) = duplicateDecls_{Decls}\ decls\ dcli_2\ lev \\
&duplicateDecls_{Decls}\ Empty\ dcli\ lev = (Empty_2, dcli) \\
&missingDecls_{Let} :: Let_2 \to Env \to Error \\
&missingDecls_{Let}\ (Let_2\ decls\ expr)\ env = errors +\!\!+ errors_2 \\
&\quad \textbf{where} \\
&\qquad errors \;\; = missingDecls_{Decls}\ decls\ env \\
&\qquad errors_2 = missingDecls_{Expr}\ expr\ env \\
&missingDecls_{Decls} :: Decls_2 \to Env \to Error \\
&missingDecls_{Decls}\ (Cons_2\ error\ expr\ decls)\ env = errors \\
&\quad \textbf{where} \\
&\qquad errors = error +\!\!+ (missingDecls_{Expr}\ expr\ env) \\
&\qquad\qquad\quad +\!\!+ (missingDecls_{Decls}\ decls\ env) \\
&\quad \text{-- Scheduling the two traversals for the nested expression} \\
&missingDecls_{Decls}\ (NestedLet_2\ error\ lev\ nested\_let\ decls)\ env = errors \\
&\quad \textbf{where} \\
&\qquad (nestedlet_2, dclo) = duplicateDecls_{Let}\ nested\_let\ env\ lev \\
&\qquad errors \qquad\qquad = error +\!\!+ (missingDecls_{Let}\ nestedlet_2\ dclo) \\
&\qquad\qquad\qquad\qquad\quad +\!\!+ (missingDecls_{Decls}\ decls\ env) \\
&missingDecls_{Decls}\ Empty_2\ env = [\,] \\
&missingDecls_{Expr} :: Expr \to Env \to Error \\
&missingDecls_{Expr}\ (Const\ \_)\ env = [\,] \\
&missingDecls_{Expr}\ (Divide\ expr_1\ expr_2)\ env = errors \\
&\quad \textbf{where} \\
&\qquad errors = (missingDecls_{Expr}\ expr_1\ env) \\
&\qquad\qquad\quad +\!\!+ (missingDecls_{Expr}\ expr_2\ env) \\
&missingDecls_{Expr}\ (Minus\ expr_1\ expr_2)\ env = errors
\end{aligned}
$$

```
    where
      errors = (missingDecls_Expr expr_1 env)
               ++ (missingDecls_Expr expr_2 env)
missingDecls_Expr (Plus expr_1 expr_2) env = errors
    where
      errors = (missingDecls_Expr expr_1 env)
               ++ (missingDecls_Expr expr_2 env)
missingDecls_Expr (Times expr_1 expr_2) env = errors
    where
      errors = (missingDecls_Expr expr_1 env)
               ++ (missingDecls_Expr expr_2 env)
missingDecls_Expr (Var name) env = (errors)
    where
      errors = mBIn name env
```

Note that in $semantics_{Strict}$ the function call $duplicateDecls_{Let}$ not only computes the total environment (using an accumulating parameters that starts empty), but it also computes the gluing data structure that stores the duplicated errors detected during the first traversal. The second traversal starts with a call to $missingDecls_{Let}$ giving that computed gluing data structure and the accumulated environment as arguments. It returns the list of errors that follows the sequential structure of the program.

Please note also that in the second traversal to a nested let expression, in the function $missingDecls_{Decls}$ for the constructor $NestedLet_2$, the program performs the two traversals to the body of that let: calls $duplicateDecls_{Let}$ and $missingDecls_{Let}$.

The use of this strict strategy that we have just seen has the following advantages:

- Strict solutions are typically composed of smaller functions, which means they are *"modular"*. However, this advantage may be lost when there is the necessity of having intermediate structures, because since all the functions need this intermediate structure any change means all of them have to be adapted;

- This solution is easily incrementalized via standard (strict) functional memoization [Saraiva and Swierstra, 1999a].

Despite being modular and easy to incrementalize, strict programs have the following disadvantages:

- It is not easy to schedule the different traversals and write such intermingled recursive functions, as this small example has showed;
- The programmer has to concern himself in defining gluing data structures. In functional programming this is done using additional data types, as showed before. In other programming paradigms this is usually done via side effects: by storing such values in `LET` abstract representation. As a result, this data type has to be modified to handle such side effects.

Next, we will see an approach to implement name analysis on `LET` that is based on circular, lazy programming.

1.4.2 Lazy Algorithms

Another approach to implement name analysis on `LET` is through circular programs. Contrary to the strict implementation we have presented in the previous section, this solution does not require two traversals, functions scheduling or intermediate data types.

The main characteristic of circular programs if that they have what appears to be a circular definition: arguments in a function call depend on results of that same call:

$$(\ldots, x, \ldots) = f \ldots x \ldots$$

Next, we will present a circular strategy for solving name analysis on `LET`, in the programming language `Haskell`:

$$semantics_{Lazy} :: Root \rightarrow Error$$
$$semantics_{Lazy}\ (Root\ program) = errors$$

where

$(dclo, errors) = dup\_and\_miss_{Let}\ program\ [\,]\ dclo\ 0$

$dup\_and\_miss_{Let} :: Let \to Dcli \to Env \to Lev \to (Dclo, Error)$

$dup\_and\_miss_{Let}\ (Let\ decls\ expr)\ dcli\ env\ lev = (dclo, errors +\!\!+ errors_2)$

where

$(dclo, errors) = dup\_and\_miss_{Decls}\ decls\ dcli\ dclo\ lev$

$errors_2 = missing_{Expr}\ expr\ dclo$

$dup\_and\_miss_{Decls} :: Decls \to Dcli \to Env \to Lev \to (Dclo, Error)$

$dup\_and\_miss_{Decls}\ (Cons\ str\ expr\ decls)\ dcli\ env\ lev = (dclo, errors)$

where

$dcli_2 = (Tuple\ str\ lev) : dcli$

$(dclo, errors_3) = dup\_and\_miss_{Decls}\ decls\ dcli_2\ dclo\ lev$

$errors_2 = missing_{Expr}\ expr\ dclo$

$errors = (mNBIn\ str\ lev\ dcli) +\!\!+ errors_2 +\!\!+ errors_3$

$dup\_and\_miss_{Decls}\ (Empty)\ dcli\ env\ lev = (dcli, [\,])$

$dup\_and\_miss_{Decls}\ (NestedLet\ str\ let_1\ decls)\ dcli\ env\ lev = (dclo, errors)$

where

$lev_2 = lev + 1$

$dcli_3 = (Tuple\ str\ lev) : dcli$

$(dclo_2, errors_2) = dup\_and\_miss_{Let}\ let_1\ dclo\ dclo_2\ lev_2$

$(dclo, errors_3) = dup\_and\_miss_{Decls}\ decls\ dcli_3\ dclo\ lev_2$

$errors = (mNBIn\ str\ lev\ dcli) +\!\!+ errors_2 +\!\!+ errors_3$

$missing_{Expr} :: Expr \to Env \to Error$

$missing_{Expr}\ (Const\ \_)\ \_ = [\,]$

$missing_{Expr}\ (Divide\ expr_1\ expr_2)\ env = errors_1 +\!\!+ errors_2$

where

$errors_1 = missing_{Expr}\ expr_1\ env$

$errors_2 = missing_{Expr}\ expr_2\ env$

$missing_{Expr}\ (Minus\ expr_1\ expr_2)\ env = errors_1 +\!\!+ errors_2$

where

$errors_1 = missing_{Expr}\ expr_1\ env$

$errors_2 = missing_{Expr}\ expr_2\ env$

$missing_{Expr}\ (Plus\ expr_1\ expr_2)\ env = errors_1 +\!\!+ errors_2$

where

$$
\begin{aligned}
&\quad errors_1 = missing_{Expr}\ expr_1\ env \\
&\quad errors_2 = missing_{Expr}\ expr_2\ env \\
&missing_{Expr}\ (Times\ expr_1\ expr_2)\ env = errors_1 +\!\!+ errors_2 \\
&\quad \textbf{where} \\
&\quad errors_1 = missing_{Expr}\ expr_1\ env \\
&\quad errors_2 = missing_{Expr}\ expr_2\ env \\
&missing_{Expr}\ (Var\ str)\ env = errors \\
&\quad \textbf{where} \\
&\quad errors = mBIn\ str\ env
\end{aligned}
$$

An example of a circular definition is in the argument *dclo* in the function $dup\_and\_miss_{Decls}$, where it is both an argument and the return value of the function. The circular nature of this definition means that the programmer can continue to define computations and the lazy nature of the engine will be able to select which values can be computed at each given time in the program chain, and be capable of producing a final result.

Using circular programming has some advantages comparing to the strict strategy we have seen in the previous section:

- This solution needs only one tree traversal, so no function scheduling is necessary;
- No intermediate structures are required. This means no extra work is required for creating and maintaining additional data structures.

Despite not requiring intermediate structures or multiple traversals, circular programs have some disadvantages:

- As can be seen, it is hard and "non-natural" to write such circular programs, and even for an advanced lazy functional programmer it is hard to write a program which is not completely circular, i.e., which terminates;
- circular programs do not provide modularity: if new functionality has to be added all the functions have to be modified, which is usually done by adding more arguments and results to the existing functions. For

example, if we wish to compute the final result of a let expression, then we need to add an additional result to all functions and consequently to update their calls. Thus, a major update would be necessary;

- Lazyness is required, which is known as being more inefficient than strict approaches [Fernandes et al., 2011], and requires a language that supports lazy evaluation.

In this thesis we will present a functional setting to implement these traversal algorithms, through the use of attribute grammars. Our setting does not require extra effort to implement functions scheduling or intermediate data types as we saw on the strict programs, but it is also more capable of coping with changes on data types or on the analysis and does not require lazy mechanisms, as circular programs do. Our solution is also modular, and can easily cope with changes both in the language and in the semantics we want to implement.

1.5 Bidirectional Attribute Grammars

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

Another application is in semantic editors generated by AGs [Kuiper and Saraiva, 1998; Reps and Teitelbaum, 1989; Söderberg, 2012]. 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

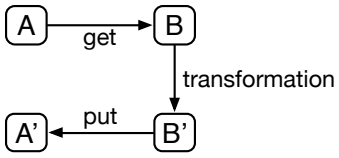


Figure 1.4: A bidirectional transformation system.

be manually synchronized when one of the transformations changes. This makes maintenance complex and error prone. In this work we will also leverage this limitation of AG by providing mechanism that make our embedding environment supporting bidirectional transformations.

A bidirectional transformation (BX) is a program which expresses 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.

For example, in a transformation $A \rightarrow B$, a bidirectional 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. This can be see in Figure 1.4. Here, a manually written get (the forward transformation) creates a new type B, which suffers a transformation into B'. This B' can be automatically transformed, via the put (backward) transformation into a new instance of type A, A', without user intervention or any kind of additional implementation.

Where a traditional approach would mean implementing both transformations manually, which is expensive, error-prone and creates obvious maintenance problems, a bidirectional system automatically derives a transformation in one direction.

It is common in bidirectional systems for the automatically generated backward transformation to have a notion of the original data type that generated the input of the transformation. Returning to Figure 1.4, this would mean that the backward transformation would have as input B', which the function has to transform into a new A', has information regarding the original A. This aids in the transformation because it helps achieving an A' as closer as possible to A, which is desirable.

In the context of grammars, a BX 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. Here, of special interest are tree-based structures such as the ones generated by concrete and abstract grammars, as seen in the previous sections. The problem with these transformations is that both the forward and the backward transformations need to be implemented by hand.

Bidirectional data transformations have been studied in different computing disciplines, such as updatability views in relational databases [Bohannon et al., 2006], programmable structure editors [Hu et al., 2004] or model-driven development in software engineering [Stevens, 2008]. In [Czarnecki et al., 2009] a detailed discussion and extensive citations on bidirectional transformations are included.

1.6 Overview

In this thesis we propose a concise embedding of AGs in `Haskell`. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [Huet, 1997] 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.

Zippers do not rely on any advanced feature of `Haskell` such as lazy evaluation or type classes. Thus, a zipper-based embedding of attribute grammars can be straightforwardly re-used in any other functional environment. Our embedding is also extended with the main modern AG extensions proposed to the AG formalism.

We present an embedding attribute grammars with modern extensions as first class attribute grammars together with a bidirectional system. By this we are able to express powerful algorithms as the composition of AG reusable components. We have used this approach in a number of applications, e.g., in developing techniques for a language processor to implement bidirectional

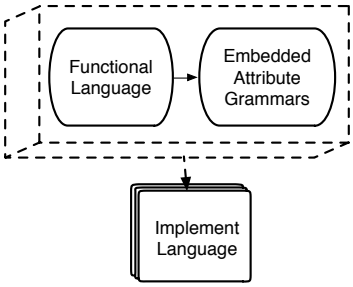


Figure 1.5: We will embed AGs in `Haskell`, which will provide an environment to define and implement languages.

AG specifications and to construct a software portal.

Because AGs provide themselves syntax and semantics for programming languages, we are creating a setting where we embed a DSL which can itself be used to define and implement any programming language. This idea can be seen in Figure 1.5.

Throughout this work, we will define AGs and extend them with modern extensions, always using real-world problems. We will do so through a small programming language that has characteristics and presents challenges as bigger, real programming languages do. This language, to which we call `LET`, provides the usual *let - in* declare/use construction found in functional programming languages.

The problems we will present and their respective solutions will, as a whole, define a small interpreter and compiler for `LET`: we will perform name analysis, extend the language, transform it into different representations and provide results for it, always using our technique for embedding AGs.

1.6.1 Main Publications

During this thesis, we have published a number of articles that describe the work presented in this document:

- Martins, P. (2012). Zipper-based Embedding of Modern Attribute Grammar Extensions. *Doctoral Symposium of the 5th International Conference on Software Language Engineering.*

- Martins, P., Fernandes, J. P., and Saraiva, J. (2012). A Purely Functional Combinator Language for Software Quality Assessment. In *Proceedings of the Symposium on Languages, Applications and Technologies*, volume 21 of *SLATE '12*, pages 51–69. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

- Martins, P., Fernandes, J. P., and Saraiva, J. (2014). A Web Portal for the Certification of Open Source Software. In *Proceedings of the 6th International Workshop on Foundations and Techniques for Open Source Software Certification*, volume 7991 of *OPENCERT '12*, pages 244–260. Springer-Verlag.

- Martins, P., Carvalho, N., Fernandes, J. P., Almeida, J. J., and Saraiva, J. (2013). A Framework for Modular and Customizable Software Analysis. In *Proceedings of the 13th International Conference on Computational Science and Its Applications*, volume 7972 of *ICCSA '13*, pages 443–458. Springer-Verlag.

- Martins, P., Fernandes, J. P., and Saraiva, J. (2013). Zipper-Based Attribute Grammars and Their Extensions. In *Proceedings of the 17th Brazilian Symposium on Programming Languages*, volume 8129 of *SBLP '13*, pages 135–149. Springer-Verlag.

- Martins, P., Saraiva, J., Fernandes, J. P., and Wyk, E. V. (2014). Generating Attribute Grammar-based Bidirectional Transformations from Rewrite Rules. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, PEPM '14*, pages 63–70. Association for Computing Machinery (ACM).

- Martins, P. and Carção, T. (2014). A Visual DSL for the Certification of Open Source Software. In *Proceedings of the 14th International Conference on Computational Science and Its Applications, ICCSA'14* (to appear).

1.6.2 Software Prototypes

We have create a package in `Hackage`: an on line repository made out of open-source libraries and tools, heavily used by the `Haskell` community. We have created the package `ZipperAG`, which can be accessed in:

`https://hackage.haskell.org/package/ZipperAG`

Here, the reader can find the implementations presented in this thesis together with more examples of zipper-based, AG implementations.

We have also created a prototype for our bidirectional system, which can be accessed in the author's web page, in:

`http://www.di.uminho.pt/~prmartins`

The web portal where our process management DSL was deployed can be accessed in:

`http://cross.di.uminho.pt`

1.6.3 Other Publications

Besides exploring research that is fundamental to the core of this thesis, we also had the opportunity to contribute to related scientific areas. These contributions have resulted in the following publications:

- Martins, P., Lopes, P., Fernandes, J. P., Saraiva, J., and Cardoso, J. M. P. (2012). Program and Aspect Metrics for MATLAB. In *Proceedings of the 12th International Conference on Computational Science and Its Applications*, *ICCSA '12*, pages 217–233. Springer-Verlag.

- Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., and Saraiva, J. (2012). SmellSheet Detective: A tool for detecting bad smells in spreadsheets. In *Proceedings of the IEEE Forum on Visual Languages*

and Human-Centric Computing, *VL/HCC '12*, pages 243–244. Institute of Electrical and Electronics Engineers (IEEE).

- Pedro Martins and Rui Pereira (2014). Refactoring Smelly Spreadsheet Models. In *Proceedings of the 14th International Conference on Computational Science and Its Applications*, *ICCSA'14* (to appear).

- Cunha, J., Fernandes, J. P., Martins, P., Pereira, R., and Saraiva, J. (2014). Refactoring meets Model-Driven Spreadsheet Evolution. In *Proceedings of the 9th International Conference on Quality in Model Driven Engineering*, *QUATIC '14* (to appear).

- Martins, P., Abreu, R., Perez, A., Cunha, J., Fernandes, J. P., and Saraiva, J. (2014). Smelling Faults in Spreadsheets. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, *ICSME '14* (to appear).

1.7 Structure of the Thesis

This thesis is structured as follow: In Chapter 2 we provide a series of definitions and notations that will be important to understand the work presented in this document. Here we also present instances and examples of the formalisms we define. We also introduce the language we will use as a running example throughout this work, and define its various grammar representations.

In Chapter 3 we introduce our embedding of AGs. We start by presenting the concept of functional zippers through simple examples. This is an important technique on which this work is based. We also define the language presented in the previous Chapter 2 in our environment.

In Chapters 4, 5 and 6 we present how different extensions are implemented in our environment. In particular, we present reference attribute grammars, higher order attribute grammars and circular attribute grammars, respectively. We always present examples of how different language-oriented tasks can be implemented with these extensions. All the different extensions

are introduced through sample code on different AG tools, and afterwards we present the solution in our setting.

On Chapter 7 we show that our environment allows the combination of different extensions. In particular, we show how a complex language-solving task can be implemented with the combination of higher order and circularity, and how these two integrate nicely.

In this work we also designed bidirectional techniques. In particular, we introduce a formalism for designing and implementing transformation specifications. From these, we defined rules for validating these specifications, and developed automatic techniques for expansion and inversion of the transformations. We also explain how AG equations can be derived from the transformation specifications, creating an environment that implements our bidirectional system. This is presented in Chapter 8.

We have applied the technique described in this work in real-world problems. We designed a DSL for process management whose underground machinery is based on our zipper-based AG environment. We then implanted this DSL on a web portal for customizable software analysis. This is presented in Chapter 9.

Finally, in Chapter 10 we present our conclusions for this work, directions for future paths of research and our final remarks.

Chapter 2

Definitions and Notations

Summary

In this chapter we introduce the theoretical background necessary to understand the rest of this work, through formalisms and notations that represent the main concepts required. We also provide examples of instances of these formalisms in the language LET, also defined here.

2.1 Introduction

In this chapter we will introduce the theoretical background necessary to understand the rest of this work. We do so through the definition of formalisms and notations for the entities we will reason about. We also present concrete examples that instantiate them.

We start by defining context-free grammars (CFGs), important when formalizing the syntactic characteristics of a language. A CFG is composed by a set of rules describing how to form textual representations from a language alphabet, which are valid according to its syntax. A CFG does not describe the semantics (the meaning) of this textual representations, only their form and structure.

An attribute grammar (AG) is another formalism that is defined in this chapter. AGs are a formal method to define the semantics of a language.

Through the description of semantic functions (called attributes) for each production of a grammar, with an AG we can associate values to the language, describing its meaning.

Both for context-free grammars and for attribute grammars, we present examples of specifications of the syntax and the semantics of a programming language which is also introduced. For context-free grammars, we also present specifications of the various forms on which a language can be syntactically specified, namely its concrete and abstract representation.

We make use of `Haskell`, a functional programming language that we use to define certain semantic operations. We do so because this is the target language of the work presented in this thesis and because it is a concise and elegant notation on which semantics can be defined.

2.2 Context-free Grammars

Context-free grammars describe the structure and syntax of a programming language by providing a technique that details them.

Definition 1. (Context-free Grammar) A context-free grammar (CFG) is a 4-tuple $G = \langle V, \Sigma, P, S \rangle$ where:

- V is the non-empty, finite set of non-terminal symbols or non-terminals. Each $v \in V$ represents a different type of phrase, and defines a sub-language of the language defined by the grammar G;
- Σ is the non-empty set of terminal symbols or terminals;
- P if the finite relation $V \to (V \cup \Sigma)^*$ ($*$ represents the Kleene star), called the rewrite rules or the productions of the grammar;
- $S \in V$ in the start symbol of the grammar.

A context-free grammar is therefore composed by a set of non-terminals, a set of terminals, a set of productions and a starting symbol.

Definition 2. (Production) A production $p \in P$ is denoted by $p : X_0 \to X_1 \dots X_n$ where:

- $X_0 \in V$ is a non-terminal, called the left-hand side of p or simply lhs, denoted $lhs(p)$;

- $X_1 \dots X_n$ are a set of terminal and non-terminal symbols, called the right-hand side of p, or simply rhs, denoted $rhs(p)$.

The total number of right-hand side symbols of a production p is defined as $\mid p \mid$. A pair $\langle p, i \rangle$ is called an occurrence of the grammar symbol $X_i \in (V \cup \Sigma)$, with $0 \leqslant i \leqslant n$. A production is applied on X if and only if $\langle p, 0 \rangle = X$, and we say that a production p is a terminal production if it has no non-terminal symbols on its right-hand side, i.e., $\forall X \in rhs(p)$, $X \in \Sigma$.

An empty right-hand side on a production is represented with the symbol ϵ, and a production with only one grammar symbol, i.e., a production with the form $A \to \epsilon$ is called an ϵ-production. It is common when defining context-free grammars to list all right-hand sides of productions with the same left-hand side using the symbol $\mid$. For example, the productions $\alpha \to \beta_1$ and $\alpha \to \beta_2$ can be written as $\alpha \to \beta_1 \mid \beta_2$.

On a context-free grammar, the left-hand side non-terminal can be rewritten to its right-hand side. Therefore, we define the relation $\Rightarrow$, called directly derives, as follows:

Definition 3. (Directly derives) For two strings $u, v \in (V \cup \Sigma)^*$, we say u yields v, written as $u \Rightarrow v$, if $\exists(\alpha \to \beta) \in P$, with $\alpha \in V$ and $u_1, u_2 \in (V \cup \Sigma)^*$ such that $u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$. This means that v is the result of applying $\alpha \to \beta$ to u.

The operation $\Rightarrow$ possesses transitive and reflexive closures, which are defined as usual, and denoted by $\Rightarrow^+$ and $\Rightarrow^*$ respectively.

Definition 4. (Context-free Language) The language of a grammar $G = \langle V, \Sigma, P, S \rangle$ is the set $\mathcal{L}(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$. A language $\mathcal{L}$, is said to be a context-free language CFL, if there exists a CFG_G, such that $\mathcal{L} = \mathcal{L}(G)$.

A language is therefore the set of sequences of terminal symbols that can be derived by rewriting the start symbol S.

The grammar G generates a sequence s if and only if $s \in \mathcal{L}(G)$. We say that a symbol $X \in (V \cup \Sigma)$ is accessible or derivable from $Y \in V$ if there is a derivation of the form $Y \Rightarrow^* \alpha_1 X \alpha_2 \in (V \cup \Sigma)^*$.

We say the grammar is unambiguous if one and only one sequence of derivations exists for every $s \in \mathcal{L}(G)$, otherwise we call it ambiguous. Two context-free grammars G_1 and G_2 are equivalent if and only if $\mathcal{L}(G_1) = \mathcal{L}(G_2)$, and are denoted as $G_1 \equiv G_2$.

Definition 5. (Complete Context-free Grammar) A context-free grammar $G = \langle V, \Sigma, P, S \rangle$ is said to be a complete context-free grammar if and only if $\forall\ X \in (V \cup \Sigma), \exists\ \mu, \nu \in (V \cup \Sigma)^* \wedge \delta\ \in\ \Sigma^*\ : S \Rightarrow^* \mu X \nu \Rightarrow^* \delta$.

A context-free grammar is therefore said to be complete if every symbol is accessible from the start symbol and every non-terminal can derive a sequence of only terminal symbols.

2.2.1 Concrete and Abstract Grammars

Context-free grammars, as we have seen them in the previous section, specify syntactic characteristics of languages. However, there is an important distinction between two classes of context-free grammars: concrete context-free grammars and abstract context-free grammars.

A concrete context-free grammar G, which is also called a concrete grammar defines how the vocabulary symbols $(V \cup \Sigma)$ can form a syntactic valid sentence of $\mathcal{L}(G)$. A concrete grammar has the aim of allowing an easy derivation of sentences of the language under consideration.

Grammars do not have the singular objective of formally defining a language, they also guide a specific type of programs, called parsers, that check if a stream of symbols is or is not valid according to the grammar. Therefore, a concrete grammar defines sentences of a language precisely as the user would write, with all the necessary syntactic symbols and with all the specific language keywords.

Parsers usually perform other semantic actions while checking the stream of symbols. One of these actions is to construct a representation of the

sentence being read, which is usually a tree, and can be the final result of the parser or an intermediate data-type used for further processing. An unambiguous concrete context-free grammar G defines a unique concrete syntax tree for every sentence of $\mathcal{L}(G)$.

Definition 6. (Concrete Syntax Tree) A concrete syntax tree (CST), also called derivation tree or parse tree, generated by a complete context-free grammar $G = \langle V, \Sigma, P, S \rangle$, for a sentence s, is defined as:

- Each node is labeled by a symbol $X \in (V \cup \Sigma \cup \epsilon)$;
- If a node labeled X_0 has children labeled $X_1, X_2, \ldots, X_n$, then there is a production $p : X_0 \rightarrow X_1, X_2, \ldots, X_n$;
- The label of the root of the tree is S;
- The concatenation of all the leaves of the tree, from left to right, form the sentence s.

There are various parsing techniques that can be used by a parser, always with the aim of understanding how a sentence can be derived starting on the starting symbol of a grammar. LL parsers, for example, use a recursive-descent technique and are examples of top-down parsers, i.e., they start with the starting symbol S and try to find how sentences match the right-hand side of productions. LR parsers, on the other hand, are examples of bottom-up parsers that start with the input and attempt to rewrite it to the start symbol. The parsing technique may impose restrictions on the concrete context-free grammar. For example, parsing techniques such as LL(1) and LR(1) require the grammar to be in specific forms [Aho et al., 2006]. This means that different concrete context-free grammars can be used to define the exact same language.

We have seen how a concrete grammar completely describes a language, usually with a huge set of symbols that precisely describe its syntax. After this is done, and after we know the syntax is correct, we want to go to the next step and structurally analyze the language, i.e., analyze the logic

behind the language sentences and how they connect to each others to form a computer program.

In order to analyze and describe the structure of a language, the concrete syntax and parsing characteristics of the language are abstracted as only the structure is relevant. Thus, an abstract context-free grammar, or simply abstract grammar describes precisely the abstract structure of a sentence, without unnecessary symbols. In an abstract grammar we loose the syntax characteristics of the language, but this is not a problem since this step is always followed by such analysis.

One important note is that it is not correct, by modern standards, to split an analysis of a language just between syntax and structure analysis. Modern parsers are composed out of multiple steps that include tokenization, macro expansion, handling pre-processor directives, among many others, usually with intermediate data-types between. We do not focus on parsing in this work, but more information on the subject can be found, for example, in [Fernandes, 2004; Stallman and Community, 2009; van den Brand et al., 2002; Aho et al., 2006; Terry, 2005].

Definition 7. (Abstract Syntax Tree) A abstract syntax tree (AST), generated by a complete context-free grammar $G = \langle V, \Sigma, P, S \rangle$, for a sentence s, is defined as:

- Each node is labelled by a production $p \in P$;
- Every node labelled by a ϵ-production is a leaf;
- Every node labelled by p, with $X_0 = pX_1 \ldots X_n$, has n children $T1 \ldots T_n$, where each T_i, with $0 \leqslant i \leqslant n$, is again an abstract syntax tree labeled with a production applied on X_i.

In context-free grammars we distinguish two different classes of terminal symbols $\Sigma = L \cup \Gamma$. L is the set of literal symbols, which consists of the symbols of the alphabet that do not play a role in the semantics of the language. Examples include keywords or punctuation symbols. Γ is the set of pseudo-terminal symbols, which are non-terminal symbols for which

productions are implicit. These are typically expressed as regular expressions [Aho et al., 2006], provided by an external lexical analyzer. Examples include integers, strings, vectors and other identifiers of the language.

Non-terminal and pseudo-terminal symbols induce a set of terms, which will from now on be regarded as types. Therefore, we define the function $\mathcal{T}_G :: (V \cup \Gamma) \rightarrow T$, where $\forall X \in (V \cup \Gamma)$, T is the set of types, and $(\mathcal{T}_G X) \in T$ is the set of all possible values of X.

2.3 Context-free Grammar Specification

Below is an example of a program in the `LET` language, which corresponds to correct `Haskell` code (to simplify our initial example, we do not consider nested *let* sub-expressions, but this extension to `LET` will be considered later in this thesis).

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

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.

The concrete representation of `LET` can be easily described by the grammar: $G_{\texttt{LET}_{Concrete}} = \langle V, \Sigma, P, S \rangle$. As usual in context-free grammars, we present the set of productions P only, and we use the standard Backus Naur Formalism (BNF):

```
(p1: Start)     Start → Main
(p2: Main)      Main  → 'Let' Declarations 'In' E
(p3: ConsDecl)  Declarations → Name '=' E Declarations
(p4: NoDecl)    Declarations → ϵ
(p5: Add)       E → E '+' T
```

```
(p6:  Take)      E → E '-' T
(p7:  Et)        E → T
(p8:  Prod)      T → T '*' F
(p9:  Div)       E → T '/' F
(P10: Tf)        T → F
(p11: Nest       F → '(' E ')'
(P12: Neg)       F → '-' F
(p13: Constant)  F → Number
(p13: Variable)  F → Name
```

From this set of productions P, all of the components of $G_{\texttt{LET}_{Concrete}}$ can be inferred: V is the set of all the symbols on the left-hand side, Σ is the set with all the symbols that only appear on the right-hand side, and S is the left-hand side symbol of the first production.

The concrete representation of `LET` defines a language starting with the keyword `'Let'`, immediately followed by `Declarations`, by the keyword `'In'` and by an expression `E`. As this is a concrete representation, we have to write all the terminal symbols that constitute literal symbols, i.e., that are not structurally important for the language but are important for parsing sentences of `LET` exactly as a user would write it. Examples include parenthesis, punctuation or symbols for arithmetic operations.

On the other hand, the terminal symbols `Name` and `Number` are pseudo-terminal symbols. They represent *strings* (for variable names) and *integers*. We assume these are externally provided by a lexical analyzer.

One important note on the definition of this grammar is the non-terminal `E`. This terminal, which describes an expression, follows a well-know technique for parsing expressions called *expression-term-factor*. This splits an expression into three non-terminals, `E`, `T` and `F`, important for avoiding ambiguities in the way the grammar describes the language. There are more ways of avoiding ambiguity, such as manually defining operators priority and giving them left or right associativity. More information on *expression-term-factor* and grammars ambiguity in general can be found in [Aho et al., 2006].

Returning to our example, this grammar is used for the syntactic analysis of `LET`. It guides a computer program (typically a parser), through the process

of deriving each concrete sentence of the language. It also assigns a unique concrete syntax tree to each syntactically valid sentence of the language. and uses it to construct a concrete tree of the language. For example, for the program *program*, and using this CFG, the concrete tree of Figure 2.1 (page 37) would be generated (or implicitly constructed).

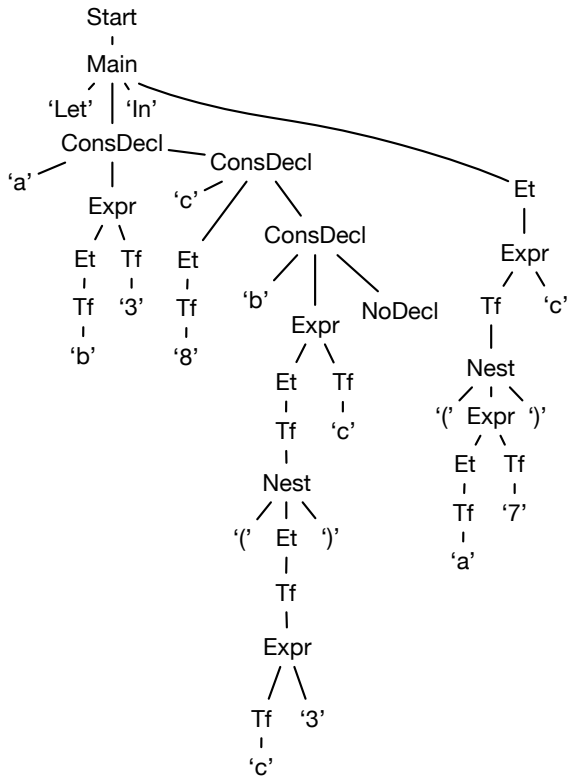


Figure 2.1: The CST for *program*.

As we have seen, when analyzing a language, syntactic analysis is only part of the process. Another important process that follows it is semantic analysis. This step differs from the previous one because when we reach it we already know the syntax of the program has no faults (or we would not have reached it in the first place). Since we need less information to structurally analyze a language (for example, we do not need any literal terminal symbols), this grammar will be much simpler:

Next, we can see the abstract grammar that structurally describes `LET`: $G_{\texttt{LET}_{Abstract}} = \langle V, \Sigma, P, S \rangle$, once again using BNF:

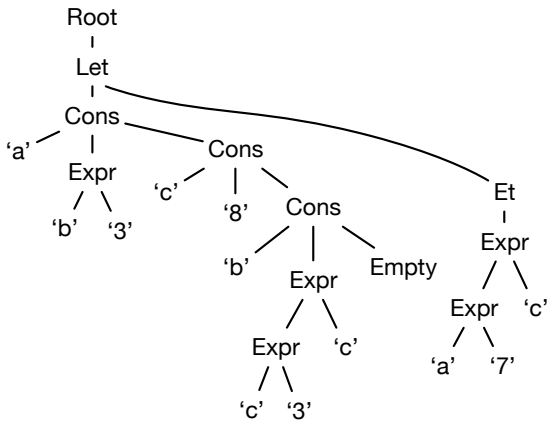


Figure 2.2: The AST for *program*.

```
(p1:  Root)   Root → Let
(p2:  Let)    Let  → Dcls Expr
(p3:  Cons)   Dcls → Name Expr Dcls
(p4:  Empty)  Dcls → ϵ
(p5:  Expr)   Expr → Expr Expr
(p6:  Var)    Expr → Name
(p7: Const)   Expr → Number
```

This CFG is much more simple than $G_{\texttt{LET}_{Concrete}}$, for two main reasons: first, no literal symbols are necessary, for the reason we have described above. Second, we can be much more clear when describing expressions, because from the concrete tree we have no problems related to ambiguity when creating its abstract counter part.

In Figure 2.2 we can see the AST that would be obtained by a parser (or any other program capable of generating an AST) for *program*.

Comparing the concrete tree from Figure 2.1 (page 37) to the abstract tree from Figure 2.2 the differences are immediately visible: despite being structurally the same, the abstract tree has less nodes and leafs and therefore is easier to reason about and to handle comparing to its concrete version.

Both grammars presented in this section are capable of syntactically defining a `LET` program, but by themselves they are not capable of completely analyzing a language. Take as example the following program:

$$
\begin{aligned}
\mathit{faulty} = \textbf{let}\ & a = c \\
& b = 3 \\
\textbf{in}\ & a * b
\end{aligned}
$$

Syntactically speaking, there is nothing wrong with the program *faulty*. It is composed by a list of definitions, either to numbers or to other variables, and a computation in the end. However, semantically speaking it is easy to notice that it is not possible to compute *faulty*, as the variable a uses another variable, c, that is not defined anywhere.

Despite CFGs being capable of defining the syntax of a language, this formalism has no way of detecting problems as the one in *faulty*. In the next sections we will introduce attribute grammars, that complement context-free grammars in the sense that we are not only capable of defining the syntax, they are also capable of defining the semantics of a language.

2.4 Attribute Grammars

Attribute grammars were first introduce by Knuth [Knuth, 1968] as an extension to CFGs. As we have seen in the previous section, CFGs define the syntax of a language. AGs extend this formalism to specify the semantic properties of the language as well.

Definition 8. (Attribute Grammar) An attribute grammar (AG) is a triple $AG = \langle G, A, D \rangle$ where:

- G is a context-free grammar such that $G = \langle V, \Sigma, P, S \rangle$;
- A is a finite set of attributes, partitioned into two sets, $A_{nont}(X)$ and $A_{loc}(p)$, $\forall X \in V$ and $\forall p \in P$. $A_{nont}(X)$ is further partitioned into two disjoint sets $A_{inh}(X)$ and $A_{syn}(X)$;
- $D = (T, E)$ is the semantic domain of the attribute grammar. T is a finite set of types and E is a finite set of semantic equations.

To create an AG, a CFG is extended as follows: attributes are associated with every grammar symbol and every production. Each attribute described

a semantic property of the language. The attributes are split into two sets: inherited attributes and synthesized attributes.

For every occurrence of a grammar symbol in a production, there is an attribute occurrence. With every production, three sets of attribute occurrences are assigned: the set of local attribute occurrences of the production, the set of input occurrences, and the set of output occurrences. The set of input occurrences consists of all inherited attribute occurrences of the left-hand side symbol of the production and all the synthesized attribute occurrences of the symbols in the right-hand side of the production. Similarly, the set of output occurrences contains all the synthesized attribute occurrences of the left-hand side symbol and all the inherited attribute occurrences of the symbols in the right-hand side of the production.

With every output attribute occurrence and every local attribute a semantic equation is defined, specifying the value of that attribute occurrence in terms of other attribute occurrences. Attribute grammars are declarative, as values of (inherited/synthesized) attribute occurrences are defined in terms of other (inherited/synthesized) attributes.

On an AG, A and D are known as the attribution rules of the attribute grammar. G specifies the (abstract) syntax of the (source) language and A and D specify the semantics of the language. Every attribute $a \in A$ is associated with either a grammar symbol $X \in V$ or a production $p \in P$. $A_{nont}(X)$ is the set of attributes associated with non-terminal X. An element $a \in A_{nont}(X)$ is denoted by $X.a$, and it can be inherited or synthesized, if $a \in Ainh(X)$ or if $a \in A_{syn}(X)$, respectively. Attributes have a type, and the function $\mathcal{T} :: A \to T$ associates a type with every attribute. This means that $\forall a \in A, (\mathcal{T}a) \in T$ is the set of all possible values of a.

All the attributes $a \in A_{loc}(p)$ are known as local attributes of a production p and are denoted by $p.a$, where all the local attributes $p.a$ represent a local attribute occurrence for the production p.

We say a production $p \in P$, $p : X_0 \to X_1 \dots X_n$, with $n \geq 0$, has an attribute occurrence $\langle p, i, a \rangle$ if $a \in A_{nont}(X_i)$, with $0 \leq i \leq n$. For every occurrence $\langle p, i \rangle$ of a symbol $X_i \in N$ in a production, sets of attribute occurrences are associated as follows:

$$O_{inh}(\langle p, i\rangle) = \{\langle p, i, a\rangle \mid a \in A_{inh}(X_i)\}$$

$$O_{syn}(\langle p, i\rangle) = \{\langle p, i, a\rangle \mid a \in A_{syn}(X_i)\}$$

$$O_{nt}(\langle p, i\rangle) = O_{inh}(\langle p, i\rangle) \cup O_{syn}(\langle p, i\rangle)$$

We defined $O_{ntoccs}(p) = \bigcup_{0 \le i \le |p|} O_{nt}(\langle p, i\rangle)$ as the set of all attribute occurrences associated with the non-terminal occurrences of the production p. Furthermore, the sets of all attribute occurrences of p is $O_{pr}(p) = O_{loc}(p) \cup O_{ntoccs}(p)$, and the set of input and output occurrences are defined, respectively, as:

$$O_{inp}(p) = O_{inh}(lhs(p)) \cup \bigcup_{1 \le i \le |p|} O_{syn}(\langle p, i\rangle)$$

$$O_{out}(p) = O_{syn}(lhs(p)) \cup \bigcup_{1 \le i \le |p|} O_{inh}(\langle p, i\rangle)$$

The function $\mathcal{T}$ is overloaded and works for attribute occurrences, which means that $\mathcal{T}\langle p, i, a\rangle = \mathcal{T}a$.

On attribute grammars, $\forall p \in P, p : X_0 \to X_1 \ldots X_n$, with $n \ge 0$, there is a set of attribute equations, denoted E_p, which is associated with a production p, and defines the value of every attribute occurrence in $O_{out}(p) \cup O_{loc}(p)$ in terms of other attribute occurrences in $O_{pr}(p)$. This implies E_p is a set of equations of the form $(\alpha_1, \alpha_2, \ldots, \alpha_x) = f(\beta_1, \beta_2, \ldots, \beta_x)$, where $k \ge 0$ and $\alpha_i \in O_{out}(p) \cup O_{loc}(p)$.

For the set of attribute equations E_p, every B_i, with $1 \le i \le k$ is either an attribute occurrence in p or an occurrence of a grammar symbol in p. When $\beta_i = \langle p, q\rangle$, for $1 \le q \le \mid p \mid$ we say it is a grammar symbol occurrence and that $\langle p, q\rangle$ is syntactically referenced in the semantic equations of AG.

A syntactic reference is therefore an occurrence of a non-terminal symbol in the production p that is directly used on the right-hand side of a semantic equation. Formally, we define as $\beta_i \in O_{pr}(p) \cup \bigcup_{0 \le p \le |q|} \langle p, q\rangle$ the symbols that occur on the right-hand side of an equation.

In de definition of E_p, f is a semantic function that maps values from

$\beta_1, \beta_2, \ldots, \beta_x$ to values of $\alpha_1, \alpha_2, \ldots, \alpha_x$, with the type $\mathcal{T}(\beta_1) \to \mathcal{T}(\beta_2) \to \cdots \to \mathcal{T}(\beta_x) \to (\ \mathcal{T}(\alpha_1) \to \mathcal{T}(\alpha_2) \to \ldots \mathcal{T}(\alpha_x)\ )$. The function $\mathcal{T}$ works on grammar symbols, as $\mathcal{T} :: A \cup V \cup \Gamma \to T$. For the special case when f represents the identity function, we call it a copy rule. We can now define $E = \bigcup_{p \in P} E_p$ as the set of all the equations in an AG.

To the sets of all attribute occurrences that are defined and used in p we call $O_{def}(p)$ and $O_{use}(p)$ respectively, which we define as:

$$O_{def}(p) = \{\alpha \mid ((\ldots, \alpha, \ldots) = f(\ldots, \beta, \ldots)\} \in E_p\}$$

$$O_{use}(p) = \{\beta \mid ((\ldots, \alpha, \ldots) = f(\ldots, \beta, \ldots)\} \in E_p \wedge \beta \in O_{pr}(p)\}$$

Definition 9. (Complete Attribute Grammar) An attribute grammar $AG = \langle G, A, D \rangle$ is a complete attribute grammar when:

- $O_{def}(p) = O_{loc}(p) \cup O_{out}(p)$;
- $\forall(\ (\ldots, \alpha_i, \alpha_j, \ldots) = f \ldots), (\ (\ldots \alpha_k \ldots) = g \ldots) \subseteq E_p, \alpha_i \neq \alpha_j \neq \alpha_k$.

is true for all $p \in P$ of the grammar G and G is a complete context-free grammar.

For a equation of the form $(\alpha_1, \alpha_2, \ldots, \alpha_x) = f(\beta_1, \beta_2, \ldots, \beta_x)$, every attribute occurrence α_i, with $1 \leq i \leq l$ depends on each attribute occurrence β_j, with $1 \leq j \leq k$. Semantic equations induce dependencies in attribute occurrences.

This, E_p induces a dependency graph $DP(p) \subseteq O_{pr}(p) \times O_{pr}(p)$, where:

- The vertices of $DP(p)$ are the attribute occurrences in $O_{pr}(p)$;
- There is a directed arc from β to α and we write $\beta \rightsquigarrow \alpha$, if α depends on β and $\alpha, \beta \in O_{pr}(p)$. This is formally written as $DP(p) = \{\beta \rightsquigarrow \alpha_1, \ldots \beta \rightsquigarrow \alpha_2 \mid ((\alpha_1, \ldots, \alpha_n) = f(\beta_1, \ldots, \beta_n) \in E_p)\}$;
- $DP = \bigcup_{p \in P} DP(p)$ is the relation of direct dependencies among attribute occurrences associated to productions.

The definition of attribute grammar presented here is, with slight differences, similar to the ones presented in [Kastens, 1980; Alblas, 1991; Pennings, 1994].

The dependency graph summarizes the attribute dependencies associated with a production p.

2.4.1 Attributed and Decorated Trees

We have seen in Section 2.2.1 that from an abstract grammar we can create an abstract tree for every sentence in a language defined by it. Similarly, the attribution rules of an attribute grammar assigns attributes to nodes of a syntax tree, to which we call attributed tree or undecorated tree.

Definition 10. (Attribute Tree) An attribute tree T is defined as follows: for all the nodes $N \in T$ that is an instance of a non-terminal symbol X, attribute instances are assigned, which correspond to attributes of X. For all attributes $a \in A_{nont}(X)$ the correspondent instance is denoted $N.a$. For all local attribute of $p, l \in A_{loc}(p)$ induces an attribute instance on the node N, with p being a production of N. Such instances are denoted as $N.l$.

To the process of computing values of attribute instances in an attributed tree T according to the semantics of the AG we call attribute evaluation or tree decoration, and a program that performs such task is called an attribute evaluator. Thus, a decorated tree is an attributed tree where all the attribute instances have been calculated.

There are various methodologies to implement evaluation on attribute grammars, called attribute evaluators [Hoover and Teitelbaum, 1986; Reps and Demers, 1987; Jones, 1990; Kaiser and Kaplan, 1993; Middelkoop et al., 2011; Bransen et al., 2014].

We call the meaning of an AG to the value of the synthesized attribute instances associated with the top-most position (root) of the decorated attributed tree. We say that two attribute grammars are equivalent if they associate the same meaning to every sentence of the language they define.

Definition 11. (Equivalent Attribute Grammars) To attribute grammars $AG_1 = \langle G_1, A_1, D_1 \rangle$ and $AG_2 = \langle G_2, A_2, D_2 \rangle$ are said to be equivalent

attribute grammars, which is written as $AG_1 \equiv AG_2$ if and only if $G_1 \equiv G_2$ (they define the same language $\mathcal{L}$) and the rules $A_1 \cup D_1$ and $A_2 \cup D_2$ associate the exact same meaning to every sentence $s \in \mathcal{L}$.

An attribute grammar can also be well defined or bad defined.

Definition 12. (Well-defined Attribute Grammar) An attribute grammar $AG = \langle G, A, D \rangle$ is said to be a well-defined attribute grammar if and only if all the attribution rules $A \cup D$ are such that for each attribute tree generated by the grammar G, all the values of the attribute instances of the tree are computable.

Uninformally, an AG is well-defined if all the values of the attribute instances within each tree T can be computed by some attribute evaluation process.

2.4.2 Circularities in Attribute Grammars

In the same way attribution rules induce dependencies among attribute occurrences, they also induce dependencies among attribute instances. In particular, we say that an attribute instance $N_j.\alpha$ depends on an instance $N_i.\beta$ if and only if for a $p = prod(N)$, $\langle p, i, \beta \rangle \rightsquigarrow \langle p, j, \alpha \rangle \in DP(p)$. This, we denote $DTR(T)$ the dependency graph for the dependencies among attribute instances in an attribute tree T.

$DTR(T)$ is always defined by taking the attribute instances of an attributed tree as its vertices. If the attribute instance $N_j.\alpha$ depeds on the attribute instance $N_i.\beta$, $DTR(T)$ will contain a direct arc from $N_j.\alpha$ to $N_i.\beta$.

Traditional definitions of well-defined AGs, such as the ones found in [Knuth, 1968; Alblas, 1991] enforce a-cyclic attribute dependency graphs induced for every attributed tree. This means that attribute instances must have a partial evaluation order, and no attribute instance may transitively depend on itself.

Definition 13. (Non-Circular Attribute Grammar)) An attribute grammar $AG = \langle G, A, D \rangle$ is called a non-circular attribute grammar if for every attribute tree generated by the AG, $DRT(T)$ is a-cyclic.

There are well-known attribute grammar techniques that statically check for circularities within attribute grammars, such as [Knuth, 1968; Alblas, 1991; Rodeh and Sagiv, 1999; Sasaki and Sassa, 2004; Schäfer et al., 2009] and algorithms that statically compute an order for the evaluation of attributes [Kastens, 1980].

2.5 Attribute Grammar Specification

In this section we will demonstrating how to specify the semantics of a language, namely a `LET` program, as an AG. Our goal when processing a `LET` program is to compute the semantics (i.e., the value) of a `LET` program. Implementing this computation introduces typical language processing challenges:

1. Name/scope analysis in order to verify whether or not all the variables that are used are indeed declared;

2. Semantic analysis in order to calculate the meaning of the program; this analysis incorporates name analysis through symbol table management and processing of the algebraic expressions that compose a program.

To introduce attribute grammars, we will focus the presentation on a name analysis task. After having solved this task, the computation of the result of a `LET` expression is very simple. That computation via AGs will be described later.

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 declaration is considered, which we will implement in the inherited attribute `dcli` (declarations in). In the *program* above, if `dcli` was to be computed in the node for $b = (c*3) - c$, it will be a list containing `a` and `c`, because these are the variables declared before this declaration. The attribute synthesized `dclo` (declarations out), synthesizes/returns all the declared variables in the program. Both these attributes are lists of identifiers.

2. Distribute all the declared variables in a program throughout the tree, which we will implement in the inherited attribute `env` (environment). This will always produce the list of declared variables, regardless of the position on the tree where the attribute is accessed. `env` is a list of identifiers.

3. Calculate the list of invalid declarations, i.e., variables that have been declared twice and variables that are being used in an expression but have not been declared. For the AG that performs the scope/name analysis this attribute will constitute the meaning of the grammar, its final result, and will be called `errs` (errors). The returning type of `errs` will be a list of identifiers.

In the definition of an the AG, we use a notation similar to the one in [Paakki, 1995], 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). Please recall that 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.

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

- `++` for lists concatenation
- `[]` represents an empty list
- `:` for the addition of an element to a list
- We also use the functions *mBIn* and *mNBIn*, as defined in Section 1.4.

Attributes have a type. Therefore, we define the type of pseudo-terminal symbols as *Name* and assume it exists and is provided externally. We also define a new type, the environment, which we denote by *Env*. *Env* represents a list of identifiers and its type is defined in `Haskell` as:

type $Env = [Name]$

It is possible to use a different notation, unrelated to `Haskell`, to define new types. They can be, for example, defined directly in the AG formalism. This is quite common, and the idea is to define new non-terminals that describe the type. *Env* for example, could be defined in an AG fashion by the following non-terminals and productions:

```
(p1: Cons) Env  → Name
(p2: Nil)  Env  → ϵ
```

The type *Env* and the non-terminal *Env* are isomorphic, as the non-terminals *Cons* and *Nil* correspond to the `Haskell` built-in functions `:` and `[]`, respectively.

In order to the define an AG that implements name analysis for `LET`, we will split it into three different semantic domains. We do so for clarity and to focus in each one individually. These will represent three semantic domains of the analysis itself: capturing the declaration of variables, distributing the environment and calculating the errors, if they exist.

2.5.1 Capturing Variable Declarations

In order to capture variable declarations, a typical solution in a functional settings is to implement a recursive function that starts with an empty list and accumulates each declaration in a list while traversing it. Such function returns the accumulated list of declaration as its final result. This technique is known as accumulating parameters [Bird, 1998] (or simply accumulators). In AGs, accumulators are typically implemented as a pair of inherited and synthesized attributes, representing the usual argument/result pair in a functional setting. This pattern can be seen in the attributes `dcli` and `dclo` that we present next. Both this attribute have the same returning type: *Env*.

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

```
inherited attribute dcli :: Env
dcli is defined for Root, Let, Dcls

(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 dupicated variable declarations of the same variable.

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

```
synthesized attribute dclo :: Env
dclo is defined for Root, Let, Dcls

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

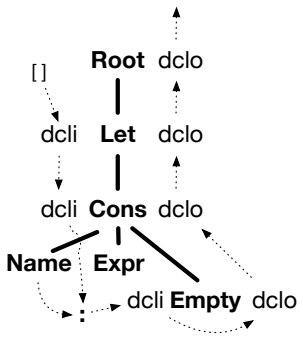


Figure 2.3: The relation between the inherited attribute `dcli` and the synthesized attribute `dclo`, implementing an accumulation pattern.

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

Another important remark about the attributes `dcli` and `dclo` is that they are only declared for the productions `p1-p3` (and `p4`, in the case of `dclo`), and not for the entire CFG. This is typical of AGs as sometimes, as is the case, specific semantics depend only on specific parts of the tree/language. The full pattern of attribute calculation can be seen for a simple tree in Figure 2.3.

2.5.2 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 type is *Env* and definition we present next:

```
inherited attribute env :: Env
env is defined for Root, Let, Dcls, Expr
```

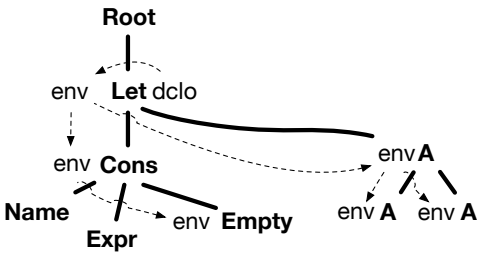


Figure 2.4: The inherited attribute `env`, distributing the environment throughout the tree.

```
(p1: Root)  Root   → Let
      { Let.env = Let.dclo }
(p2: Let)   Let    → Dcls Expr
      { Dcls.env = Let.env
      , Expr.env = Let.env }
(p3: Cons) Dcls1   → Name Expr Dcls2
      { Expr.env  = Dcls1.env
      , Dcls2.env = Dcls2.env }
(p5: Expr) Expr1 → Expr2 Expr3
      { Expr2.env = Expr1.env
      , Expr3.env = Expr1.env }
```

The attribute `env` is present everywhere in the tree with the same value. The equations go all the way up the tree to obtain the `dclo` attribute of the root. The inherited attribute `env` and its relation with `dclo` can be seen in Figure 2.4.

2.5.3 Calculating Invalid Identifiers

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

This list represents the meaning of the grammar and is calculated by the attribute `errs`, whose returning type is *Error*, which can be defined in an

AG fashion by the following non-terminals and productions:

```
(p1: Cons) Error  → Name
(p2: Nil)  Error  → ϵ
```

Next, we present the definition os *errs*:

```
synthesized attribute errs :: Error
errs is defined for Root, Let, Dcls, Expr

(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: Expr) Expr1 → Expr2 Expr3
      { Expr1.errs = (Expr2.errs) ++ (Expr3.errs) }
(p6: Variable)  Expr → Name
      { Expr.errs = mBIn Name.lexeme Expr.env }
(p7: 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 duplicated 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`.

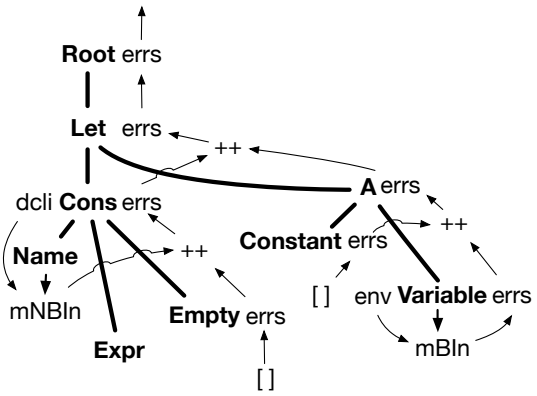


Figure 2.5: The synthesized attribute `errs`.

Whenever variables are used inside expressions we have to see if they have been declared before. This means that the semantics for `errs` in the production `p9` checks the list produced by `env` (containing all the variables of the program) and to see if the variable is present. In Figure 2.5 we can see how this attribute is defined throughout the abstract tree of `LET` and how it relates to the attributes `dcli` and `env`.

The strict and circular programs from Sections 1.4.2 and 1.4.1 can be generated from such an attribute grammar specification using the techniques proposed in [Kuiper and Saraiva, 1998; Saraiva and Swierstra, 1999a; Saraiva, 1999].

2.5.4 Decorated Tree

We have seen in the previous sections how an AG, which we split into semantic domains, can implement the semantics of a programming language, `LET`. These domains are made out of 4 attributes that together provide a meaning for the grammar, i.e., compute errors in the declarations and use of variables.

As we have seen in Section 2.4.1, on an attribute grammar attributes are assigned to tree nodes and their definitions is interdependent on other attributes in the same node or in nodes that are immediately above or behind the current site. To this tree with semantic equation in each node we gave the name of decorated tree, and it is the computation of all these semantic

equations that provides the final result of the grammar.

In Figure 2.6 (page 54) we can see the decorated tree generated for the *program* of Section 2.3. In particular, in this figure we have the exact same AST as we have seen in Figure 2.2 (page 38). As we have seen, we use an AST because it provides the same structural information as an CST but without unnecessary syntactic clutter, and we decorate that tree with semantic information.

For the specific case of the decorated tree of Figure 2.6, the returned result will be an empty list, as there are no faults for the *program* it defines. Although AGs are generic, and define the semantic of a language in the same way that a CFG defines the syntax of all programs of certain language, the decorated trees are specific for every specific instance of, in this case, a `LET` program. This happens for the exact same reason an AST is different for two different program: the user might use different constructs and strategies for equally valid programs, and the AST (and the decorated tree) represent the syntax (and the semantics) for that specific program.

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.

In attribute grammars, programmers do not have to define intermediate gluing data structures nor concern themselves in scheduling traversals, like in straightforward implementations we presented in Section 1.4.1. In fact, well-known AG techniques automatically infer such data structures and how to traverse the AST from the AG specification. Moreover, they are modular: we can add new functionality by just add a new fragment to the AG specification.

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

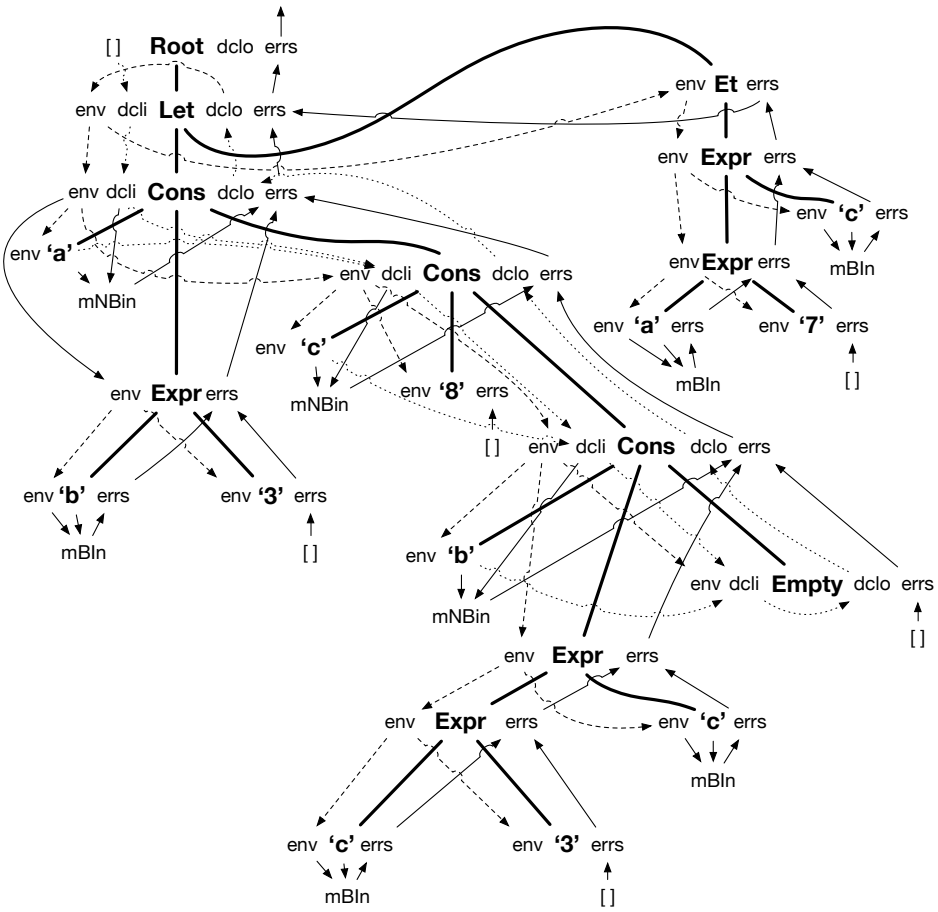


Figure 2.6: The decorated tree for the LET *program* of Section 2.3. The capture of variables declarations is represented with pointed lines, the distribution of variables with dashed lines and the calculation of identifiers with black lines.

2.6 Conclusions

In this chapter we have seen definitions and notations for context-free grammars, attribute grammars and Σ-algebras, formalisms that will be used throughout this thesis and that are needed to understand the work developed in this context.

In the next chapter we will present our setting for embedding attribute grammars in `Haskell`.

Chapter 3

Embedding Attribute Grammars

Summary

In this chapter we present an embedding of attribute grammars in a functional setting. This embedding is based on functional zippers [Huet, 1997], which we also introduce. We will use the navigational power of this technique to define computations in `Haskell` *that retain the main characteristics of attribute grammars, namely their modularity and expressiveness.*

3.1 Introduction

In this chapter we define how attribute grammars can be embedded in a functional setting using functional zippers.

We will define data types in `Haskell` to represent our grammars, on top of which we will define attributes as they were presented in the previous chapter, which is sketched in Figure 3.1. As they were defined previously, attributes in the canonical definition of AGs may only depend on information from neighbor nodes, but we will leverage this limitation in the following chapters.

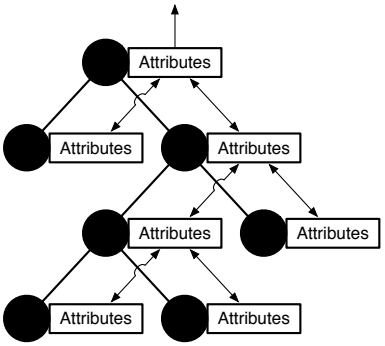


Figure 3.1: Overview of what we will define in this chapter.

In order to embed AGs we relly on the notion of zippers. A zipper is a technique for representing a data structure where traversals are simplified through primitives that allow simple navigation of the underlying structure. This makes this setting convenient and powerful to describe attribute grammars functionaly: the same simple mechanisms provided by zippers that allow movement up or down the tree, also allow us to write functions that, as attribute equations, are defined based on their context on a specific structure.

We start by describing different approaches that relate to the ones we are presenting, either because they use a zipper-base setting to describe and implement programs, or because they describe different techniques to embed attribute grammars that do not rely on zippers.

After, we describe in detail the concept of zippers through their definition and through examples that show the construction and usage of such technique.

We then describe how the attribute grammars can be described in our setting, by providing zipper-based implementations of both the context-free grammars and the attributes we have seen before for `LET`. As we will see, the specification of attribute grammars in our setting is very similar to the one presented in the previous chapter.

3.2 Functional Zippers

In this section we will introduce functional zippers (or simply zippers), which are a central part of our technique for embedding AGs in a functional setting.

Zippers [Huet, 1997] were originally introduced 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 other programming languages, functional or not, lazy or not ([Huet, 1997]), which means a technique expressed via zippers, for example an attribute grammar, can be achieved in other environments as well.

In order to illustrate the concept of zippers, let us introduce a simple `Haskell` data type that represents binary leaf trees:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree} = \mathit{Fork}\ \mathit{Tree}\ \mathit{Tree}\\ &\qquad\quad\ \ |\ \mathit{Leaf}\ \mathit{Int}\end{aligned}$$

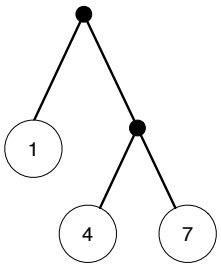
which has only two data constructors, *Fork* and *Leaf* and integers as leaf values. Let us also introduce a possible instance if this data type:

$$\begin{aligned}&\mathit{tree} = \mathit{Fork}\ (\mathit{Leaf}\ 1)\\ &\qquad\qquad\ \ (\mathit{Fork}\ (\mathit{Leaf}\ 4)\\ &\qquad\qquad\qquad\ \ (\mathit{Leaf}\ 7))\end{aligned}$$

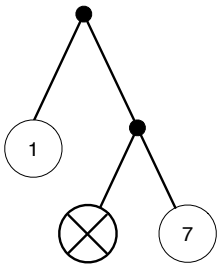
which represents a tree with two nodes and three leafs. Its visual representation can be seen in Figure 3.2a.

We may notice that, in particular, each of the subtrees occupies a certain location in tree, if we consider it as a whole. That location may be represented by the subtree under consideration and by the rest of the tree, which is viewed as the context of that subtree.

One of the possible ways to represent this context is as a path from the top of the tree to the site which is the focus of attention (where the subtree starts). This means that it is possible to use contexts and subtrees



(a) No focus selected.



(b) Focus on the production *Plus*.

Figure 3.2: Representation of the binary tree *tree*, with and without a focus of attention.

to define any position in the tree, while creating a setting where navigation is facilitated by the contextual help of a path applied to a tree.

For example, if we want to put the focus in *Leaf* 4, as can be seen in Figure 3.2b, its context in tree is:

$$\begin{aligned} tree = Fork\ (Leaf\ 1) \\ (Fork\ \otimes \\ (Leaf\ 7)) \end{aligned}$$

where $\otimes$ points the exact site in the where *Leaf* 4 appears. On of the possible ways of representing this context is as a path from the top of the tree to the position which is the focus of attention. To reach *Leaf* 4 in tree, we need to go *right* (down the right branch) and then *left* (down the left one). In practice, this means that given the tree in Figure 3.2a and the path `[right, left]` would suffice in indicating the site of the tree we want to focus our attention in.

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

Using this idea, we may represent contexts as instances of the following data-type:

$$\textbf{data}\ Cxt\ a = Top$$

$$\mid\ L\ (Cxt\ a)\ a$$
$$\mid\ R\ a\ (Cxt\ a)$$

A value $L\ c\ t$ represents the left part of a branch of which the right part was t and whose parent had context c. Similarly, a value $R\ t\ c$ represents the right part of a branch of which the left part was t and whose parent had context c. The value *Top* represents the top of the tree. Returning to our example, we may represent the context of *Leaf* 4 in tree as:

$$context = L\ (R\ (Leaf\ 1)\ Top)\ (Leaf\ 7)$$

which is the same as saying we go right once $(R\ (Leaf\ 1)\ Top)$, to a focus where our left side is *Leaf* 1 and our context is the root of the tree (Top) and then from here we go left, to a position where our context is $(R\ (Leaf\ 1)\ Top)$ and our right side has *Leaf* 7.

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

$$\textbf{type}\ Loc\ a = (a, Cxt\ a)$$

which represents a pair with a tree together with a context. We are now ready to present the definition of useful functions that manipulate locations on binary trees. We can define a function that goes down the left branch of a tree:

$$left :: Loc\ Tree\ \rightarrow\ Loc\ Tree$$
$$left\ (Fork\ l\ r, c) = (l, L\ c\ r)$$

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

$$right :: Loc\ Tree\ \rightarrow\ Loc\ Tree$$
$$right\ (Fork\ l\ r, c) = (r, R\ l\ c)$$

and behaves exactly as the function *left* but doing the exact inverse. We also define a function to go up on a tree location:

$$
\begin{aligned}
&\mathit{parent} :: \mathit{Loc}\ \mathit{Tree} \rightarrow \mathit{Loc}\ \mathit{Tree} \\
&\mathit{parent}\ (t, L\ c\ r) = (\mathit{Fork}\ t\ r, c) \\
&\mathit{parent}\ (t, R\ l\ c) = (\mathit{Fork}\ l\ t, c)
\end{aligned}
$$

In the function *parent*, if we had a location *Loc* whose context was a left side (L), we create a new location where we move the right side of the context to a new tree and keep the left side as the context (c). We do exactly the opposite if we had a location whose context was a right side (R). Finally, we create a function that creates a tree location from a tree, where we create an empty context:

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

and another that extracts a tree from a tree location, which simply takes the first element of the location (pair):

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

These functions can be used, for example, to focus on the subtree *Leaf* 4 of *tree*. Recall, by looking at Figure 3.2b that *Leaf* 4 was reachable by going down the right branch of tree and then down the left branch of the resulting subtree. So, focusing on *Leaf* 4 is achieved by creating a tree location, and then going right followed by going left, as can be seen below:

$$
\mathit{subtree} = \mathit{left}\ (\mathit{right}\ (\mathit{zip}\ \mathit{tree}))
$$

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

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

with *edit* predictably yielding the result:

$$\begin{aligned} edited = Fork\ &(Leaf\ 1) \\ &(Fork\ (Leaf\ 5) \\ &\quad (Leaf\ 7)) \end{aligned}$$

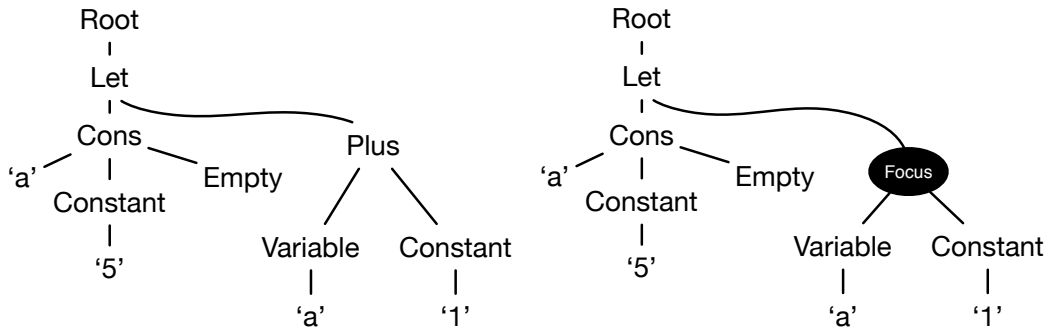
The zipper data structure that we have reviewed in this section provides an elegant and efficient way of manipulating locations inside a data structure. In the next section we will see how we can use zippers to navigate through `LET` programs.

3.2.1 Generic Zippers

In this section we will show why generic zippers are useful in handling big data types and how we can use a specific generic zipper library to navigate through `Haskell` instances of `LET` programs. We start by recalling the `Haskell` data type presented in Section 1.4, which can also be straightforwardly obtained from the abstract syntax of the `LET` language:

$$\begin{aligned} &\textbf{data}\ Root = Root\ Let \\ &\textbf{data}\ Let = Let\ Dcls\ Expr \\ &\textbf{data}\ Dcls\ = Cons\ String\ Expr\ Dcls \\ &\qquad\qquad \mid Empty \\ &\textbf{data}\ Expr = Plus\ \ \ Expr\ Expr \\ &\qquad\qquad \mid Minus\ \ Expr\ Expr \\ &\qquad\qquad \mid Times\ \ Expr\ Expr \\ &\qquad\qquad \mid Divide\ \ Expr\ Expr \\ &\qquad\qquad \mid Variable\ String \\ &\qquad\qquad \mid Constant\ Int \end{aligned}$$

All of the data types presented here are isomorphic to the abstract grammar for `LET` that we have seen in Section 2.3 (page 38). We have given `Haskell`'s data types the same name we have given non-terminals on the abstract grammar of `LET`, and we have given `Haskell`'s data constructors the same name we have given the productions on the abstract grammar of `LET`.



(a) No focus selected. (b) Focus on the production *Plus*.

Figure 3.3: Tree representation of $zipper_{example}$.

Using this data type, a `LET` program can be expressed as an element of *Root*. In order to introduce the concept of zipper, we consider again a simple `LET` program:

$$\begin{aligned} zipper_{example} = \; & \textbf{let } a = 5 \\ & \textbf{in } \; a + 1 \end{aligned}$$

and its representation as an instance of the `Haskell` data type *Root*:

$$\begin{aligned} zipper_{example} = \mathit{Root}\ (\mathit{Let} \quad & (\mathit{Cons}\ \texttt{"a"}\ (\mathit{Constant}\ 5) \\ & \quad \mathit{Empty}) \\ & (\mathit{Plus}\ (\mathit{Variable}\ \texttt{"a"}) \\ & \quad (\mathit{Constant}\ 1))) \end{aligned}$$

The visual representation of this instance can be seen in Figure 3.3a, and is similar to the ASTs for a `LET` program we have seen in Section 2.3 (page 38).

In Figures 3.3a and 3.3b we can see the same tree, which is a graphic representation of $zipper_{example}$, but in the latter we can see that a specific position, *Plus*, is the focus of attention in the tree, i.e., is the place we want to navigate to in order to perform certain operations.

We can describe the node *Plus* in the tree (the focus), by describing the path that needs to be taken, starting in the top of the tree, in order to reach the focus. In particular, we need to go *down* from the *Root* and then *right*, to the right branch. This is similar to what we have seen in the previous section, and could have followed the same technique of implementing

a custom, simple zipper library to perform this task. However, doing so has two great disadvantages:

1. The `Haskell` data type we have created the zipper library for is extremely simple: only one data type and two data constructors, *Fork* and *Leaf*. Implementing the library for the much more complex *Root* data type would be time consuming and error prone, a task whose complexity would grow with the complexity of the data type we need to represent a language (and even though *Root* is more complex than *Tree*, `LET` is still a very simple programming language);

2. Every time we need a new implementation with new data types, or every time we need to change the current data type (for example, if we want to update part of the language), that implies re-writing and changing the zipper library.

For these reasons, in the embedding we will present in this work work we use the generic zipper `Haskell` library of Adams [Adams, 2010]. This library works for both homogeneous and heterogeneous data types. The library can traverse any data type that has an instance of the `Haskell`'s `Data` and `Typeable` type classes [Lämmel and Jones, 2003].

Typical of zipper libraries, this one provides a set of functions, such as *up*, *down*, *left* and *right* that allow the programmer to easily navigate through a generic structure. The function *getHole* returns the subtree which is the current focus of attention. Furthermore, this library provides a hand full of potentially helpful functions that we would have to manually write.

In our setting, on top of the zipper library of Adams [Adams, 2010] we have implemented several simple abstractions that facilitate the embedding of attribute grammars. In particular, we have defined:

- $(.\$) :: \mathit{Zipper}\ a \to \mathit{Int} \to \mathit{Zipper}\ a$, for accessing any child of a structure given by its index starting at `1`;

- $\mathit{parent} :: \mathit{Zipper}\ a \to \mathit{Zipper}\ a$, to move the focus to the parent of a concrete node;

- $(.|) :: \mathit{Zipper}\ a \to \mathit{Int} \to \mathit{Bool}$, to check whether the current location is a sibling of a tree node;
- $\mathit{constructor} :: \mathit{Zipper}\ a \to \mathit{String}$, which returns a textual representation of the data constructor which is the current focus of the zipper.

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

In order to see the generic zippers library in action, together with our abstractions, we may use the algebraic expression which represents the program in Figure 3.3a, $\mathit{zipper}_{\mathit{example}}$, and easily wrap it in a zipper,

$$\mathit{expr}_{\mathit{zipper}} = \mathit{to}_{\mathit{Zipper}}\ \mathit{zipper}_{\mathit{example}}$$

and check the constructor of the current node:

$$\mathit{constructor}\ \mathit{expr}_{\mathit{zipper}} \equiv \texttt{"Root"}$$

which makes perfect sense since we have wrapped $\mathit{zipper}_{\mathit{example}}$ inside a zipper but have not go changed the focus on the structure. To go to the first child and check the constructor name, we write:

$$\mathit{child} = \mathit{expr}_{\mathit{zipper}}\ .\$\ 1$$
$$\mathit{constructor}\ \mathit{child} \equiv \texttt{"Let"}$$

and go to the second child (to the focus displayed on Figure 3.3b):

$$\mathit{focus} = \mathit{child}\ .\$\ 2$$
$$\mathit{constructor}\ \mathit{focus} \equiv \texttt{"Plus"}$$

Finally, we can define functions such as $\mathit{lexemeConstant}_1 :: \mathit{Zipper}\ a \to \mathit{Int}$, where

$$\mathit{lexemeConstant}_1\ (\mathit{focus}\ .\$\ 2) \equiv 1$$

extracts information from the zipper, simulating a standard lexer. Throughout the rest of this thesis, the name of these lexeme functions will always have the form $\mathit{lexemeConstructor}_{\mathit{child}}$, where *constructor* corresponds to the

current data constructor and *child* corresponds to the number of the child whose lexeme we want to obtain.

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

3.3 LET as an Embedded Attribute Grammar

With zippers introduced, we can now show how the AG presented in the previous chapter can be specified in the functional language `Haskell`. On our setting, we will use the expressive power of zippers to defined small, modular computations that depend on other small computations in other sites of the trees. This is exactly how we defined AGs.

We start with the inherited attribute *dcli*. Recall that this is an inherited attribute that goes top-down in the tree, collecting declarations of variables. We can define it in `Haskell` as:

```
dcli :: Zipper Root → Env
dcli ag = case (constructor ag) of
  "Root" → []
  _      → case (constructor (parent ag)) of
             "Cons"  → (dcli (parent ag))
                         ++ [lexemeCons1 (parent ag)]
             "Empty" → dcli (parent ag)
```

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

This AG fragment is very similar to the attribute `dcli` presented in Chapter 2 (page 48). For *Cons* for example, we say *dcli* is the concatenation of information from the parent node and local information. This is similar both

in this `Haskell` fragment and in the attribute as it was defined before.

Note that the usage of the wildcard symbol ('_') means that we are declaring *dcli* for all the tree nodes, whereas in the AG defined in Section 2.5 it is only declared for a few productions. This is not a problem, as even if *dcli* is only defined for a few production, which we can always do in our setting, it behaves exactly the same everywhere, so we are simplifying its definition.

Next, we present the synthesized attribute *dclo*:

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

This attribute collects the whole list of declared variables. Therefore, it goes up 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.

Once again, this is very similar to the AG fragment `dclo` from Chapter 2 (page 49), as we can easily see.

A similar approach is used when defining *env*:

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

where we defined the attribute for the top most production and then instruct it to go down 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` [Van Wyk et al., 2008] and references to remote attributes in `LRC` [Kuiper and Saraiva, 1998]).

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*:

```
errs :: Zipper Root → Error
errs ag = case (constructor ag) of
  "Root"     → errs (ag .$ 1)
  "Let"      → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Cons"     → mNBIn (lexemeCons1 ag) (dcli ag)
                 ++ errs (ag .$ 2) ++ errs (ag .$ 3)
  "Empty"    → []
  "Plus"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Divide"   → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Minus"    → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Time"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Variable" → mBIn (lexemeVariable1 ag) (env ag)
  "Constant" → []
```

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

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

The structured nature of our embedding might provide an easier setting for debugging as the entire definition of one attribute is localized in one

semantic function. Furthermore, we believe that the individual attribute definitions in our embedding can straightforwardly be understood and derived from their traditional definition on an attribute grammar system, as can be observed comparing the attribute definitions in the previous section with the ones in this section.

An advantage of the embedding of DSLs in a host language is the use of target language features as native. In our case, this applies, e.g., to the `Haskell` functions `++` and `:` for list concatenation and addition, whereas in specific AG systems the set of functions is usually limited and predefined. Also, regarding distribution of language features for dynamic loading and separate compilation, it is possible to divide an AG in modules that, e.g., may contain contain data types (representing the grammar) and functions (representing the attributes).

The zipper-based embedding AG is both a concise and elegant specification of an attribute grammar and a correct `Haskell` program. This means that we can execute the specification without the need of compiling it. For example, if we write:

$$errs\ (to_{Zipper}\ (zipper_{example})) \equiv [\,]$$

this is a real `Haskell` program being interpreted.

Note this is very different from the classical implementation of AGs. systems like `LRC`, `Silver`, `JastAdd` or `Eli` compile an attribute grammar specification (like the one presented in Chapter 2) into a target language (`Java`, `C`, `Haskell`, ...). In the next section we describe approaches to AGs similar to ours.

If we compare this AG-based approach to the ones we presented in Section 1.4, we can see that ours is different in some points:

- The circular approach was not modular, and the strict one lost the modularity with the need to introduced intermediate data types. Our solution is mode modular than both, as the entire computation is split into small functions easy to change;

- Our approach does not require a lazy evaluation engine;

- With our approach the programmer does not have to be concerned with function scheduling. Attributes are written individually and it is their evaluation as a whole that represents the necessary tree traverses;

- Our approach does not require gluing data types, that require an addition effort both to be implemented and to be maintained.

3.4 Functional Embeddings of Attribute Grammars

There are various approaches and techniques that relate to the one present in this work. Next, we present such approaches.

3.4.1 Zipper-based approaches

Uustalu and Vene have shown how to embed attribute computations using co-monadic structures, where each tree node is paired with its attribute values [Uustalu and Vene, 2005]. This approach is notable for its use of a zipper as in our work. However, it appears that this zipper is not generic and must be instantiated for each tree structure. Laziness is used to avoid static scheduling. Moreover, their example is restricted to a grammar with a single non-terminal and extension to arbitrary grammars is speculative.

Badouel *et al.* define attribute evaluation as zipper transformers [Badouel et al., 2007, 2013]. While their encoding is simpler than that of Uustalu and Vene, they also use laziness as a key aspect and the zipper representation is similarly not generic. This is also the case of [Badouel et al., 2011], that also requires laziness and forces the programmer to be aware of a cyclic representation of zippers.

Yakushev *et al.* describe a fixed point method for defining operations on mutually recursive data types, with which they build a generic zipper [Yakushev et al., 2009]. Their approach is to translate data structures into a generic representation on which traversals and updates can be performed, then to translate back. Even though their zipper is generic, the implementation is

more complex than ours and incurs the extra overhead of translation. It also uses more advanced features of `Haskell` such as type families and rank-2 types.

3.4.2 Non-zipper-based approaches

Circular programs have been used in the past to straightforwardly implement AGs in a lazy functional language [Johnsson, 1987; Kuiper and Swierstra, 1987]. These works, in contrast to our own, rely on the target language to be lazy, and their goal is not to embed AGs: instead they show that there exists a direct correspondence between an attribute grammar and a circular program.

Regarding other notable embeddings of AGs in functional languages [de Moor et al., 2000a; Viera et al., 2009; Viera, 2013], they do not offer the modern AG extensions that we provide, with the exception of [Viera, 2013] 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 [de Moor et al., 2000a] or heterogeneous collections [Viera et al., 2009; Viera, 2013]. The use of heterogeneous lists in the second of these approaches replaces the use in the first approach of extensible records, which are no longer supported by the main `Haskell` compilers. In our framework, attributes do not need to be collected in a data structure at all: they are regular functions upon which correctness checks are statically performed by the compiler. The result is a simpler and more modular embedding. On the other hand, the use of these data structures ensures that an attribute is computed only once, being then updated to a data structure and later found there when necessary. In order to guarantee such a claim in our setting we need to rely on memoization strategies, often costly in terms of performance.

Our embedding does not require the programmer to explicitly combine different attributes nor does it require combination of the semantic rules for a particular node in the abstract syntax tree, as is the case in the work of Viera *et al.* [Viera et al., 2009; Viera, 2013]. In this sense, our implementation requires less effort from the programmer.

3.5 Conclusion

In this chapter we have seen how canonical attribute grammars can be described in a functional setting using zippers. These allow the definition of functions with different behaviors depending on their context within a tree. This is exactly how attributes are defined.

The solution we have seen here closely resembles the classic formalisms for context-free and attribute grammars, with our approach being expressive and capable of implementing the syntax and the necessary semantics for `LET` on the target functional language `Haskell`.

In the next chapters we will see that zippers are not only powerful enough to define classic attribute grammars, they also provide enough expressiveness to define various attribute grammar extensions, from which we start by presenting references.

Chapter 4

Reference Attribute Grammars

Summary

In this chapter we introduce a new extension to attribute grammars: references. References allow the specification of attributes whose dependencies are in random sites in the tree allowing, for example, the definition of graph structured on abstract syntax trees. The extension is presented using the attribute grammar system `JastAdd` *and an example is presented both in this system and in our embedding.*

4.1 Introduction

In the previous chapter we have seen how attribute grammars can be embedded in `Haskell` with the support of functional zippers. In this chapter we will introduce reference attribute grammars and show how this extension can also be embedded in out setting.

References allow attributes to be defined with dependencies in random sites on the AST, and not only in parent/children nodes as in classic AGs. This allows the implementation, for example, of graph structures on the tree. We will use zippers to embed references and define computations pointing to random points of a tree.

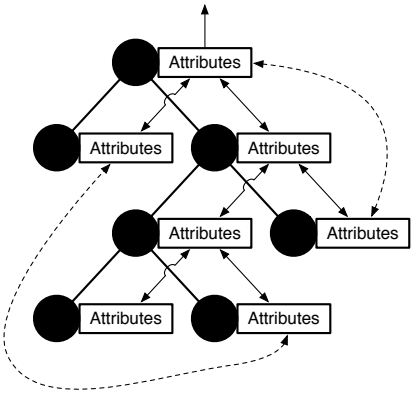


Figure 4.1: Overview of what we will define in this chapter. Attributes with references can be seen as dashed lines.

Figure 4.1 shows schematically the type of attribute grammars we will define in this chapter. This time we will be able to define attributes that can point to random tree sites, as can be seen by the dashed lines.

The AG system `JastAdd` is a well-known AG system that supports references. We will use it to exemplify how references are used in attribute computations within an attribute grammars. This example will also work as example for the embedding of this new kind of attribute grammars.

4.2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) were first introduced by Magnusson and Hedin [Hedin, 1999]. 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.

A problem with the standard formalism for attribute grammars is that specifications often become very low level in cases where we are dealing with dependencies that are far from our attributes, where the properties of a specific node has dependencies on properties of distant nodes.

With references one can define attribute with direct dependencies to any

site on the tree, allowing for example the construction of graph structures within attribute grammars, and language analysis tasks such as name and type analysis for object-oriented languages can elegantly be defined with this extension [Hedin, 1999; Ekman and Hedin, 2006; Magnusson and Hedin, 2000]: the nodes on the AST that represent a use of a variable contain mechanims to access its declaration to know, for example, its type.

4.2.1 Reference Attribute Grammars in `JastAdd`

`JastAdd` is a meta compilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-oriented features.

A tutorial for `JastAdd` can be found in [Hedin, 2011], where the inner works and the attribute grammar declarative nature of this system is explained, and keys problems are solved as examples. Here, we will just sketch a solution that uses RAG, where we will present the key fragments of how it would be defined on this system.

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

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 (again) valid `Haskell` code:

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

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 $(c * 3) - c$.

Implementing the syntax for nested sub-expressions means extending the AST of `LET`, as can be seen next:

```
(p1:  Root)     Root → Let
(p2:  Let)      Let  → Dcls Expr
(p3:  Cons)     Dcls → Name Expr Dcls
(p4:  ConsLet)  Dcls → Name Let Dcls
(p5:  Empty)    Dcls → ϵ
(p6:  Plus)     Expr → Expr Expr
(p7:  Minus)    Expr → Expr Expr
(p8:  Times)    Expr → Expr Expr
(p9:  Divide)   Expr → Expr Expr
(p10: Var)      Expr → Name
(p11: Const)    Expr → Number
```

The only difference between this grammar and the one presented in Chapter 4 (page 38) is in the production *p4*. The non-terminal *Dcls* has one production more, which states that a variable can now be assigned to a *Let* tree, instead of the assignments we have seen so far that would only allow assignments to expressions, as can be seen in the production *p3*.

This grammar can be implemented in `JastAdd` as:

```
Root ::= Let;
Let  ::= Dcls Expr;
abstract Dcls;
Cons     : Dcls ::= <Name : String> Expr Dcls;
Empty    : Dcls;
```

abstract Expr;
Plus : *Expr* ::= *left* : *Expr right* : *Expr*;
Minus : *Expr* ::= *left* : *Expr right* : *Expr*;
Times : *Expr* ::= *left* : *Expr right* : *Expr*;
Divide : *Expr* ::= *left* : *Expr right* : *Expr*;
Variable : *Expr* ::= <*Name* : *String*>;
Constant : *Expr* ::= <*Number* : *int*>;

In `JastAdd` we declare tokens (lexical terminals symbols) in square brackets, and we instantiate them as types in `Java`, in this case *int* and *String*. Apart from that, it follows a notation very close to the BNF notation seen previously for CFGs.

In order to construct a symbol table as we have seen in the previous chapter, we could write:

import *java . util.*∗;
aspect ConstructSymbolTable {
  syn HashMap < *String*, *Expr* > *Root.st* () = *getLet* () . *st* ();
  syn HashMap < *String*, *Expr* > *Let.st* () = *new HashMap* ();
}

We use `Java` data types, such as *HashMap*, to implement our intermediate data structures. `JastAdd` automatically generates *getNONTERMINAL* methods to access children. For example, we can see that for *Root* we can use a *getLet* method as *Let* is the name of a child (in this case, the only one).

As you can see, we can use ordinary `Java` types for the attribute types, and the right hand side of an equation is simply `Java` code. However, you have to make sure that any right-hand side has no externally visible side effects. By definition, attribute grammars are declarative, and no side effects are allowed in their specification. So if another equation would like to add something to an existing hash map attribute, you would have to clone it first.

One interesting note is that `JastAdd` supports the Extended Backus Naur Form (EBNF) for attribute grammars, which means we can declare lists in grammars using the Kleene star:

Let ::= *Dcls* ∗ *Expr*;

$Dcls ::= <Name : String > Expr;$

Here, the non-terminal *Let* contains a list of zero or more declarations (*Dcls∗*) and an expression (*Expr*). In these cases, `JastAdd` generated *Dcls* as an `Java` data type *List*, for which it generated the necessary functions such as *getDclsList*. This technique, a characteristic of `JastAdd`, facilitates searching through an AST as we have seen in the previous chapter: with canonical AGs our solution is to create an intermediate symbol table to carry information around.

In practice, this means we can declare name analysis on our AST as:

```
eq Root.getLet () . lookup (String var_name) {
  for (Dcls d : getDclsList ()) {
    String name = d . getName ();
    if (name ! = null)
      return d;
  }
  return null;
}
```

where instead of using explicit symbol tables, we define an attribute *lookup* that simply goes through the `JastAdd` generated *List* of declarations (*Dcls*) that is part of our AST. In this example, *getDclsList* is a generated method that returns a list on which we can define typical `Java` code.

In this example, *lookup* returns references to AST nodes when, in the list of declarations, we find a reference to a node whose name is equal to the argument of *lookup*.

This feature is not present in our embedding, but we are still able of defining references in attributes to increase the modularity of our AG solution, as we shall see in the next section.

4.3 Embedding Reference Attribute Grammars

We start by presenting the necessary extension on `LET` in order to support nested sub-expressions. As we have seen, syntactically the language does not change much. We only need to add a new construct to the data type *Dcls*:

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

with *ConsLet* representing nested blocks of code. The rest remains exactly as we presented in Chapter 2.

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

Typical solutions to this problem involve a complex algorithm where each block is traversed twice. This implies that for each inner block, a full traversal of the outer block is necessary to capture variable declarations. These are then used in the inner block together with a first traversal of the inner block to capture the total number of variables that are needed to check for scope/name rules. Only after the inner block is checked can the second traversal of the outer block be performed and only then can wrong declarations and use of identifiers be detected. The idiosyncrasies of implementing the analysis for nested blocks is further explained in previous work [Saraiva, 1999].

In order to be able to detect multiple declarationsof identifiers, we will need to know the level in which a variable is declared. We will therefore start by defyning a new attribute, *lev*, and its equations:

$$\begin{aligned}&\mathit{lev} :: \mathit{Zipper}\ a \to \mathit{Int}\\ &\mathit{lev}\ \mathit{ag} = \textbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \textbf{of}\\ &\quad \texttt{"Root"} \to 0\end{aligned}$$

```
"Let"  → case (constructor (parent ag)) of
           "ConsLet" → lev (parent ag) + 1
           _         → lev (parent ag)
_      → lev (parent ag)
```

The top of the tree and the main block will be at level 0. For *Let*, we have to inspect the parent node. If it is a *ConsLet*, we are in a nested block and we have to increment the level value. For all the other cases, we use a strategy that we have seen before: we use the wild card 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 produces more concise attribute grammars.

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.

References are implemented as zippers whose current focus is the site of the tree we want to reference, as we can see by the type of *dcli*:

```
dcli :: Zipper Root → [(String, Zipper Root)]
dcli ag = case (constructor ag) of
  "Root" → []
  "Let"  → case (constructor (parent ag)) of
             "Root"    → dcli (parent ag)
             "ConsLet" → env (parent ag)
  _      → case (constructor (parent ag)) of
             "Cons"    → dcli (parent ag)
                         ++ [(lexemeCons1 (parent ag)
                            , parent ag)]
             "ConsLet" → dcli (parent ag)
                         ++ [(lexemeConsLet1 (parent ag)
                            , parent ag)]
             "Empty"   → dcli (parent ag)
```

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

Using zippers as references, together with attributes being first-class citizens in the target language, means that we can re-define the semantic function *mNBIn* as:

$$
\begin{aligned}
&mNBIn :: (String, Zipper\ Root) \to [(String, Zipper\ Root)] \to Error \\
&mNBIn\ tuple\ [\,] \qquad\qquad\quad = [\,] \\
&mNBIn\ (a_1, r_1)\ ((a_2, r_2) : es) = \mathbf{if}\ (a_1 \equiv a_2) \wedge (lev\ r_1 \equiv lev\ r_2) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{then}\ [a_1] \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{else}\ mNBIn\ (a_1, r_1)\ es
\end{aligned}
$$

Now *mNBIn* checks the both is the variable name and if the declaration level match, extending scope rules to check for declarations only in the same scope, as duplicated 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 wish 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.

$$
\begin{aligned}
&errs :: Zipper\ Root \to Error \\
&errs\ ag = \mathbf{case}\ (constructor\ ag)\ \mathbf{of} \\
&\quad \texttt{"Root"} \quad\ \to errs\ (ag\ .\$\ 1) \\
&\quad \texttt{"Let"} \quad\ \to errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
&\quad \texttt{"Cons"} \quad \to mNBIn\ (lexemeCons_1\ ag, ag)\ (dcli\ ag) \\
&\qquad\qquad\quad +\!\!+ errs\ (ag\ .\$\ 2) +\!\!+ errs\ (ag\ .\$\ 3)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{"ConsLet"} \rightarrow mNBIn\ (lexemeConsLet_1\ ag, ag)\ (dcli\ ag) \\
&\qquad\qquad +\!\!+\ errs\ (ag\ .\$\ 2) +\!\!+ errs\ (ag\ .\$\ 3) \\
&\texttt{"Empty"} \rightarrow [\,] \\
&\texttt{"Plus"} \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
&\texttt{"Divide"} \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
&\texttt{"Minus"} \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
&\texttt{"Time"} \rightarrow errs\ (ag\ .\$\ 1) +\!\!+ errs\ (ag\ .\$\ 2) \\
&\texttt{"Variable"} \rightarrow mBIn\ (lexemeVariable_1\ ag)\ (env\ ag) \\
&\texttt{"Constant"} \rightarrow [\,]
\end{aligned}
$$

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.

With the embedded attribute grammar defined, we can execute the newly defined semantics with:

$$errs\ (to_{Zipper}\ program) \equiv [\,]$$

To summarize this section, references to non-local sites in the tree are represented by zippers whose focus is on the respective site. This capability together with attributes being first-class citizens in the embedding 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, Circular Attribute Grammars, are implemented in our setting.

4.4 Conclusions

In this chapter we have presented RAGs and provided implementations of this extension in our zipper-base setting.

In our setting we have used the contextual nature of zippers, where a structure always has a focus that specifies a certain position in the tree, to elegantly provide zippers with specific nodes as references to specific positions in a tree.

We extended our embedding of attribute grammars so that attributes are capable of being defined through the usage of references, and the semantics

we defined are more modular and easier to extend than the ones defined using canonical attribute grammars.

In the next chapter we will see further implementations of extensions of attribute grammars in our setting. In particular we will introduce, define and embed circular attribute computations.

Chapter 5

Circular Attribute Grammars

Summary

In this chapter we present a new type of attribute grammars, where circular computations are allowed, i.e., attributes can be defined as depending transitively on themselves. We will design and present an attribute grammar that, circularly, performs transformations and optimizations in the AST of `LET`*. This attribute grammar will first be presented as implemented in the AG system* `Kiama`*, a library for language processing that supports circularity, and then in our zipper-based setting, where we also introduce generic solutions to solve circularity.*

5.1 Introduction

In the canonical definition of AGs, the semantics implemented by attributes have to depend on other attributes that are positioned directly above of below the current node. In the previous chapter we have seen how this limitation can be leveraged by references, a setting in which attributes can now be defined as depending on semantics defined for random sites in an abstract tree. By doing so, we increased the expressiveness of attribute grammars by allowing the definition of attributes that, for example, allow graph structures to be defined.

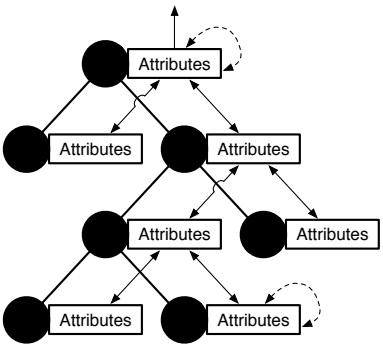


Figure 5.1: Overview of what we will define in this chapter. Attributes can now depende on themselves in circular computations.

In this chapter we will present a new extension for attribute grammars, where attributes can depend on themselves, allowing semantics to be defined on a recursive scheme. Such computations are only feasible when a certain fixed point can be reached, which defines a stopping point for an otherwise infinite and uncomputable algorithm. Figure 5.1 presents schematically this new extension comparing the the different ones we have seen in the previous chapters. This time, attributes can be implemented in circular computations, as can be seen by the dashed lines.

When attributes can be circularly defined in computations that have stopping conditions defined, a new group of problems become not only possible but easier to define. We will provide an example of such a problem, where we will define an AG capable of transforming, and therefore simplifying an AST of `LET`, creating a new tree which is easier to handle and to reason about.

Circularity will be introduced using the AG system `Kiama`, a library that supports various features for language processing such as tree rewritings, pretty printings or attribute grammars, in particular circular attribute computations. We will see how there can be defined in `Kiama`, after which we will define them in our zipper-based setting, on which we will not only define circular attributes, we will also present generic functions that handle circularity with minimum effort required by the user.

5.2 Circular Attribute Grammars

In the original definition of AGs, the definition of an attribute can depend directly only on attributes of neighbor nodes in the tree. Being them above or below the current node, they have to be direct neighbors. Furthermore, the dependencies between attributes may not be cyclic. This restriction on circular definitions on AGs is lifted by Circular Attribute Grammars (CAG), such as the ones presented by [Farrow, 1986] and [Jones, 1990].

Definition 14. (Circular Attribute Grammar)) An attribute grammar $AG = \langle G, A, D \rangle$ is called a circular attribute grammar if for every attribute tree generated by the AG, $DRT(T)$ is cyclic.

An attribute grammar is called an Circular Attribute Grammar if it has an attribute that depends transitively on itself. CAGs allow an attribute to have a circular dependency between other 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 attributes are defined by a monotonic computation that necessarily reaches a stopping condition, i.e., all circular attributes (attributes involved in a cyclic dependency) have a fixed point that can be computed with a finite number of iterations.

In this chapter we will consider `LET` with sub-expressions, as presented earlier, but we are now interested in computing the result of a program, instead of the name analysis for which we have already defined AGs. We will do so through circular attributes.

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

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

$$invalid = \textbf{let } x = y$$

$$
\begin{array}{rl}
y = \textbf{let} & a = b \\
& b = a \\
\textbf{in} & b \\
\textbf{in} & x + y + 1
\end{array}
$$

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

CAGs have multiple applications, and works such as [Farrow, 1986; Jones, 1990; Sasaki and Sassa, 2004] have shown how circular definitions in AGs provide simple specifications for problems from different areas such as code optimization and transformation or data flow analysis. In this chapter, we will see an example of an CAG implementation capable of performing the former.

One example of the utility of CAGs lies in handling abstract trees of the `LET` language, as a circular attribute computation on a AST of a `LET` program can be used to iteratively simplify the tree for further processing.

We have presented in Chapter 4 a version of `LET` with nested sub-expressions, and defined an AG capable of performing name analysis for this version of the language. However, we have never provided an AG capable of calculating the meaning of a program, i.e., calculating its resulting value.

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

Due to the complex nature of `LET`, with nested sub-expressions and a *declare-anywhere* discipline, the task of providing a meaning for the language is complex and involves multiple tree traversals. Therefore, in this section we will provide an AG that is capable of "flattening" a `LET` program[1].

[1]This step is not strictly necessary, as the meaning of `LET` can be calculated without this step. In Chapter 5 we will see a method of obtaining a meaning for this language without any "flattening".

By flattening we mean capable of transforming a complex AST into one where all variables are assigned to their partial result, not to an expression or a nested block, and where the only expression that needs to be solved is the one that provides the actual meaning of the language. For example, for the program *program*:

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

its "flattened" version is

$$
\begin{aligned}
\mathit{program} = \textbf{let}\ & a = 19 \\
& c = 8 \\
& w = 4864 \\
& b = 16 \\
\textbf{in}\ & c * w - a
\end{aligned}
$$

This process simplifies the computations for the calculation of the meaning of `LET`, by avoiding a potentially very complex AST to one which is simpler and where there is only one expression to be solved. Please note that the resulting "flattened" program is a (valid) program of the AG presented in Chapter 2.

If we were to compute the result of *program* by hand, one would need to start by computing the value of a and b in the outer scope, because the value of c is known. We would then need to compute c in the inner block, with the expression $a * b$, after which we know the value of w, and so on. We are using a fix-point computation to simplify *program*.

Next, we will see how circular attributes can be implemented in the AG system `Kiama`, through the implementation of a circular computation that "flattens" a `LET` program.

5.2.1 Circular Attribute Grammars in `Kiama`

`Kiama` [Sloane et al., 2010] is an embedding of AGs in the object oriented programming language `Scala` that provides a set of features similar to specific AG systems, such as cached, uncached, circular, higher-order and parameterized attributes. Similar to our embedding, `Kiama` Attribute Grammars use standard `Scala` notations and features, such as pattern-matching functions.

With `Kiama` being an embedding of AGs similar to the one we are presenting here, there are many similarities between these two settings. The AST for a `LET` program in `Kiama` is a good example, with an isomorphic relation to the embedding of grammars we have seen in `Haskell` in Chapter 3 (page 63).

In order to embed a grammar, we declare a set of classes that correspond to data types in `Haskell` (and non-terminals in the grammar), and a set of case classes that correspond to type constructors in `Haskell` (and productions in the grammar). For example, in:

case class *Root* (**let** : *Let*)
case class *Let* (*dcls* : *Dcls*, *expr* : *Expr*)

we define the non-terminals *Root* and *Let*, where *Root* has only one child, **let** of type *Let*, and *Let* has two children (or symbols in its right-hand side, if we see it as a grammar): *dcls* of type *Dcls* and *expr* of type *Expr*.

To define the list of declarations, we write:

sealed abstract **class** *Dcls*
case class *Cons* (*name* : *String*, *expr* : *Expr*, *dcls* : *Dcls*)
  extends Dcls
case class *ConsLet* (*name* : *String*, **let** : *Let*, *dcls* : *Dcls*)
  extends Dcls
case class *Empty* () *extends Dcls*

This corresponds to the non-terminal *Dcls*, which contains three productions. *Cons* has three children, *name* of type *String*, *expr* of type *Expr* and *dcls* of type *Dcls*. The production *ConsLet* is very similar to *Cons* but the second child has the name **let** and is of type *Let* and the third production, *Empty*, is an ϵ-production (contains no symbols on its right-hand side).

The declaration of expressions has a similar syntax:

```
sealed abstract class Expr
case class Plus     (left : Expr, right : Expr) extends Expr
case class Minus    (left : Expr, right : Expr) extends Expr
case class Times    (left : Expr, right : Expr) extends Expr
case class Divide   (left : Expr, right : Expr) extends Expr
case class Variable (name : String)             extends Expr
case class Constant (value : Int)               extends Expr
```

where we define the non-terminal *Expr* with 6 productions: *Plus*, *Minus*, *Times*, *Divide*, *Variable* and *Constant*.

Returning to CAGs, we can define the circular computation that goes through a tree and continually transforms it until a specific fixed point is reached. The circularity here is present in the form of an attribute that is repeatedly iterated, and therefore depends on itself as new iterations use the result of previous ones.

`Kiama` provides mechanisms for circular attribute computations, meaning that their creation is extremely simple. In particular, it provides a circular function that enables circular attributes to be defined. Its general form is:

```
val a : U ⇒ V =
  circular (v) {
    case... ⇒ ...
      ...
  }
```

where v (of type V) is the initial value of the circular attribute, and the cases are used to define values in terms of other attributes, that can potentially be circular.

For the particular case of "flattening" `LET` through a circular attribute, we can easily define a circular computation as:

```
def flattenLetFixedPoint (n : Root) : Root = {
  lazy val fix : Root ⇒ Root =
    circular (n) {
```

```
        case n ⇒ flatten (fix (n))
      }
    fix (n)
  }
```

The function *flatten* tries to "flat" the AST as much as possible in one iteration. The implementation of this attribute will be provided in the next section. For now it suffices to know its functionality, as the important part is that in

$$circular\ (n)\ \{...\}$$

(n) is the initial value of the circular computation, and

$$\textbf{case}\ n \Rightarrow flatten\ (fix\ (n))$$

is the computation of a step in reaching the fixed point.

This circular computation starts out having the unsolved AST n as the initial value, and the step function applies *flatten* to the tree in order to perform an iteration. When the result of *flatten* is equal to its original input, we are done and the fixed point was reached.

In the next section we will present an implementation of a similar circular computation using our zipper-based embedding.

5.3 Embedding Circular Attribute Grammars

We have described CAGs and provided a small example, implemented in the embedded AG setting `Kiama`, of a circular computation that simplifies an abstract tree of a `LET` program. In this section we will see a complete implementation of a circular computation using our zipper-based setting.

We have seen how `Kiama` provides a specific primitive for defining circular attributes. In our setting, we have defined a specific generic function that is capable of performing the same fixed point computations, called *fixed_point*:

$$\begin{aligned} fixed\_point :: Zipper\ a \to (Zipper\ a \to Bool) \to (Zipper\ a \to b) \\ \to (Zipper\ a \to Zipper\ a) \to b \end{aligned}$$

```
fixed_point ag cond calc incre =
  if     cond ag
  then calc  ag
  else  fixed_point (incre ag) cond calc incre
```

The arguments of this function are as follows:

- *ag* :: *Zipper a*: the tree on which we want to compute the fixed point computation.

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

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

  In the case of `LET`, for example, 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 any complex resolution because it only contains values.

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

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

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

Returning to our running example, we need to define a set of attributes that we will use as arguments to *fixed_point*. We shall start by defining the attribute *isSolved*, which checks for the termination of the circular computation:

```
isSolved :: Zipper Root → Bool
isSolved ag = case (constructor ag) of
  "Root"      → isSolved $ ag .$ 1
  "Let"       → isSolved $ ag .$ 1
  "Cons"      → (isConstant $ ag .$ 2) ∧ (isSolved $ ag .$ 3)
  "ConsLet"   → False
  "EmptyList" → True
```

This attribute checks if the AST is already solved, i.e., if all the attribute assignments are in their "flat" form, which is the same as saying that they are assigned directly to a constant. The interesting part of this attribute is its definition in the tree node *Cons*. Here we check if the right-hand side of the assignment, which corresponds to the second node, is a constant with the local attribute *isConstant*, and continue checking for the rest of the list. If we find a *ConsLet* we can return *False* right away as a "flattened" version of `LET` does not have nested expressions.

One important note is how local attributes can be easily implemented in our zipper-based setting as a small function. This is the case of *isConstant*, as can be seen below:

```
isConstant :: Zipper Root → Bool
isConstant ag = case (constructor ag) of
  "Constant" → True
  _ → False
```

The attribute that performs one iteration in this circular computation is called $flat_{AG}$. This attribute will traverse the tree once while constructing a new one with as many "flat" assignments as it can perform in one iteration. It is defined as:

```
flat_AG :: Zipper Root → Root
flat_AG ag = case (constructor ag) of
  "Root" → Root (flat_Let (ag .$ 1))
```

and for the tree node *Root* it simply rebuilds the top of the tree and calls a new attribute to continue the process:

```
flat_Let :: Zipper Root → Let
flat_Let ag = case (constructor ag) of
  "Let" → Let (flat_Dcls (ag .$ 1)) (lexemeLet_2 ag)
```

In the same way the process of flattening was extremely simple for the top production of the tree, it is equally simple for the production *Let*. As none of them have assignments, their only job is to recursively call an attribute that goes down the tree while rebuilding the tree nodes they are specified in.

Therefore, $flat_{Let}$ creates a new tree with the tree node *Let* and calls a new attribute $flat_{Dcls}$, that is implemented for the data type *Dcls* (please remember that we can see the data types in our embedding as non-terminals in a grammar, so we can equally say that $flat_{Let}$ is defined for the non-terminal *Dcls*). Its definition can be seen below:

```
flat_Dcls :: Zipper Root → Dcls
flat_Dcls ag = case (constructor ag) of
  "Cons" → if ¬ isConstant (ag .$ 2) ∧ isSolvable (ag .$ 2)
             then Cons lexemeCons_1 ag
                       Constant (calculate (ag .$ 2))
                       flat_Dcls (ag .$ 3)
             else Cons lexemeCons_1 ag
                       lexemeCons_2 ag
                       flat_Dcls (ag .$ 3)
  "ConsLet"    → if isSolved (ag .$ 2)
                   then Cons lexemeConsLet_1 ag
                             Constant (calculate (ag .$ 2))
                             flat_Dcls (ag .$ 3)
                   else ConsLet lexemeConsLet_1 ag
                                flat_Let (ag .$ 2)
```

$$flat_{Dcls}\ (ag\ .\$\ 3)$$

$$\texttt{"EmptyList"} \rightarrow \mathit{EmptyList}$$

The attribute $flat_{Dcls}$ is responsible for performing the actual "flattening" on variable assignments. For the tree node *Cons*, we have to check if 1) the assignment is not to a constant (is already "flat") and 2) if the expression is solvable. If this happens, we can reconstruct a tree with a "flatted" assignment using the attribute *calculate*, and continue recursively to the rest of the declarations list. If this condition does not hold, it means that either the assignment is already to a constant or the expression can not be solved, in which case we will have to wait for the next iteration of the circular computation. In this case, we just reconstruct the tree as it was.

The semantics for the tree node *ConsLet* are very similar to *Cons*. This time we do not need to check if the assignment is already "flat" as we are dealing with a nested expression, which is what we want to eliminate in he first place. The only condition we need to check is if the sub-expression is solved, in which case we can transform the current tree node into a *Cons*, again with the help of the attribute *calculate*. Otherwise we rebuild the tree as it was.

For tree node *EmptyList* we only rebuild the tree, as there is nothing to do here.

The attributes we have seen depend on a set of other attributes. The attributes *calculate* and *isSolvable* are extremely simple, with the former being defined as:

$$\begin{aligned}
&\mathit{calculate} :: \mathit{Zipper\ Root} \rightarrow \mathit{Int}\\
&\mathit{calculate}\ ag = \textbf{case}\ (\mathit{constructor}\ ag)\ \textbf{of}\\
&\quad \texttt{"Root"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1)\\
&\quad \texttt{"Let"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 2)\\
&\quad \texttt{"In"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1)\\
&\quad \texttt{"Plus"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1) + \mathit{calculate}\ (ag\ .\$\ 2)\\
&\quad \texttt{"Divide"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1)\ `\mathit{div}`\ \mathit{calculate}\ (ag\ .\$\ 2)\\
&\quad \texttt{"Minus"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1) - \mathit{calculate}\ (ag\ .\$\ 2)\\
&\quad \texttt{"Time"} \rightarrow \mathit{calculate}\ (ag\ .\$\ 1) * \mathit{calculate}\ (ag\ .\$\ 2)\\
&\quad \texttt{"Variable"} \rightarrow \mathit{getVarValue}\ (\mathit{lexemeVariable}_1\ ag)\ ag
\end{aligned}$$

```
"Constant" → lexemeConstant1 ag
```

The attribute *calculate* goes down the tree and applies the arithmetic operations necessary to compute the value of an expression, getting the values of variables as necessary with the attribute *getVarValue*. When this attribute is called, we are absolutely sure that all the assignments of the variables needed are already in their "flat" form, as we only call *calculate* after the condition represented by the attribute *isSolvable*. This attribute, which checks for the solvability of expressions, is presented next:

```
isSolvable :: Zipper Root → Bool
isSolvable ag = case (constructor ag) of
  "Plus"     → isSolvable (ag .$ 1) ∧ isSolvable (ag .$ 2)
  "Divide"   → isSolvable (ag .$ 1) ∧ isSolvable (ag .$ 2)
  "Minus"    → isSolvable (ag .$ 1) ∧ isSolvable (ag .$ 2)
  "Time"     → isSolvable (ag .$ 1) ∧ isSolvable (ag .$ 2)
  "Variable" → isVarSolved (lexemeVariable1 ag) ag
  "Constant" → True
```

The function *isSolvable* has semantics very similar to *calculate*: it goes down an expression to see if it contains constants (*Constant*), in which case it is solvable, or variables (*Variable*), in which case we have to see if this variable is solved with the attribute *isVarSolved*. The attributes *isvarSolved* and *getVarValue* can be defined using the same strategy used in Chapter 3, where we defined a series of interdependent computations that were capable of traversing the tree and capturing variables assignments. In the next chapter we will see different techniques to define them.

With these attributes defined, we are now in a position where we can use use the generic *fixed_point* function and "flatten" an AST of `LET`. Please recall that this function takes four arguments: our AG in the form of a zipper, a function that checks for termination, a function that is applied whenever the fixed point is reached, and a function that performs one iteration.

In our case, we use *fixed_point* as follows to successfully "flatten" a valid `LET` program.

$solve :: Zipper\ Root \rightarrow Zipper\ Root$
$solve\ ag = fixed\_point\ ag\ isSolved\ id\ (to_{Zipper} \circ flat_{AG})$

To resume the circular computation described in this section, what it does is:

1. Try to "flat" as much assigments as possible by checking if all the variables they depend on are already assigned to a constant,
2. Check if everything in the tree is "flat",
3. Try to flat one more time until `2.` holds,
4. Return the "flatted" tree.

Although we ommit all these attributes in the previous section, when we presented a `Kiama` version or a circular "flattening" attribute, they would be equally necessary. The exception is *isSolved* because, for the reasons we have seen, we have created a setting where a circular computation can be stopped through any user-provided stopping condition.

5.4 Conclusions

In this chapter we have seen a new extension with which we can further extend the way we define attribute grammars and the semantics they represent. With circularity presented, we can define powerful circular recursive computations where attribute depend only on themselves.

This methodology to solve problems using AG requires only that the user provides the method with which each step of the computation is performed, and a stopping point indicating when the stopping point is reached.

The approach we have seen is completely generic and can be potentially applied to any zipper-based AG defined in our environment, which shows how adaptable our setting is not only on defining AGs but also in defining non canonical, and more powerful forms of this formalism.

In the next chapter we will continue the extension of attribute grammars by introducing their higher order form.

Chapter 6

Higher Order Attribute Grammars

Summary

In this chapter we introduce a new extension to attribute grammars, higher order attribute grammars. In higher order attribute grammars attributes can be tree structures which are themselves attributable, allowing the anchoring and production of new trees. This extension will be introduced with the AG system LRC, and example implementations will be provided in out zipper-based embedding.

6.1 Introduction

In the previous chapters we have seen extensions to attribute grammars that allow attributes to have dependencies on random sites on an AST and allow them to be circularly defined. In this chapter we will see another extension that allows attributes to be trees that are themselves attributable.

In higher order attribute grammars attributes can be defined for the result of other attributes. In Figure 6.1 we can see the extension to AGs we are implementing in this chapter.

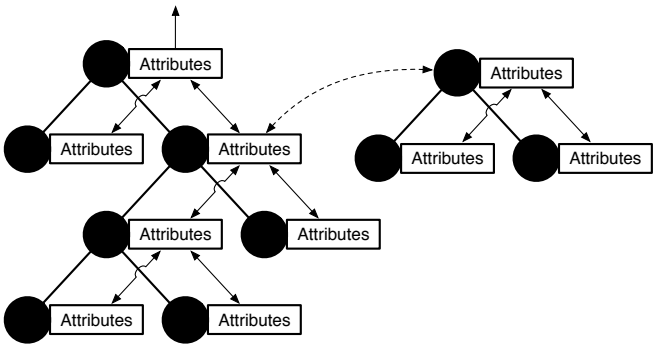


Figure 6.1: Overview of what we will define in this chapter. Attributes can now define new attributable trees.

We will introduce higher order in attribute grammars through the AG system `LRC`, a system that generates incremental language tools and has native support for higher order.

Similar to previous chapters, we will embed higher order using zippers on the functional language `Haskell` and present an example on how this extension can be used in the analysis of `LET`. We will do so by continuing the previous example and implementing an attribute grammar that calculates the result of a program, through higher order attributes. We will also show how the canonical solutions from previous section can be rewritten using this extension.

6.2 Higher Order Attribute Grammars

So far we have seen definitions of attribute grammars where the underlying AST is fix (does not change) during attribute evaluation.

While is true complex computations can be defined using this formalism, a power which is further extended through the addition of references as we have seen in the previous chapter, truth is attributes are completely constrained by their defining equations. This is not true for the abstract tree that defines the syntax.

This limitation means that the attributes have no way of generating trees, which can for example represent intermediate steps in a language-

transformation environment. The formalization of higher order attribute grammars provides a basis for addressing the limitations of the canonical, first order attribute grammars we have seen so far.

Higher Order Attribute Grammars (HOAGs) were first introduced by Vogt *et al.* and introduce a setting where the structure tree can be expanded as the result of attribute computations [Vogt et al., 1989]. The new parts of the tree can themselves have semantics defined using attributes. Since they allow multiple tree transformations through higher order attributes, they have seen use, for example, in editing environments [Teitelbaum and Chapman, 1990].

Comparing with traditional AGs, HOAGs provide a setting where:

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

In [Vogt et al., 1989], the semantics of higher order attribute grammars are expressed by transforming them into a first class AG. Next, we present an HOAG written in LRC. LRC follows Vogt approach by transforming on HOAG into a classical attribute grammar, before it produces the corresponding evaluator.

6.2.1 Higher Order Attribute Grammars in LRC

To demonstrate how HOAGs can be used, we will use LRC [Kuiper and Saraiva, 1998]. LRC is a graphical language-oriented tool that accepts as input higher order attribute grammars, specifying a language, and generates attribute generators from them. The underlying machinery of this tool possesses efficient incremental evaluators, which use function caching and several other optimizations. More information regarding these evaluators and the techniques it uses can be found in [Pennings et al., 1992; Carle and Pollock, 1996; Saraiva et al., 1997].

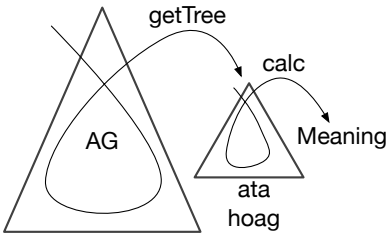


Figure 6.2: Higher order attribute grammars in LRC.

In LRC grammars are defined using a specific notation, which closely resembles the BNF notation for the abstract CFGs we have seen in Chapter 2. In particular, we can define the AST for LET defined in the same chapter (page 38) in LRC as:

```
Root : Root (Let)
     ;
Let  : Let (Dcls Expr)
     ;
Dcls : Cons  (Name Expr Dcls)
     | Empty ()
     ;
Expr : Plus      (Expr Expr)
     | Minus     (Expr Expr)
     | Times     (Expr Expr)
     | Divide    (Expr Expr)
     | Variable  Name
     | Constant Number
     ;
```

A grammar defined in LRC is so similar to one defined using BNF that any explanation is unnecessary. The only main different are the non-terminals *Name* and *Number*, which are strings and numbers respectively, and that we have to define in LRC as:

```
Name : Var (STR)
     ;
Number : Const (INT)
```

;

where we provide instructions to lexe these two symbols, providing type informations relating *Name* to a string (*STR*) and *Number* to a number (*INT*).

Creating an higher order attribute in `LRC` implies writing the following AG equations:

1. Declare a new *ata* (attributable attribute), and provide its type. This is achieved with:

   ata Let hoag;

   where we define a new attribute *hoag*, which is a tree of type *Let*.

2. Assign this new attribute, *hoag*, to an older attribute that generate the tree which will become our new AG. To do so, we write:

   hoag = *AG* . *getTree*;

   where we are assigning *hoag* to the information produced by the attribute *getTree*, previously defined on AG.

3. Finally, we can ask for any attribute on our new HOAG *hoag* (which is why we defined it as an attributable attribute), such as:

   result = *hoag* . *calc*;

   where we ask for the computation of the previously defined attribute *calc*.

The whole process is resumed in Figure 6.2: an attribute, *getTree*, generated a new attributable attribute, *hoag*, where new attributes can be defined, in this case, *calc*. Although the example we provided recreates a process where only one new tree is created, this process could have multiple steps with multiple HOAGs representing, for example, multiples phases of a compiler.

In the next section we will see how this new type of attributes can be embedded in our zipper-based setting.

6.3 Embedding Higher Order Attribute Grammars

In our zipper-based embedding of AGs, HOAGs are easy to implement. As we can take an `Haskell` data type, wrap it inside a zipper and define attributes for it, we can easily take an attribute that produces a tree, wrap it immediately inside a zipper and implement attribute for it.

Retuning to the circular example of the previous section, we defined an attribute capable of simplifying a `LET` program, by reducing all the variable assignments to ones that contain only a number on their right-hand sides. As the reader may recall, this computation produced a new, simplified tree, but the meaning of it was never defined.

Defining an HOAG that takes the "flattened" version of the tree and calculates its meaning is extremely simple and straightforward, as it only implies taking a tree that was the result tree of an attribute and provide new attributes for it. With this said, we already defined an attribute that is capable of taking a "flat" tree and providing its meaning: *calculate*, which can be seen in the previous chapter (page 99).

The attribute *calculate* is capable of reducing a nested expression to its numeric value (this is precisely what "flattening" is). Therefore, we can simply take the result of the circular computation from the previous chapter, and apply this attribute to its result:

$$
\begin{aligned}
&\mathit{meaning} :: \mathit{Zipper\ Root} \to \mathit{Int} \\
&\mathit{meaning}\ \mathit{ag} = \mathbf{let}\ \mathit{ata} = \mathit{solve}\ \mathit{ag} \\
&\qquad\qquad\quad \mathbf{in}\ \ \mathit{calculate}\ \mathit{ata}
\end{aligned}
$$

Here, we are circularly transforming the tree with *solve ag* and creating an higher order attribute (*ata*), which we solve with the attribute *calculate*.

These simple piece of code is actually taking a whole tree and providing, through *calculate*, new semantics to the result of an attribute. This shows the simplicity of both defining high order attributes and creating interactions between different AG extensions in our setting. We created a high order attribute that is produced using information provided by a circular attribute

because it was useful, and this apparently complex scheme is actually a simple way of solving this problem. Applying *meaning* to our original *program* of Chapter 4 (page 78) yields the expected result:

$$meaning\ program = 38893$$

Please note that the circular lazy solution presented in Section 1.4 would need a major restructuring to support this computation. In our setting we just create an higher order attribute and defined a simple zipper AG to implement the meaning of a program.

6.3.1 Semantic Functions and Higher Order Attributes

One practical application of HOAGs is for expressing recursive functions as AG computations. Please recall the AG presented in Chapter 3. The attributes *dcli*, *dclo* and *env* were responsible for capturing variables declarations, which were stored as simple lists in `Haskell` and, through two semantic function, analysis of whether a variable was declared twice or was used but not declared were performed with two `Haskell` functions, *mBIn* and *mNBIn* (Section 1.4).

The problem with this approach is that although we are using AGs as a technique to avoid complex semantic functions, where scheduling and intermediate data types have to be taken into account, we still resort to them for secondary computations, as is the case with *mBIn* and *mNBIn*. The example provided is simple, but it does not scale well if we need to carry around more information in the environment, such as variable types or memory allocation information (if we are using vectors).

One solution is to transform the list of variables into a tree that we can carry around, a solution where this tree is produced and carried by the attributes *dcli*, *dclo* and *env*. This way, whenever we need to perform name analysis we can do so with attributes for this newly defined tree structure, to which we call an higher order attribute grammar.

One interesting application of HOAGs is in the function that provides errors to the user. So far, the solution we have seen in Chapter 3 only creates

a list of variables, which neither provides any kind of contextual information nor indicates if the problem is a duplicated declaration or a variable being used but not declared.

Next we present a new solution for the name analysis of Chapter 3, where we will provide contextual information to variables being used but not declared, and indicate when a variable was declared twice. For example, for the program:

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

our new system will produce the result:

```
Variable - z - undeclared.
Variable - a - declared twice.
```

We start by defining the data types that will carry the variables information around the tree, in an AG fashion:

$$
\begin{aligned}
\mathbf{data}\ \mathit{VarList} = {} & \mathit{VarList}\ \mathit{VarList}\ \mathit{String} \\
\mid {} & \mathit{NoVar}
\end{aligned}
$$

This data type is a traditional recursive definition of a list, that will now be used as our environment. It contains two constructors: *VarList* to store information about declared variables, and *NoVar*, for an empty list of variables.

Next, we need to redefine the attribute *dcli*. This attribute captures all the variable assignments in the program, and its new definition is presented next:

$$
\begin{aligned}
& \mathit{dcli} :: \mathit{Zipper}\ \mathit{Root} \rightarrow \mathit{VarList} \\
& \mathit{dcli}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of} \\
& \quad \texttt{"Root"} \rightarrow [\,] \\
& \quad \_ \qquad \rightarrow \mathbf{case}\ (\mathit{constructor}\ (\mathit{parent}\ \mathit{ag}))\ \mathbf{of} \\
& \qquad\qquad \texttt{"Cons"} \rightarrow \mathit{VarList}\ (\mathit{dcli}\ (\mathit{parent}\ \mathit{ag}))
\end{aligned}
$$

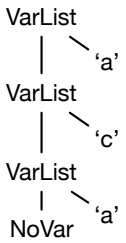


Figure 6.3: Intermediate structure representing an environment. This will be the equivalent to the attributable attribute (*ata*) that we saw in the previous section on `LRC`.

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad (dcli\ (ag\ .\$\ 3)) \\ \texttt{"Empty"} \rightarrow dcli\ (parent\ ag) \end{array}$$

For the attributes *dclo* and *env* only their return type changes, from [*String*] to *VarList*. As these two attributes do not generate our intermediate tree, they just carry it around (please recall Figure 2.6, on page 54). For the program *faulty*, the final attribute *env* would represent a tree containing the information that can be see in Figure 6.3.

We now need to update the semantic functions *mBIn* and *mNBIn*, so they produce textual information instead of the simpler list of variables they used to produce. With their reimplementation, we will change *mBIn* and *mNBIn*, so these will become attributes on an higher order attribute grammar of type *VarList*, which corresponds to the tree produced by the attributes *dcli*, *env* and *dclo*. Therefore, *mBIn* is implemented as:

$$\begin{array}{l} mBIn :: Zipper\ VarList \rightarrow String \rightarrow String \\ mBIn\ ag\ name = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\ \quad \texttt{"VarList"} \rightarrow \textbf{if}\ (lexemeVarList_2\ ag \equiv name) \\ \qquad\qquad\quad \textbf{then}\ \texttt{""} \\ \qquad\qquad\quad \textbf{else}\ mBIn\ (ag\ .\$\ 1)\ name \\ \quad \texttt{"NoVar"} \quad \rightarrow \texttt{"Variable - "} +\!\!+ name +\!\!+ \texttt{" - undeclared"} \end{array}$$

If a variable must be in (*mBIn*) the environment and we can not find it, it means it was undeclared. The attribute *mNBIn* suffers a similar upgrade:

$$\begin{array}{l} mNBIn :: Zipper\ VarList \rightarrow String \rightarrow String \\ mNBIn\ ag\ name = \textbf{case}\ (constructor\ ag)\ \textbf{of} \end{array}$$

```
"VarList" → if (lexemeVarList_2 ag ≡ name)
            then "Variable - " ++ name ++ " - declared twice."
            else mNBIn (ag .$ 1) name
"NoVar"   → ""
```

only this time we are checking when a variable must not be in (*mNBIn*) the environment, which will mean it was declared twice.

Finally, the textual information produced has to be concatenated and presented to the user, which is done in the attribute *errs*:

```
errs :: Zipper Root → String
errs ag = case (constructor ag) of
  "Root"     → errs (ag .$ 1)
  "Let"      → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Cons"     → let s1 = let ata = to_Zipper (dcli ag)
                        in  mNBIn ata (lexemeCons_1 ag)
                 s2 = errs (ag .$ 2)
                 s3 = errs (ag .$ 3)
             in  s1 ++ s2 ++ s3
  "Empty"    → ""
  "Plus"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Divide"   → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Minus"    → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Time"     → errs (ag .$ 1) ++ errs (ag .$ 2)
  "Variable" → let ata = to_Zipper (env ag)
               in  mBIn (lexemeVariable_1 ag) ata
  "Constant" → ""
```

In this attribute, all the information captured is transformed and concatenated into a string, which represents the final, textual information the user will see. The interesting part is that the attributes *dcli*, *dclo* and *env* produce a tree, but our setting requires a tree to be wrapped inside a zipper to be considered an AG and to be attributed. Therefore, in the line:

```
s1 = let ata = to_Zipper (dcli ag)
     in  mNBIn ata (lexemeCons_1 ag)
```

we need to create a new zipper, to which we call *ata* as it will work as an HOAG on which we can run the attribute *mBIn*. The exact same idea has to be applied to the production *Variable*.

6.4 Conclusions

In this chapter we have seen how higher order attribute grammars can be implemented in our zipper-based setting. Higher order allows definitions of attribute grammars in which attributes can produce results which are themselves attribute grammars.

In our setting, creating an attribute grammar means wrapping structures inside a zipper and creating functions that use contextual information from that zipper to specify semantics. In practice, this means an higher order grammar is obtained just by wrapping the result of an attribute inside a zipper and defining the supporting functions, making it a powerful extension that is extremely easy to implement.

In the next chapter we will see how our setting allows the combination of higher order and circularity.

Chapter 7

Circular and Higher Order Attribute Grammars

Summary

In this chapter we introduce one technique of defining semantics for a language where we define an higher order attribute containing a circular computation. This solution shows how we can combine two of the AG extensions we have presented so far and define semantics where their usage is intermingled in the final solution.

7.1 Introduction

In this chapter we will present an example of how circular and higher order attribute grammars can be combined. We will use our running example to define yet another approach to provide a meaning for a `LET` program. This solution does not require the tree "flattening" operation we have seen in Chapter 5.

In the solution we will present, we will start with an AST of a program on which we implement an attribute which is at the same type circular and of higher order. In particular, we will construct an higher-order symbol table of a `LET` program and circularly solve it until we can easily extract a meaning

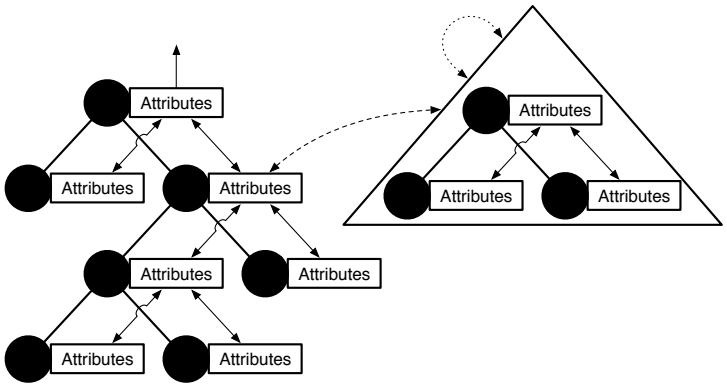


Figure 7.1: Overview of what we will define in this chapter. In our setting we can apply circular computations on higher order attributes.

from it. This is presented in Figure 7.1. As we see, we are capable of defining higher order attribute on which we apply the circular computations seen in Chapter 5.

Defining a circular attribute for tree transformation is not new. In [Söderberg and Hedin, 2013] the authors investigate how circular attributes can interact with tree rewriting, while creating well defined AGs. A generalized evaluation algorithm that can handle grammars with interdependent circular attributes and tree rewrites is also presented, a technique that the authors define as being a particular case of circular RAGs.

In the definition we will present we will not use references, but we will circularly construct and reconstruct an higher order attribute grammar in order to define the semantics of `LET`.

7.2 A Symbol Table as an Higher Order Attribute

In this section we will see another technique for defining the semantics for `LET`, but this time we will do so without going through intermediate "flattening" steps. We will define an auxiliary structure, more precisely a symbol table, which will be modeled as an HOAG where its lookup operations will be defined as attributes.

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

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

$$
\begin{aligned}
&\textbf{data } \mathit{Root}_{HO} = \mathit{Root}_{HO}\ \mathit{Dcls}_{HO}\ \mathit{Expr}\\
&\textbf{data } \mathit{Dcls}_{HO} = \mathit{ConsHO} \quad \mathit{String} \quad \mathit{IsSolved}\ \mathit{Expr} \quad \mathit{Dcls}_{HO}\\
&\qquad\qquad \mid \mathit{ConsLet}_{HO} \quad \mathit{String} \quad \mathit{IsSolved}\ \mathit{Dcls}_{HO}\ \mathit{Dcls}_{HO}\\
&\qquad\qquad \mid \mathit{NestedDcls}_{HO}\ \mathit{Dcls}_{HO}\ \mathit{Expr}\\
&\qquad\qquad \mid \mathit{Empty}_{HO}\\
&\textbf{data } \mathit{IsSolved} = \mathit{IsSolved}\ \mathit{Int} \mid \mathit{NotSolved}
\end{aligned}
$$

These three 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, *ConsHO* and $\mathit{ConsLet}_{HO}$, 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. The data constructor $\mathit{NestedDcls}_{HO}$ 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:

$$
\begin{aligned}
&createST_{Root} :: Zipper\ Root \rightarrow Root_{HO} \\
&createST_{Root}\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
&\quad \texttt{"Root"} \rightarrow Root_{HO}\ (createST\ ag)\ (lexemeLet_2\ (ag\ .\$\ 1))
\end{aligned}
$$

Here, the first argument of $Root_{HO}$ is the attribute that creates the symbol table and $lexemeLet_2\ (ag\ .\$\ 1)$ extracts the expression that constitutes the meaning of this program. Please recall that the abstract tree for `LET` has the form:

and therefore to extract the top level expression we have to go to the first child of *Root*, a *Let*, and then we can apply the *lexeme* function to the second child, *Expr*, which is why we write $lexemeLet_2\ (ag\ .\$\ 1)$.

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

$$
\begin{aligned}
&createST :: Zipper\ Root \rightarrow Dcls_{HO} \\
&createST\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
&\quad \texttt{"Root"} \quad \rightarrow createST\ (ag\ .\$\ 1) \\
&\quad \texttt{"Let"} \quad \rightarrow createST\ (ag\ .\$\ 1) \\
&\quad \texttt{"Cons"} \quad \rightarrow \textbf{let}\ var\ \ = lexemeCons_1\ ag \\
&\qquad\qquad\qquad\quad\ \ expr = lexemeCons_2\ ag
\end{aligned}
$$

```
                  in  ConsHO var NotSolved expr (createST (ag .$ 3))
"ConsLet" → let var = lexemeConsLet_1 ag
                  nested = let nested = createST      (ag .$ 2)
                               expr   = lexeme_In_1 ((ag .$ 2) .$ 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.

With $createST_{Root}$ and *createST* defined we have now created a new tree on which attributes can be defined. The new 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 previous chapters. 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
  "RootHO"        → auxIsVarSolved name ag
  "NestedDclsHO" → auxIsVarSolved name ag
  _               → isVarSolved      name (parent ag)
```

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

```
auxIsVarSolved :: String → Zipper Root_HO → Bool
auxIsVarSolved name ag = case (constructor ag) of
  "RootHO"        → auxIsVarSolved name (ag .$ 1)
```

```
"NestedDclsHO" → auxIsVarSolved name (ag .$ 1)
"ConsHO"       → if   lexeme_ConsHO_Var_1 ag ≡ name
                 then auxIsVarSolved name (ag .$ 2)
                 else auxIsVarSolved name (ag .$ 4)
"ConsLetHO"    → if   lexeme_ConsLetHO_Var_1 ag ≡ name
                 then auxIsVarSolved name (ag .$ 2)
                 else auxIsVarSolved name (ag .$ 4)
"IsSolved"     → True
"NotSolved"    → False
"EmptyHO"      → oneUpIsVarSolved name ag
```

The synthesized *auxIsVarSolved* attribute goes up 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
  "NestedDclsHO" → isVarSolved       name (parent ag)
  _              → oneUpIsVarSolved name (parent ag)
```

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

One important note about the three attributes *isVarSolved*, *auxIsVarSolved* and *oneUpIsVarSolved* is their interdependence. The first two, *isVarSolved* and *auxIsVarSolved*, search for a variable in a block, with the former going to the topmost position of a block and the latter going top-down in search of the variable. In case nothing is found, *oneUpIsVarSolved* goes up one block. The relation between these three attributes is illustrated in Figure 7.2.

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.

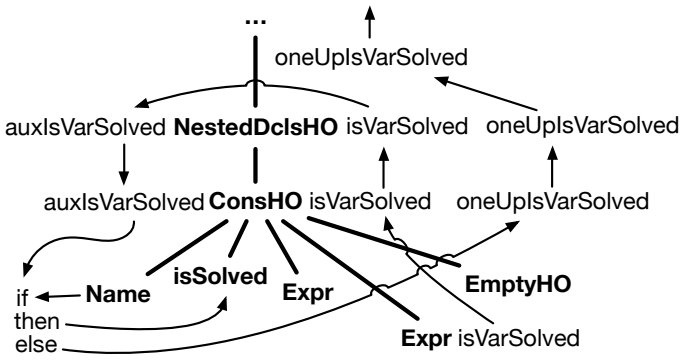


Figure 7.2: Dependency between *isVarSolved*, *auxIsVarSolved* and *oneUpIsVarSolved*.

$$
\begin{aligned}
&\mathit{getVarValue} :: \mathit{String} \to \mathit{Zipper}\ \mathit{Root}_{HO} \to \mathit{Int}\\
&\mathit{getVarValue}\ \mathit{name}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of}\\
&\quad \texttt{"RootHO"} \to \mathit{auxGetVarValue}\ \mathit{name}\ \mathit{ag}\\
&\quad \texttt{"NestedDclsHO"} \to \mathit{auxGetVarValue}\ \mathit{name}\ \mathit{ag}\\
&\quad \_ \to \mathit{getVarValue}\ \mathit{name}\ (\mathit{parent}\ \mathit{ag})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{auxGetVarValue} :: \mathit{String} \to \mathit{Zipper}\ \mathit{Root}_{HO} \to \mathit{Int}\\
&\mathit{auxGetVarValue}\ \mathit{name}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of}\\
&\quad \texttt{"RootHO"} \to \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 1)\\
&\quad \texttt{"NestedDclsHO"} \to \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 1)\\
&\quad \texttt{"ConsHO"} \to \mathbf{if}\ \ \mathit{lexeme\_ConsHO\_Var\_1}\ \mathit{ag} \equiv \mathit{name}\\
&\qquad\qquad \mathbf{then}\ \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 2)\\
&\qquad\qquad \mathbf{else}\ \ \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 4)\\
&\quad \texttt{"ConsLetHO"} \to \mathbf{if}\ \ \mathit{lexeme\_ConsLetHO\_Var\_1}\ \mathit{ag} \equiv \mathit{name}\\
&\qquad\qquad \mathbf{then}\ \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 2)\\
&\qquad\qquad \mathbf{else}\ \ \mathit{auxGetVarValue}\ \mathit{name}\ (\mathit{ag}\ .\$\ 4)\\
&\quad \texttt{"IsSolved"} \to \mathit{lexeme\_IsSolved\_1}\ \mathit{ag}\\
&\quad \texttt{"EmptyHO"} \to \mathit{oneUpGetVarValue}\ \mathit{name}\ \mathit{ag}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{oneUpGetVarValue} :: \mathit{String} \to \mathit{Zipper}\ \mathit{Root}_{HO} \to \mathit{Int}\\
&\mathit{oneUpGetVarValue}\ \mathit{name}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of}\\
&\quad \texttt{"NestedDclsHO"} \to \mathit{getVarValue}\ \ \mathit{name}\ (\mathit{parent}\ \mathit{ag})\\
&\quad \_ \to \mathit{oneUpGetVarValue}\ \mathit{name}\ (\mathit{parent}\ \mathit{ag})
\end{aligned}
$$

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

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

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

7.3 Circularity in Higher Order Attributes

We will now show how a symbol table can be resolved using circular, fixed-point based computation. To do so, we have to define the attributes that will be used as arguments of *fixed_point*, starting with the attribute that ends the circular computation by defining the fixed point (*cond*):

$$
\begin{aligned}
&\mathit{isSolved} :: \mathit{Zipper}\ \mathit{Root}_{HO} \rightarrow \mathit{Bool}\\
&\mathit{isSolved}\ \mathit{ag} = \mathbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathbf{of}\\
&\quad \texttt{"RootHO"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \vee \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2)\\
&\quad \texttt{"NestedDclsHO"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1)\\
&\quad \texttt{"ConsHO"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 4)\\
&\quad \texttt{"ConsLetHO"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 4)\\
&\quad \texttt{"EmptyHO"} \rightarrow \mathit{True}\\
&\quad \texttt{"IsSolved"} \rightarrow \mathit{True}\\
&\quad \texttt{"NotSolved"} \rightarrow \mathit{False}\\
&\quad \texttt{"Plus"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2)\\
&\quad \texttt{"Divide"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2)\\
&\quad \texttt{"Minus"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2)\\
&\quad \texttt{"Time"} \rightarrow \mathit{isSolved}\ (\mathit{ag}\ .\$\ 1) \wedge \mathit{isSolved}\ (\mathit{ag}\ .\$\ 2)\\
&\quad \texttt{"Variable"} \rightarrow \mathit{isVarSolved}\ (\mathit{lexemeVariable}_1\ \mathit{ag})\ \mathit{ag}\\
&\quad \texttt{"Constant"} \rightarrow \mathit{True}
\end{aligned}
$$

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 $solveST_{Root}$, which together with *solveST* performs one iteration of the fixed point computation, solving as many variables as possible.

$$
\begin{aligned}
&solveST_{Root} :: Zipper\ Root_{HO} \to Zipper\ Root_{HO} \\
&solveST_{Root}\ ag = \mathbf{let}\ solved\_decl = solveST\ (ag\ .\$\ 1) \\
&\qquad\qquad\qquad\qquad\ \ top\_expr\ \ = lexeme\_RootHO\ ag \\
&\qquad\qquad\qquad\quad \mathbf{in}\ to_{Zipper}\ (Root_{HO}\ solved\_decl\ top\_expr)
\end{aligned}
$$

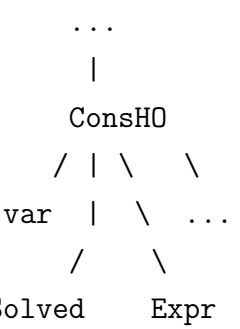
In this definition the topmost expression is ignored and *solveST* tries to solve the declarations. (If the meaning expression only contains constants *isSolved* will notice and terminate the fixed point computation before $solveST_{Root}$ is called.)

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

$$
\begin{aligned}
&solveST :: Zipper\ Root_{HO} \to Dcls_{HO} \\
&solveST\ ag = \mathbf{case}\ (constructor\ ag)\ \mathbf{of} \\
&\quad \texttt{"ConsHO"} \to \\
&\qquad \mathbf{if}\quad (\neg\ isSolved\ (ag\ .\$\ 2) \wedge isSolved\ (ag\ .\$\ 3)) \\
&\qquad \mathbf{then\ let}\ var\ \ = lexemeConsHO_1\ ag \\
&\qquad\qquad\qquad\ \ res\ \ = IsSolved\ (calculate\ (ag\ .\$\ 3)) \\
&\qquad\qquad\qquad\ \ expr = lexemeConsHO_A\ ag \\
&\qquad\qquad\ \mathbf{in}\ \ ConsHO\ var\ res\ expr\ (solveST\ (ag\ .\$\ 4)) \\
&\qquad \mathbf{else}\ \ \mathbf{let}\ var\ \ = lexemeConsHO_1\ ag \\
&\qquad\qquad\qquad\ \ res\ \ = lexemeConsHO_2\ ag \\
&\qquad\qquad\qquad\ \ expr = lexemeConsHO_3\ ag \\
&\qquad\qquad\ \mathbf{in}\ \ ConsHO\ var\ res\ expr\ (solveST\ (ag\ .\$\ 4) \\
&\quad \texttt{"ConsLetHO"} \to \\
&\qquad \mathbf{if}\quad (\neg\ isSolved\ (ag\ .\$\ 2) \wedge isSolved\ (ag\ .\$\ 3))
\end{aligned}
$$

```
        then let var  = lexemeConsLetHO_1 ag
                 res  = IsSolved (calculate (ag .$ 3))
                 expr = lexemeConsLetHO_3 ag
             in  ConsLet_HO var res expr (solveST (ag .$ 4))
        else let var    = lexemeConsLetHO_1 ag
                 solved = lexemeConsLetHO_2 ag
                 newST  = let newST = solveST (ag .$ 3)
                              expr  = lexemeNestedDclsHO_2 (ag .$ 3)
                          in  NestedDcls_HO newST expr
             in  ConsLet_HO var solved newST (solveST (ag .$ 4))
    "EmptyHO"       -> Empty_HO
    "NestedDclsHO" -> solveST (ag .$ 1)
```

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



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

solveST works as follows:

1. First check if the variable is not solved but if its expression/nested block is solved (all the variables it uses are solved). This is performed with the line $\neg$ *isSolved* (*ag* .$ 2) $\wedge$ *isSolved* (*ag* .$ 3).

2. If the condition holds, we can solve the variable, which means we *calculate* (defined below) the value of either the expression or the nested block and update the constructor to *isSolved*.

3. If the condition does not hold, we cannot do anything yet, so we will reconstruct this part of tree exactly as we read it.

    - If we are dealing with a variable defined by a nest block, we will try to see if any nested definitions can be solved, by calling *solveST* in the nested block: $solveST\ (ag\ .\$\ 3)$

4. The attribute always ends by going to the next declaration, which corresponds to the fourth child: $solveST\ (ag\ .\$\ 4)$

With the attributes *isSolved* and $solveST_{Root}$ 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:

$$
\begin{aligned}
&calculate :: Zipper\ Root_{HO} \rightarrow Int \\
&calculate\ ag = \textbf{case}\ (constructor\ ag)\ \textbf{of} \\
&\quad \texttt{"RootHO"} \quad\quad \rightarrow calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"NestedDclsHO"} \rightarrow calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"Plus"} \quad\quad \rightarrow calculate\ (ag\ .\$\ 1) + calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"Divide"} \quad\quad \rightarrow calculate\ (ag\ .\$\ 1)\ /\ \ calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"Minus"} \quad\quad \rightarrow calculate\ (ag\ .\$\ 1) - calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"Time"} \quad\quad \rightarrow calculate\ (ag\ .\$\ 1) * \ calculate\ (ag\ .\$\ 2) \\
&\quad \texttt{"Variable"} \quad\quad \rightarrow getVarValue\ (lexemeVariable_1\ ag)\ ag \\
&\quad \texttt{"Constant"} \quad\quad \rightarrow lexemeConstant_1\ ag
\end{aligned}
$$

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

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

$$
\begin{aligned}
&solve :: Zipper\ Root \rightarrow Int \\
&solve\ ag = \textbf{let}\ ho\_st = to_{Zipper}\ (createST_{Root}\ ag) \\
&\qquad\qquad \textbf{in}\ \ fixed\_point\ ho\_st\ isSolved\ calculate\ solveST_{Root}
\end{aligned}
$$

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.

7.4 Conclusions

In this chapter we have presented one more technique that we can use in order to define semantics for the language `LET`. This technique does not imply using features except the ones already presented in the previous chapter, namely circularity and higher order.

Because we are embedding AGs in the host language `Haskell` and our attributes are first-class citizens, we are capable on interchanging our solutions between them and also using other `Haskell` features. This is what we see in this solution, were we create an higher-order attribute in which we can apply an `Haskell` function, in this case our solution to implement circularity.

In the next chapter we will see a new feature that provides bidirectional transformations between grammars, which are implemented as AGs and can be embedded using our approach.

Chapter 8

Bidirectional Attribute Grammars

Summary

In this chapter we show how rewrite rules (with non-linear right hand sides) that specify a forward/get transformation can be inverted to specify a partial backward/put transformation. These inverted rewrite rules can then be extended with additional rules based on characteristics of the source language grammar and forward transformations to create, under certain circumstances, a total backward transformation. Finally, these rules are used to generate attribute grammar specifications implementing both transformations.

8.1 Introduction

In this chapter we will see how rewrite rules are used to define transformation specifications and how these can be automatically inverted to generate a backward transformation in the opposite direction. We will also see different techniques to extend and empower these transformations and for generating the attribute grammar equations that implement the them. The work presented in this chapter can be found in the following publication: [Martins

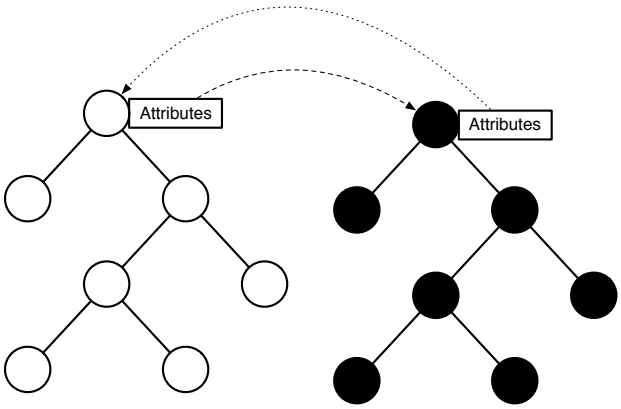


Figure 8.1: Overview of what we will define in this chapter. We will define transformation specifications that provide automatic transformation between grammars.

et al., 2014b].

We use a simple but common example to describe our approach to generating bidirectional transformation for attribute grammars. The techniques described here are generalized and extended throughout this chapter to handle more complex and interesting cases.

This is schematically represented in Figure 8.1. The transformation specifications we will present in this chapter will be capable of defining attribute grammars that can transform between different grammars, on which we can define attributes. In the case of our embedding, this will imply the generation of zipper-based attributes capable of transforming between `Haskell` data types on which we can define attributes using the techniques we presented in the previous chaptes.

There have been various works on this area, as data transformations are an active research topic with multiple strategies applied on various fields, some with a particular emphasis on rule-based approaches. Czarnecki and Helsen [Czarnecki and Helsen, 2006] present a survey of such techniques, but while they mention bidirectionality, they do not focus on it.

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 [Jouault and Kurtev, 2006]. BOTL

[Hibberd et al., 2007], an object-oriented transformation language, defines a relational approach to transformation of models conforming to metamodels. Despite discussing non bijective transformations, no specification is given regarding how consistency should be restored when there are multiple choices on either direction.

A well-regarded approach to bidirectional systems is through lens combinators [Bohannon et al., 2006; Foster et al., 2007]. These define the semantic foundation and a core programming language, for bidirectional transformations on tree-structured data, but it only works well for surjective (information decreasing) transformations, our system can cope with rather heterogeneous source and target data types.

The approach followed in [Matsuda et al., 2007] uses a language for specifying transformations very similar to the one presented in this work, with automatic derivation of the backward transformation. Similar to our approach, this system statically checks whether changes in views are valid without performing the backward transformation, but they do not provide type-solving techniques such as the one available on our setting, where decisions between mapping different sets of non terminals are completely automated.

In the context of attribute grammars, Yellin's early work on bidirectional transformations in AGs defined attribute grammar inversion [Yellin, 1988]. In attribute grammars inversion, an inverse attribute grammar computes an input merely from an output, but in our bidirectional definition of attribute grammars, a backward transformation can use links to the original source to perform better transformations. Thus, our approach can produce more realistic source trees after a change to the target.

The specifications we will see in this chapter will generate attribute grammars that specify them. As the reader will see, this chapter is divided into sections where we describe the underlying formalisms and techniques for our bidirectional environment, followed by and sections where we show how these can be transformed into an AG.

Whenever we need to present AG code, we will use the syntax of the AG system `Silver`. We do so because this system uses DSL for AGs that closely resembles the formalism we have presented in Chapter 2. Furthermore, as we

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

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

Figure 8.2: Concrete syntax of arithmetic expressions.

will see, `Silver` supports a huge variety of techniques that help defining AGs (including circularity, higher order and references). More important, some of the features of `Silver` are very helpful in defining AG equations from our transformation specifications. Due to its intuitive syntax, we do not need to explain how AGs are written in `Silver` for the pieces of code we will present, but special features will be presented as needed.

8.1.1 Σ-Algebra

We will define here the algebraical background that we will use to define rewrite rules for the bidirectional system that we will present in Chapter 8. Chirica et al. [Chirica and Martin, 1979] were the first to define AG as algebras. By doing so we benefit from existing terminology

We start by defining an operator scheme: $\Sigma = \langle S, F, \sigma \rangle$ where:

- S, a set of sorts (sort names);
- F, a set of operator or function names;
- σ, maps F to $S^* \times S$.

On AGs, sorts correspond to nonterminals and terminals, operators correspond to production names, and signatures in σ correspond to productions. Constants are treated as nullary operators.

Definition 15. (Σ-algebra) A Σ-algebra is defined as:

- $\{A_s\}_{s \in S}$ - an S-indexed family of sets, called *carrier sets*;
- $\{f^A : A_{s_1} \times A_{s_2} \times ... \times A_{s_n} \to A_s \mid f \in F, \sigma(f) = S_1 \times S_2 \times ... \times S_n \to S\}$. 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, $\sigma(f)$.

If the set of variables is empty, we write $W_\Sigma()$ or W_Σ. This is the ground word algebra containing only words with no variables, known as ground words or ground terms.

An example of a word is $plus(mult(a, b), c)$, where a, b and c are variables. We need to order patterns (words with variables) from least specific to most specific. We use a standard notion of specificity in that one word if more specific than another if the set of ground terms created from all instantiations is a subset of such ground terms for another pattern.

An important definition is the one of rule specificity. One rule is more specific than the other if both match the same pattern but one goes further than the other in defining patterns for subtrees, or subtrees of subtrees, and so on.

Definition 16. (Term algebra) We define term algebras (or simply *words*) for $\Sigma = \langle S, F, \sigma \rangle$, denoted $W_\Sigma(X)$. Let $X = \{X_s\}_{s \in S}$ be a S-indexed family of sets of variables. Then, $W_\Sigma(X) = \langle \{W_s\}_{s \in S}, \{f^W | f \in F\} \rangle$ where:

- Each f^W with $\sigma(f) = s_1 \times s_2 \times ... \times s_n \to s$ builds a *word* of the form $f(w_1, w_2, ..., w_n)$ where $w_i \in W_{s_i}$;
- The carrier sets $\{W_s\}_{s \in S}$ are the sets of words constructed in this way. Specifically,
  1. if $x \in X_s$, then the variable x is a word in W_s;
  2. if $f \in F$ is a nullary function then the name f (sometimes $f()$) is a wrd in W_s. That is, $f^W() = f$;
  3. if $f \in F$, with $\sigma(f) = s_1 \times s_2 \times ... \times s_n \to s$ then the word $f(w_1, w_2, ..., w_n)$, where $w_i \in W_{s_i}$, exists in W_s.

In AG terms, the ground word algebra is essentially a nice way to write down trees, and words with variables are the same as the patterns we will use to define transformations. The thing to note is that word algebras define our patterns. The variables in word algebras are indexed by their type, which is something that we also need to do. So they are an appealing way to present our patterns.

Due to the commonality between structures in attribute grammars, algebra, and rewrite rules different terms from these domains have similar meanings. The terms *term*, *word* and *tree*; the terms *production*, *operator*

and *constructor* and the terms *type*, *sort* and *nonterminal* have the same meaning and are interchangeable.

8.2 Simple Transformations

In this section we will see how rewrite rules are used to define transformation specifications and how these can be automatically inverted to form a backward transformation.

Please recall the grammars for `LET` presented in Chapter 2 (page 36 for the concrete grammar and page 38 for the abstract grammar), which were written using the BNF notation for context free grammars. These two grammars are similar to algebras and can be easily translated into an algebraic setting. To do so, we will use Σ-Algebras, as defined in the same chapter.

In Figures 8.2 and 8.3, we can see the operator scheme Σ^E for the source language E and Σ^A for the target language A, respectively. More precisely, we can see the algebraic equivalent for the concrete and abstract grammars presented in Section 2.3.

Non-terminal and terminal symbols become sorts. The sorts E, T, F in S^E correspond to the non-terminals commonly used in this example. The sort *digits* represents an integer literal symbol, 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 written in an algebraic setting correspond to operators. For example, $add, sub, et, mul, div, tf, nest, neg, const \in F^E$ for the concrete syntax of Figure 8.2. 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:

$$\sigma^E(add) = E \text{ ‘ + ’ } T \to E.$$

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

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

Figure 8.3: Abstract syntax of arithmetic expressions.

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 $W_\Sigma(String)$ but we omit *String* below. We overload *String* to denote the sort, as in Figure 8.2, and here to denote the carrier set of strings.

8.2.1 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

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

$$get^E_A(add(l, \text{`}+\text{'}, r)) \rightarrow plus(get^E_A(l), get^T_A(r))$$
$$get^E_A(sub(l, op, r)) \rightarrow minus(get^E_A(l), get^T_A(r))$$
$$get^E_A(et(t)) \rightarrow get^T_A(t)$$
$$get^T_A(mul(l, \text{`}*\text{'}, r)) \rightarrow times(get^T_A(l), get^F_A(r))$$
$$get^T_A(div(l, \text{`}/\text{'}, r)) \rightarrow divide(get^T_A(l), get^F_A(r))$$
$$get^T_A(tf(f)) \rightarrow get^F_A(f)$$
$$get^F_A(neg(\text{`}-\text{'}, r)) \rightarrow minus(constant(\text{"}0\text{"}), get^F_A(r)))$$
$$get^F_A(nest(\text{`}(\text{'}, e, \text{`})\text{'})) \rightarrow get^E_A(e)$$
$$get^F_A(const(digits(d)) \rightarrow constant(d)$$

Figure 8.4: Forward transformation specification.

and provide the forward transformation specification from Σ^E to Σ^A, which is shown in Figure 8.4.

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 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:

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

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:

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

.

The left and right-hand sides of the rewrite rules are then words in a word algebra for the operator scheme that include 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.

8.2.1.1 Restrictions on the Forward Transformation

We place a number of restrictions on forward transformation specifications from $\Sigma^S = \langle S^S, F^S, \sigma^S \rangle$ to $\Sigma^T = \langle S^T, F^T, \sigma^T \rangle$ to ensure that a backward transformation can be generated. Some of these restrictions are removed in later sections. We also assume only type correct words are used here and throughout the rest of this thesis.

1. First, $|sm(X)| \leqslant 1, X \in S^S$. Each sort in the source maps to one or zero items in the target;

2. Second, words on the left-hand side of a rewrite rule must have the form $get_Y^X(p(v_1, ..., v_n))$ for some $p \in F^S$ in which $v_1, \ldots, v_n$ are variables or are terms that do not contain variables.

   Furthermore, all variables on the left-hand side must either (i) appear on the right-hand side, or (ii) be of a sort that contains only one value, for example the sorts of so-called constant terminal symbols;

3. When viewed as patterns, the words on the left-hand side of the rules must not be overlapping, that is there can be no substitution of their variables that results in the same word;

4. We also need to ensure that the forward transformation specifies a total function from Σ^S to Σ^T;

   Definition 17. (Total transformation) A transformation specification from source $\Sigma^S = \langle S^S, F^S, \sigma^S \rangle$ to target $\Sigma^T = \langle S^T, F^T, \sigma^T \rangle$ is said to be a total transformation if and only if for each $X \in S^S$ and $Y \in sm(X)$ there exists a rule with the left-hand side of the form $get^X_Y(p(v_1, ..., v_n))$ for each $p \in F^S$ in which $v_1, \ldots, v_n$ are variables.

   In terms of attribute grammars this is the definition of an attribute grammar that passes the closure test ([Knuth, 1968]);

5. Words on the right-hand side are also restricted. For a rule with the left-hand side $get^X_Y(p(v_1, ..., v_n))$ the right-hand side must be a word in which for any sub-word of the form $get^{X'}_{Y'}(w)$ the word w must be a variable. We will lift this restriction in a later sections on this chapter.

8.2.1.2 Generating Attribute Grammar Equations

Since our aim is to implement both the forward, and generated backward, transformations in attribute grammars we need to convert the rewrite rules to attribute grammar equations.

Generating attribute grammar equations from rules of this type specified above is quite straightforward. For example, the rule:

$$get^E_A(add(l, \text{`}+\text{'}, r)) \rightarrow plus(get^E_A(l), get^T_A(r))$$

is expressed as the following attribute grammar equation on the *add* production that has the signature $e :: E ::= l :: E \text{ `}+\text{'} r :: T$:

$$e.get^E_A = plus(\ l.get^E_A,\ r.get^T_A)$$

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

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

Figure 8.5: Backward transformation generated by the direct inversion of the forward transformation specification of Figure 8.4.

This means that the attribute $e.get_A^E$ is generating a a tree starting with the node $Plus$, and wil apply the attribute get_A^E to generate its left side and the attribute get_A^T to generate the right one.

In later examples we will see that rewrite rules can be more complex and thus the translation to attribute grammar code is less direct, but for now this definition suffices.

8.2.2 Generating the Backward Transformation

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

8.2.2.1 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 in the previous section.

The second step is to invert the rewrite rules. The result of this process for the rules in Figure 8.4 produces the rules in Figure 8.5. Given the restriction on the forward transformation, this process is relatively straightforward. A rule of the form

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

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

$$put_X^Y(w'(v_1', \ldots v_n')) \rightarrow w(put_{X_1}^{Y_1}(v_1'), \ldots, put_{X_n}^{Y_n}(v_n'))$$

in which the variables v_i' are of sort Y_i. This can be seen in the inverted rules in Figure 8.5.

8.2.2.2 Extending the Rules

Consider a transformation in an abstract syntax that creates the subtree $times(\_, plus(\_, \_)$. While the second argument of mul 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

$$put_X^Y(v) \rightarrow w(put_{X'}^Y(v))$$

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

$$put_F^A(e) \rightarrow nest(\text{`('}, put_E^A(e), \text{`)'})$$

that can be seen in Figure 8.5 shows that a term of type E can be converted to one of type F by wrapping it in parenthesis, that is, in the term $nest(\text{`('}, \_, \text{`)'})$.

We specialize the inverted rewrite rules of this form so that their left-hand sides are of the form $get_Y^X(p(v_1, \ldots, v_n))$ 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 $gp_Y^X(p(v_1, \ldots, v_n))$ for some $p \in F^S$ such that return type of p is X, for some α, $\sigma^\Sigma(p) = \alpha \rightarrow X$ and for some variables $v_1, ..., v_n$, and

   (b) there exists a rule of the form $gp_Y^X(t) \rightarrow w(gp_{Y'}^X(t))$

   then add the rule $gp_Y^X(p(v_1, ..., v_n)) \rightarrow w(gp_{Y'}^X(p(v_1, ..., v_n)))$

2. Repeat step 1 until no more rules can be added.

For example, the rule $put_F^A(e) \rightarrow nest(\text{`('}, put_E^A(e), \text{`)'})$ in Figure 8.5 shows that a term of type E can be converted to one of type F by wrapping it in parenthesis, that is, in the term $nest(\text{`('}, \_, \text{`)'})$.

From the original rules we would add the following rule:

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

and then repeat this process until we have as much extended rules as possible. The extended, inverted rewrite rules can now be checked for totality with the original definition of totality presented on page 135.

8.2.2.3 Generating Attribute Grammar Equations

From these extended set of rules, we generate AG equations as described before in Section 8.2.1.2, resulting in three equations on productions such as *plus*: one for put_E^A, put_T^A, and put_F^A.

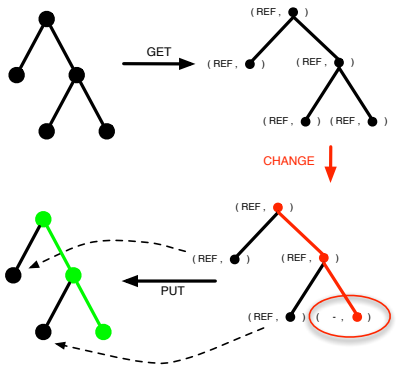


Figure 8.6: Using links back.

Notably, the generated *put* transformation does not add any unnecessary parenthesis, a common problem with simple string-based printing mechanisms.

8.3 Links Back: making use of the Original Source Term

The perceptive reader might have noticed there is a specific production, *neg*, for which generating attribute equations presents a new challenge. The issue is that the inversion of that rule and the rule for *sub* creates backward transformation rules with overlapping left-hand sides, as seen in Figure 8.5.

To address this, the *get* transformation adds a link back on generated abstract (sub) trees that refers back to the respective concrete (sub) tree that generated them.

Figure 8.6 shows this situation diagrammatically: the source tree and the target with links back is shown on top. In transforming the target tree, links back do not exist on newly constructed nodes as indicated by the lack of arrows back to the source tree. The portion in the oval (in red) was newly created, as well as the nodes on the spine to the root of the tree.

To be concrete, consider the phrase

$$sub(et(tf(const(digits(\text{"}0\text{"})))), \text{`}-\text{'}, x)$$

that became $minus(constant(\text{"0"}), x)$ in the target. If the original (sub) tree in the abstract tree was maintained during a transformation then it has a link back to the original tree of sort E and can return that when queried for its *put* attribute put_E^A. This ensures that we can recover the original source tree in the *put* transformation, when the *put* transformation is mapping it back to its original type, here E.

In practice, this means that AG code generated form the backward transformation has to maintain information regarding the original tree that created. In `Silver` we can use a specific feature called annotations, which are just attributes that get assigned when the tree is constructed and before it is given any attributes and before it is used in the construction of other tree. As the name states, they represent only notes or additional pieces of information glued to a tree. Annotations are defined as:

annotation link :: *Link*;
nonterminal Link;

abstract production linkE
this :: *Link* ::= *e* :: *E* { }

abstract production linkT
this :: *Link* ::= *e* :: *T* { }

abstract production linkF
this :: *Link* ::= *e* :: *F* { }

abstract production none
this :: *Link* ::= { }

where we define an annotation called *link* that can have four different types: a *link* can be either a *linkE*, a *linkT*, a *linkF* or *none*. We then define the nonterminal A for the abstract grammar as:

nonterminal A with link;

`Silver` will automatically produce trees of type A with an extra parameter of type *Link*. Therefore, we can generate attributes such as:

$$e.get_A^E = minus(\ l.get_A^E,\ r.get_A^T,\ link = linkE(this))$$

where the forward transformation automatically injects information regarding the original subtree that was target of a transformation.

The backward rules can use this information on the abstract side to control the transformation. This can be seen in the generated attribute equation for *minus* for put_E^A below:

$$
\begin{aligned}
&minus \ : \ s :: A ::= l :: A \ r :: A \\
&s.put_E^A = \textbf{case } s.link \textbf{ of} \\
&\qquad\qquad link_E(e) \rightarrow e \\
&\qquad\qquad link_T(t) \rightarrow et(t) \\
&\qquad\qquad link_F(f) \rightarrow et(tf(f)) \\
&\qquad\qquad none \qquad \rightarrow sub(\ l.put_E^A,\ r.put_E^A\ )
\end{aligned}
$$

where the link is always used unless no link exists, in which case we have to default to the transformation technique described in the previous section.

One important remark is that in this process, maintenance and usage of the links back (and the *canBe* relation) is completely automatic and requires no extra coding.

8.3.1 Allowing Overlapping Rewrite Rules

In order to generate equations we need to sort the rewrite rules based on the rule specificity of the left-hand side and use this left-hand side in a **case**-expression to select the most specific one.

On the transformation that have *minus* as domain (Figure 8.5), we sort the rewrite rules so that the most precise one is used before the more general ones: the transformation tries to apply $put_F^A(minus(constant(\text{"0"}), f))$ first, because it is more specific.

The techniques described so far create a setting where we can describe complex transformations between grammars with various non-terminals and produce, from these rules, attribute specifications for the transformation in both directions.

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

- $S^C = \{Root_C, Decls_C, Decl_C, Vars_C, Type_c,$
  $\text{`,'}, \text{`;'}, String\}$,
- $F^C = \{root_C, consDecl_C, nilDecl_C, multiDecl_C,$
  $consVar_C, oneVar_C, intType_C, floatType_C,$
  $arrayType_C, \text{`,'}, \text{`;'}, String\}$
- $\sigma^C(root_C) = Decls_C \rightarrow Root_C,$
  $\sigma^C(consDecl_C) = Decl_C\ Decls_C \rightarrow Decls_C,$
  $\sigma^C(nilDecl_C) = \epsilon \rightarrow Decls_C,$
  $\sigma^C(multiDecl_C) = Type_C\ Vars_C\ \text{`;'} \rightarrow Decl_C,$
  $\sigma^C(consVar_C) = String\ \text{`,'}\ Vars_C \rightarrow Vars_C,$
  $\sigma^C(oneVar_C) = String \rightarrow Vars_C,$
  $\sigma^C(intType_C) = \text{`}int\text{'} \rightarrow Type_C,$
  $\sigma^C(floatType_C) = \text{`}float\text{'} \rightarrow Type_C,$
  $\sigma^C(arrayType_C) = \text{`}array\text{'}\ Type_C \rightarrow Type_C.$

Figure 8.7: Concrete syntax for declarations.

8.4 Supporting Non-linear, Compound Rules and Partial Transformations

In this section we extend the process described above to apply to non-linear rules and to compound rewrite rules (or simply compound rules. We also describe a means for handling two situations in which the generated backward transformation is partial. The first is when some hand-written manipulation of the target tree can move it back into the domain of the backward transformation, the second is fall-back case in which some portion of the backward transformation must be written manually.

8.4.1 Non-linear, Compound Rule Specification

We start by presenting in Figures 8.7 and 8.8 a new pair of concrete and abstract algebras/grammars. The concrete syntax allows sequences of declarations of the form *int* $x,\ y,\ z$; while the abstract requires simpler declarations of just one variable, and thus this example becomes *int* x; *int* y; *int* z; in the abstract.

To support transformations such as this, we allow the right-hand side of rewrite rules to have attribute operators (get_Y^X) that contain a term (a tree) instead of just a variable. Such rules are called compound rules. In particular, for the concrete grammars defined in Figure 8.7, consider the two rules for the production $multiDecl_C$. These are the first two rewrite rules in Figure 8.9.

The first rule is similar to the ones we have seen in previous sections. In the more interesting second rule, the right-hand side creates a new term/tree in the concrete syntax ($multiDecl_C(t, rest, ';')$) on which we recursively apply the forward transformation $get_{Decl_A}^{Decl_C}$.

To generate the forward transformation AG equations for the first rule we use the strategy presented in Section 8.2.1.2. For the second rule, the same process now defines a new concrete tree and then accesses the $get_{Decl_A}^{Decl_C}$ attribute on that "locally" created tree.

This process, described in general below, creates the following equation for the $get_{Decl_A}^{Decl_C}$ on the $multiDecl_c$ production:

$$
\begin{aligned}
&multiDecl_c\ :\ d :: Decl_c ::= t :: Type_C\ vars :: Vars_C\ ';' \\
&d.get_{Decl_A}^{Decl_C} = \textbf{case}\ d\ \textbf{of} \\
&\qquad multiDecl_C(t, oneVar_C(v, \_), \_) \rightarrow varDecl(t.get_{Type_A}^{Type_c}, v) \\
&\qquad multiDecl_C(t,\ consVar_C(v, \_, rest), \_) \rightarrow \\
&\qquad\qquad seqDecl(\ varDecl(t.get_{Type_A}^{Type_C},\ v), \\
&\qquad\qquad\quad (multiDecl_C(t, rest, ';')).get_{Decl_A}^{Decl_C})
\end{aligned}
$$

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

- $S^A = \{Root_A, Decl_A, Type_A, String\}$
- $F^A = \{root_A, seqDecl_A, skipDecl_A, intType_A,$
  $floatType_A, arrayType_A, String\}$
- $\sigma^A(root_A) = Decl_A \rightarrow Root_A,$
  $\sigma^A(seqDecl_A) = Decl_A\ Decl_A \rightarrow Decl_A,$
  $\sigma^A(skipDecl_A) = \epsilon \rightarrow Decl_A,$
  $\sigma^A(varDecl_A) = Type_A\ String \rightarrow Decl_A,$
  $\sigma^A(intType_A) = \epsilon \rightarrow Type_A,$
  $\sigma^A(floatType_A) = \epsilon \rightarrow Type_A,$
  $\sigma^A(arrayType_A) = Type_A \rightarrow Type_A.$

Figure 8.8: Abstract syntax for declarations.

8.4.2 Inverting the Rewrite Rules

Inverting the rules for $multiDecl_C$ creates non-linear rewrite rules (or simply non-linear rules) with side conditions. The inverted rules for $multiDecl_C$ are shown below:

- $put^{Decl_A}_{Decl_C}(varDecl_A(t, v)) \rightarrow$
  $multiDecl_C(put^{Type_A}_{Type_C}(t),\ oneVar_C(v), \text{`;'})$
- $put^{Decl_A}_{Decl_C}(seqDecl_A(varDecl(t, v), right)\ )$
  $multiDecl_C(put^{Type_A}_{Type_C}(t),\ consVar_C(v, \text{`,'}, rest))$
  where $put^{Decl_A}_{Decl_C}(right) = multiDecl_C(put^{Type_A}_{Type_C}(t), rest, \text{`;'})$

Previously, on forward transformation rules,

1. Variables on the left were wrapped with a *put* operator on the right of the *put* rule, and

2. Variables on the right wrapped by a *get* operator became (unwrapped) variables on the left of the *put* rule. This can be seen in the first inverted rule above.

For the second rule, we generalize this process so that in the forward transformation rules, a term trm on the right wrapped by a *get* operator becomes new variable ($right$ in the case above) on the left of the *put* rule and creates a side condition of the form $put(right) = trm'$, where trm' is the result of applying this process to trm. Note that this generalized process wraps variables of sorts in the target with a *put* operator, but not those with a sort in the source. In the above example t is wrapped with $put_{Type_C}^{Type_A}$ but $rest$ is not.

8.4.3 Generating Attribute Grammar Equations

Generating the attribute equations for compound, non-linear rules with side conditions used either in the forward or generated backward transformation requires extending the process described in the Section 8.2.1.2, where the case-expressions are used as components in the attribute equations when no links back to the source are applied, and thus fit into the equations as previously specified. The pattern p in the left-hand side of the rule becomes the pattern on the left of the case clause.

8.5 Tree Repairs

In the previous example, the sequences of declarations in the source and target take different forms. In the source, it is a list formed by traditional *cons* and *nil* operators. But in the target, a more general tree structure is allowed using *seq* and *skip* (empty) operators.

Using the techniques described above, the range of the *get* transformation is not the full range of valid sequences of declarations in the target and thus the generated backward transformation is a total transformation, it is only a partial transformation. Only sequences that have a list-like shape in the abstract can be mapped back to the concrete.

- The sort map: $sm^{get} :: F^C \to 2^{F^A}$:
  $sm^{get}(Root_C) = \{Root_A\}$,
  $sm^{get}(Decl_A) = \{Decls_C,\ Decl_C\}$,
  $sm^{get}(Type_A) = \{Type_C\}$

- The rewrite rules rw^{get}:

$$get^{Decl_C}_{Decl_A}(multiDecl_C(t, oneVar_C(v), \text{`;'})) \to$$
$$\quad varDecl(get^{Type_C}_{Type_A}(t), v)$$
$$get^{Decl_C}_{Decl_A}(multiDecl_C(t, consVar_c(v, \text{`,'}, rest), \text{`;'})) \to$$
$$\quad seqDecl(varDecl(get^{Type_C}_{Type_A}(t), v), multiDecl_C(t, rest, \text{`;'}))$$
$$get^{Root_C}_{Root_A}(root_C(c)) \to root_A(get^{Decls_C}_{Decl_A}(c))$$
$$get^{Decl_C}_{Decls_A}(consDecl_C(d, rest)) \to seqDecl_A(get^{Decl_C}_{Decl_A}(d), get^{Decls_C}_{Decl_A}(rest))$$
$$get^{Decls_C}_{Decl_A}(nilDecl_C()) \to skipDecl_A()$$
$$get^{Type_C}_{Type_A}(intType_C()) \to intType_A()$$
$$get^{Type_C}_{Type_A}(floatType_C()) \to floatType_A()$$
$$get^{Type_C}_{Type_A}(arrayType_C()) \to arrayType_A()$$

Figure 8.9: Forward transformation specification for declarations.

To see this, note that the inversion of the rule:

$$get^{Decl_A}_{Decls_C}(consDecl_C(d,\ rest)) \rightarrow seqDecl_A(get^{Decl_C}_{Decl_A}(d),\ get^{Decls_C}_{Decl_A}(rest))$$

yields the following rule, with variables changed to be more appropriate for the abstract syntax, for the backward transformation:

$$seqDecl_A(d_1,\ d_2) \rightarrow consDecl_C(get^{Decl_A}_{Decl_C}(d_1),\ get^{Decls_A}_{Decl_C}(d_2))$$

Note that from an abstract production $seqDecl_A$, we need to get a $Decl_C$ from the left child d_1 and a $Decls_c$ from the right child d_2. If d_1 is of the form $varDecl(t, n)$ this is no problem since there will be a transformation rule from these back to a $multiDecl_C$, via one of the rules in Figure 8.9. But if d_1, after some transformation on the abstract tree has taken place, is a $seqDecl_A$ or a $skipDecl_A$ then this tree is not in the domain of the backward transformation.

However, in many cases like this, it is possible to *repair* such trees and convert them back to a list-like structure so that they are in the domain of the *put* transformation. This what we call a tree repair.

We extend these techniques to generated AG equations to detect if the target tree is in the domain of the *put* transformation: in this case, a $needsRepair^{Decl_A}_{Decl_C}$ attribute that is true on nodes of sort $Decl_A$ that need to be repaired before transforming back to a $Decl_C$ sort in the source.

Generating equations for a $needsRepair^{Decl_A}_{Decl_C}$ attribute is quite straightforward since is has the same structure as the equations for the corresponding attribute $put^{Decl_A}_{Decl_C}$. Constructed trees are replaced by $false$ and error-cases are replaced by $true$.

In such cases, the user must write attribute equations, $repair^{Decl_A}_{Decl_C}$ in this case, that convert the tree into a form that is in the domain of the *put* transformation. In this case an accumulating inherited attribute can be used to provide a $Decl_A$ with the tail of the list that should follow it. The point being that such repairs can be made, but equations must be written by hand.

In generating the *put* attribute equations, the framework will then re-

place attribute accesses of the form $n.put^{Decl_A}_{Decl_C}$ with expressions that query the $needsRepair^{Decl_A}_{Decl_C}$ attribute: if it is true the tree is first repaired before performing the *put* transformation, otherwise the $put^{Decl_A}_{Decl_C}$ attribute is safely accessed.

This allows our approach to *gracefully degrade* in situations where the generated backward transformation is partial, but the trees in the target can be manipulated (repaired) to be in the domain of the generated backward transformation. For real-world languages we expect some (hopefully small) portions of the language may need repair, but then the generated backward transformation can be used automatically on the rest of the language.

8.6 Embedding Bidirectional Attribute Grammars

In the previous chapter we describe a system for generating attribute grammar implementations of bidirectional transformations given only a specification of the forward transformation. This approach is applied here to the embedding of AGs using zippers. Here we sketch the structure of the generated bidirectional transformation in our zipper-based setting, while the full details can be found in the earlier sections.

Returning to our running example of the `LET` language presented and developed throughout the previous chapters, we have worked with its abstract for as it is easier to handle and to reason about.

However, the CST of `LET` 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 representation. This representation was presented before in Chapter 2 (page 36) as a CFG, and now we present it in the form of an `Haskell` data type:

$$\begin{aligned}
&\textbf{data}\ Root_C = Root_C\ Let_C \\
&\textbf{data}\ Let_C = Let_C\ Dcls_C\ In_C \\
&\textbf{data}\ Dcls_C = ConsLet_C \quad String\ Let_C\ Dcls_C \\
&\qquad\qquad \mid ConsAssign_C\ String\ E \quad Dcls_C
\end{aligned}$$

```
                | EmptyDclsC
data E = Add E T
       | Sub E T
       | Et  T
data T = Mul T F
       | Div T F
       | Tf  F
data F = Nest  E
       | Neg   F
       | Var   String
       | Const Int
```

Nonterminals `RootC`, `LetC` and `DclsC` have a single corresponding nonterminal in the abstract representation, `Root`, `Let` and `Dcls` respectively (please recall the abstract grammar for `LET` presented in Chapter 2, page 38). 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 to 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[1]:

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

This transformation faces the exact same problems we have seen in Section 8.2.2.2. The constructors `Et` and `Tf` do not have corresponding constructors in the abstract syntax. However, deriving the backward transformation from these mappings presents new challenges. Some decision must be made to determine if an *Expr* on the abstract side is mapped to an E, T or F and this decision should be made for each node in the AST. The simple, naive solution is to map every *Expr* back to F and wrap everything in parenthesis, but this is far from ideal as it unnecessarily produces a complicated concrete representation.

Also as we have seen before, the production *Neg* also presents additional challenges. This production is transformed according to the mapping:

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

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

Although on the previous sections we have seen AG code written as a DSL in `Silver`, our embedding provides sufficient expressiveness to support such transformations.

[1]Production *Neg* omitted on purpose.

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

$$\begin{aligned}\textbf{data}\ \mathit{Link} = {} & \mathit{IsRoot}_C\ \mathit{Root}_C \mid \mathit{IsLet}_C\ \mathit{Let}_C \quad \mid \mathit{IsIn}_C\ \mathit{In}_C \\ & \mid \mathit{IsDcls}_C\ \mathit{Dcls}_C \mid \mathit{Is}_E\ E \mid \mathit{Is}_T\ T \mid \mathit{Is}_F\ F \mid \mathit{Empty}\end{aligned}$$

which represents a link to the original node in the CST for which the AST node was created. This is the equivalent of using annotations in `Silver`.

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

In our setting, the transformations are represented by a set of synthesized attributes *get* that is named *getFrom_To*, with *From* representing the type that is being mapped to *To*. This is the equivalent of writing get_{To}^{From}, using the syntax from the previous sections.

Next, we present an example of an attribute that implements the mapping from Root_C to *Root*:

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

where *createLink* is defined as the function that takes a zipper and creates an instance of *Link*. The semantics are simple and very similar to what we have seen implemented in `Silver`: 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.

The function *createLink* simply extracts information from a zipper and creates a *Link*:

$$\begin{aligned}&\mathit{createLink} :: \mathit{Zipper}\ a \rightarrow \mathit{Link} \\ &\mathit{createLink}\ \mathit{ag} = \textbf{case}\ (\mathit{getHole}\ \mathit{ag} :: \mathit{Maybe}\ \mathit{Root}_C)\ \textbf{of}\end{aligned}$$

```
  Just (e) → IsRoot_C e
  _ →        case ...
```

This function has to be defined for all the different type constructors (that would appear instead of the three points), but this code is generated so no additional effort is required by the user of the system.

As the reader might have noticed, defining AG equations in our zipper-based embedding closely follows the implementations we have seen in `Silver`, as we are using the exact same bidirectional background. For example, if the user recalls the put attribute (defining the backward transformation) for *Add*, it will ask for $putExpr_E$ of its left child and $putExpr_T$ of its right since these are the correct types for its left and right children, and in our system each *Expr* knows how to translate itself back to any of the E, T, or F non-terminals. To do so, we will have exactly the same number of attributes, in the same order.

Let us continue with a more interesting example. Next we present the attribute that transforms parts of an abstract tree whose node is of type *Expr* into nodes of the concrete tree whose type is F:

```
putExpr_F :: Zipper Root → F
putExpr_F ag = case (getLink ag) of
  Is_E e  → Nest e
  Is_T t  → Nest (Et t)
  Is_F f  → f
  Empty → case (constructor ag) of
    "Plus"  → let left  = putExpr_E (ag .$ 1)
                  right = putExpr_T (ag .$ 2)
              in  Nest (Add left right)
    "Minus" →
      case (getHole ag :: Maybe Expr) of
        Just (Minus (Constant 0 _) _ _) → Neg (putExpr_F (ag .$ 2))
        otherwise →      let left  = putExpr_E (ag .$ 1)
                             right = putExpr_T (ag .$ 2)
                         in  Nest (Sub left right)
    "Times" → let left  = putExpr_T (ag .$ 1)
```

$$
\begin{array}{rl}
 & \quad right = putExpr_F\ (ag\ .\$\ 2) \\
 & \textbf{in}\ \ Nest\ (Et\ (Mul\ left\ right)) \\
\texttt{"Divide"} \rightarrow & \textbf{let}\ left\ \ = putExpr_T\ (ag\ .\$\ 1) \\
 & \quad right = putExpr_F\ (ag\ .\$\ 2) \\
 & \textbf{in}\ \ Nest\ (Et\ (Div\ left\ right)) \\
\texttt{"Constant"} \rightarrow & Const\ (lexemeConstant_1\ ag) \\
\texttt{"Variable"} \rightarrow & Var\ \ \ (lexemeVariable_1\ ag)
\end{array}
$$

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

- The first thing the attribute computation does is extracting the link from the target 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.

  We would use annotations in `Silver`, a special feature where additional information is automatically added and managed by this system. Since we are embedding AGs, we have to use standard `Haskell` features, which is why we use traditional data types. Nevertheless, an attribute in `Silver` is syntactically very similar to one in `Haskell`, as one can verify.

- Whenever the types do not match, the system automatically detects if any special constructs can be used. Take for example the line $Is_E\ e \rightarrow Nest\ e$. 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. Again, this is exactly what happens in `Silver`.

- 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*.

As expected, in the full implementation of the backwards transformation there are also the functions $putExpr_E$ and $putExpr_T$, with definitions very similar to $putExpr_F$.

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

8.7 Conclusions

In this chapter we have shown how rewrite rules can be used to specify forward transformation, be automatically inverted to specify backward transformations, and then be implemented in attribute grammars where the quality of the transformation is enforced.

It is important to note that the features our bidirectional system supports are completely automatic for many applications, freeing the programmer of having to write complex attribute equations that have to perform multiple pattern matching, manage both the links back and their types, prioritizing transformations, etc.

The only part of our system which is not automatic are the tree repairs, but even in these cases we generate attributes to check for the need of repairs, simplifying the programmers work as much as possible.

Chapter 9

Tools

Summary

In this chapter we will present a practical application of our embedded AG system. We defined a combinator language which creates an abstract tree that represents scheduling of processes. This tree is given semantics with zipper-based AG technique, and integrated into a web portal that certifies software.

9.1 Introduction

If the reader recalls the introduction of this thesis, on Chapter 1, we have stated that our setting, with the embedding of AGs, provides a good environment for implementing DSLs in `Haskell`. AGs provide a good methodology for defining semantics and `Haskell`'s data types and parser libraries provide easy ways of parsing languages and instantiating ASTs.

In this chapter we will show how we used this setting to implement a combinator language for certification process management and how that language was used on the construction of a web portal for software analysis, always with the support of our zipper-based environment.

The DSL we present here is produced by a set of `Haskell` combinators that define an abstract syntax tree. The advantages of using this combinator system is that type correctness is enforced, which means syntactically we

know we are dealing with a correct tree. Traditionally, this would be the responsibility of a parser.

When we have an AST as an `Haskell` data type, we can use our technique of wrapping this data type inside a zipper and define attributes as functions over this zipper structure.

The resulted combinator language, with AG-provided semantics, supports a portal for certifying software. The user defines a chain of certification processes, which correspond to different analysis and transformational tools, and that chain is transformed into an abstract representation that is analyzed and transformed in `Haskell` and that ends up with the generation of low-level scripts that implement it.

9.2 Embedding DSLs for Language Analysis

In this section we describe a domain specific language that allows programmers to describe in an abstract level how software artifacts can be combined into powerful software certification processes. The DSL is the building block of a web-based, open-source software certification portal that will be presented in the next section. This work we briefly describe here is present in detail in [Martins et al., 2012].

Here, we will introduce the language as an embedding of a combinator library written in the `Haskell` programming language. The semantics of this language are expressed via zipper-based attribute grammars that are embedded in `Haskell`, which provide a modular and incremental setting to define the combination of software artifacts.

We understand a certification as the process of analyzing a software solution while producing an information report about it. Certifications are expected to process an Open Source Software (OSS) solution and provide a technical analysis of it, decreasing the exposure to risk associated to the adoption of OSS.

Also, users should be capable of combining already existing certifications into more complex analyses. The language that we introduce in this section aims at allowing an easy configuration of the flow of information among

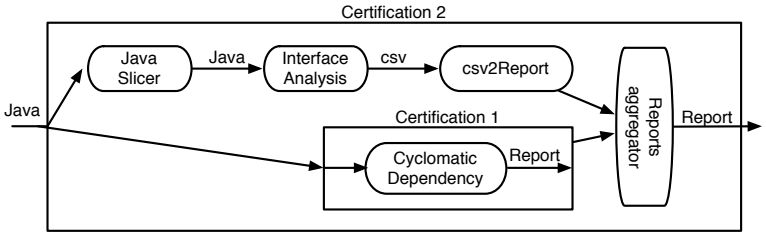


Figure 9.1: The flow of information implemented in Certification 2.

processes/tools that run either in parallel or in sequence to create certifications, and at the automatic analysis and generation of low-level scripts that implement such configuration.

There are similar works where languages were defined to express sequential computations. In [Campos and Barbosa, 2009], an implementation of the orchestration language `Orc` [Kitchin et al., 2009] is introduced as an embedded domain specific language in `Haskell`. In this work, `Orc` was realized as a combinator library using the lightweight threads and the communication and synchronization primitives provided by the `Haskell` library [Jones et al., 1996]. Despite the similarities on the use of combinators written in `Haskell`, this work differentiates from ours because we do not rely on any existing orchestration language. Rather, we generate low level `Perl` scripts from combinators whose inputs are direct references to system processes. Also, our processes management does not rely on Concurrent `Haskell`, but on the parallelization features of the target system via system calls on the script.

The use of attribute grammars as the natural setting to express the embedding of DSLs in `Haskell` is proposed in [Saraiva, 1999; Saraiva and Swierstra, 1999b,c; Swierstra et al., 1999]. These embeddings use powerful circular, lazy functional programs to execute DSLs. Such circular, lazy evaluators are a simple target implementation of AGs used by several systems [Kuiper and Saraiva, 1998; Swierstra et al., 2004].

In Figure 9.1 we sketch the flow of information that has been specified in order to produce a sample certification called `Certification 2`. This is a certification that expects `Java` programs and that analyzes them according

to two distinct sub-processes that are independent with respect to each other and therefore can be executed in parallel.

One-off the process chain of `Certitication 2` is composed by a series of software units, namely `Java Slicer`, `Interface Analysis` and `csv2Report`. The other, which is itself a certification called `Certification 1`, implements a cyclomatic dependency analysis while producing a report.

Finally, and since all certifications must produce reports that conform to our format, the two distinct flows of information are aggregated by a reports aggregator, a component whose single responsibility is precisely to aggregate outputs.

9.2.1 Defining Combinators

The combinator language that we propose is written in `Haskell`. We start by defining the data types for certifications and components, which we present next:

```
data Certification = Certification Name Processing_Tree
data Component = Component Name InputList OutputList BashCall

data Language = Java          -- .java
              | C_Source      -- .c
              | C_Header      -- .h
              | Cpp           -- .cpp
              | Haskell       -- .hs
              | XML           -- .xml
              | Report        -- Report XML
type Arg        = String
type Name       = String
type BashCall   = String
type InputList  = [(Arg, Language)]
type OutputList = [(Arg, Language)]
```

A certification has a name (e.g. `Certification 1` as in Figure 9.1) and defines the particular information flow to achieve a desired global analysis.

A component is represented by a name, the list of arguments it receives and the list of results it produces. These lists, that are represented by type synonyms *InputList* and *OutputList*, respectively, have similar definitions and consist of varying numbers of arguments and results. The arguments that are defined for a particular component are then passed to concrete bash calls. This is the purpose of the type *BashCall*, which consists of the name of the process to execute on the system.

In order to represent the flow of information defined for a certification, we have defined the data type $Processing_{Tree}$, which is introduced next:

$$
\begin{aligned}
\textbf{data}\ Processing_{Tree} = {} & Root_{Tree} \quad Processing_{Tree} \\
\mid {} & Sequence_{Node}\ Processing_{Tree}\ Processing_{Tree} \\
\mid {} & ParallelNode\ \ Processing_{List}\ \ Processing_{Tree} \\
\mid {} & Process_{Cert}\ \ Certification \\
\mid {} & Process_{Comp}\ \ Component \quad Arg\ Arg \\
\mid {} & Input \\
\textbf{data}\ Processing_{List} = {} & Processing_{List} \quad Processing_{Tree}\ Processing_{List} \\
\mid {} & Processing_{ListNode}\ Processing_{Tree}
\end{aligned}
$$

The simplest processing tree that we can construct is the one to define a certification with a single component. This is expressed by the constructor $Process_{Comp}$, which also expects a name to be associated to the component and the specification of the options to run the component with.

A certification can also be defined by a single sub-certification, here represented by $Process_{Cert}$. In addition to these options, more complex certifications can be constructed by running processes in sequence, using $Sequence_{Node}$, and in parallel, using *ParallelNode*. Constructor *ParallelNode* takes as arguments a processing list and a processing tree.

The first argument represents a list of trees whose processes can run in parallel. The second argument is used to fulfill our requirement that all results of all parallel computations must be aggregated using a component. Therefore, this processing tree must always be a component (and this is ensured by our type checking mechanism that is capable of aggregating in-

formation into one uniform, combined output.

For sequencing operations, we define the combinator `>-`. This combinator defines processes that are to be executed in a chain, i.e, where the output of a process serves as input to the process that follows it. When sequencing processes, it is also the case that if one of process in the chain fails the entire chain will also fail.

The use of the combinator `>-` must always be preceded by the use of constructor *Input*, which signals the beginning of an information flow. Then, as many components and certifications are added as needed, as long as they are, again, connected by `>-`. Next, we show an example of a chain of events defined using this combinator:

Input`>-`(*jSlicer*, `"-j"`, `"-i"`)`>-`(*iAnalysis*, `"-i"`, `"-csv"`)`>-`*certif*

There is also a combinator that enables the parallel composition of processes, This type of composition is actually supported by two combinators, `>|` and `>|>`. The first one is responsible for launching a varying number of processes in parallel, while the second is mandatory after a sequence of `>|`, and uses and chains all outputs of all processes to a component that is capable of aggregating them. Next, we present an example of how these combinators work together:

Input`>-`*cert3* `>|`
Input`>-`*cert1*`>-`*cert5* `>|`
Input`>-`(*jSlicer*, `"-j"`, `"-x"`)`>-`*cert8*`>|>`(*aggr*, `"-x"`, `"-r"`)

Combinator `>|` takes either a processing tree, a component, a certification or a set of processes constructed using the other combinators. The arguments of `>|` must always begin by constructor *Input*, to give a clear idea of the flow of information. In the case of this listing it is indeed easy to spot where the information enters a parallel distribution.

As for combinator `>|>`, it is mandatory for it to appear in the end of a parallelized set of processes. It is used to aggregate all the outputs of all the child processes into a single standard output, and it is able of combining varying numbers of parallel processes using an aggregation component.

With these combinators we are already capable of defining a certification. Next, we present an example of a certification:

$$
\begin{aligned}
&\mathit{Input}\texttt{>-}(\mathit{comp1}, \texttt{"-s"}, \texttt{"-ast"})\texttt{>-}\mathit{parallel} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{>-}(\mathit{comp2}, \texttt{"-h"}, \texttt{"-r"})\texttt{>-}\mathit{cert} \\
&\mathit{parallel} = \mathit{Input}\texttt{>-}(\mathit{comp3}, \texttt{"-ast"}, \texttt{"-x"})\texttt{>|} \\
&\qquad\qquad \mathit{Input}\texttt{>-}(\mathit{comp4}, \texttt{"-ast"}, \texttt{"-x"})\texttt{>|} \\
&\qquad\qquad \mathit{Input}\texttt{>-}(\mathit{comp5}, \texttt{"-ast"}, \texttt{"-x"})\texttt{>|>}(\mathit{compAggr}, \texttt{"-x"}, \texttt{"-h"})
\end{aligned}
$$

In this example we have introduced a parallel computation in the middle of a sequence of processes. This is visible through the use of the combinator `>-`, which chains computations. An example of where such a scenario could be useful is in the case of having a set of processes to analyze an AST, but having an input as source code. This code then needs to be converted to an AST using, in our illustration, *comp1*.

Following a manual approach to implementing a script for this scenario would lead to a complex development process. Indeed, one would have to manually edit it to make sure the process corresponding to *comp1* feeds each process on the parallel computation (that in this case is composed by 3 subprocesses but that could easily grow further). The parallel computations are described with two combinators: `>|`, which describes sequences to be run in parallel, and `>|>`, which channels all the information from the parallel computations into an aggregator.

Furthermore, imagine that we do not want the results of the parallel computation, but rather we want them to be compared against a repository of results to analyze the characteristics of our AST. For this, a certification *cert* has been implemented, but it does not take as input the same format that is returned by our parallel processes.

A possible solution using our combinator language is to channel the result of the parallel processes to an auxiliary component *comp2* that converts the formats so the information can be fed to the certification. But this is something that is not easily implemented by hand.

The overall proposal of performing everything manually would be considerably difficult and error-prone. One would have to mess with legacy scripts,

potentially built by different programmers, to understand them, and to create the correct chain of information. And further difficulties still need to be resolved if one wants to ensure script robustness, and that the processes are controlled in terms of processing times and failures, for example. This is completely automatic in our setting.

The significant effort of manually building scripts is furthermore severely compromised considering script evolution. Indeed, small changes in particular certification sub-processes may lead to severe overall changes being required.

We believe that our combinator approach has the advantage of not only making it easier to create flows of information among process, that can be easily edited, but also of being highly modular. Indeed, our combinators receive as arguments small fragments that can be edited, managed and transformed in simple ways. Also, it does not require a significant effort for a programmer to change a particular certification, making it able of producing different results.

9.2.2 Type Checking

Since we have chosen to use `Haskell` in our implementation, we inherit the advanced features of both the language and its compilers. In particular, the powerful type system of GHC helps us providing some static guarantees on the certifications that are developed. Indeed, the order in which the combinators of our language are applied within a certification is not arbitrary, and the uses that do not respect it will automatically be flagged by the compiler.

The simplest example of this situation is the attempt to construct a processing tree without explicitly using constructor *Input*, but of course more realistic examples are also detected, e.g., not wrapping up a set of parallel computations with the use of the combinator `>|` as well as the application of an aggregator.

We also analyze if the types match in the flow of information defined for a certification. This means that the input type of a process must match the

output type of the process feeding it. Taking the example on Figure 9.1, the output type of `Interface Analysis` must be the same as the input type of `csv2Report`, which in this case is `csv`.

As we have already mentioned, our type checking is performed on trees of the type of $Processing_{Tree}$ and, because we used an AG-based approach, this analysis is broken down into tree nodes which, in our case, are represented by the `Haskell` constructors of the $Processing_{Tree}$ data type.

9.2.3 Script Generation

We have shown a how set of processes can elegantly be combined into a certification, either in a sequence, in parallel or in any combinations of these two. We have also shown how these combinators are easy to read, understand and modify, and how we implemented a supporting type checking mechanism that guarantees a correct match of types throughout the processing chain.

In this section we describe how we can generate `Perl` scripts that implement the certifications that are created using our combinators. These scripts can be seen as the low level representation of our certifications: they describe the processing chain, handle the individual processes for their completeness and manage the flow of information throughout all the processes the certification is made of.

The script generation follows the same AG-based strategy that we have applied to type check our certifications. The basic idea behind it is that each tree node, that represents a part of the processing tree, generates the corresponding sub-piece of the global script, and that the overall meaning of the AG is the entire, fully working, script.

Next, we present an example of a script that was generated by the following combination: *Input*`>-`(*comp1*, `"-j"`, `"-x"`)`>-`(*comp2*, `"-k"`, `"-o"`).

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::CaptureOutput qw/capture_exec/;
use IO::File;
```

```
$| = 1;
my $stdout0 = $ARGV[0]; # This is actually stdin

my $cmd1 = "./comp1 -j -x";
my ($stdout1, $stderr1, $success1, $exit_code1)
  = capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "** The process $cmd1 does not exist! **"; }
if (not $success1)
  { die "** The process $cmd1 failed with msg: $stderr1 ! **";}

my $cmd2 = "./comp2 -k -o";
my ($stdout2, $stderr2, $success2, $exit_code2)
  = capture_exec( $cmd2 . "<<END\n" . $stdout1 . "\nEND" );
if ($?) { die "** The process $cmd2 does not exist! **"; }
if (not $success2)
  { die "** The process $cmd2 failed with msg: $stderr2 ! **";}

print $stdout2;
```

In this script both components are executed via the system call command `capture_exec`, their existence is verified and their `STDERR` output is checked for problems. Afterwards, their results are channeled to the process that comes next, which in the case of the first component is the second component, and in the case of the second component is the `STDOUT` of the script since the computations ended.

The scripts that we generate perform process control and scheduling of computations both on chained and in parallel flows of information while still being readable and understandable. Nevertheless, constructing such scripts manually is still an error-prone task even for small certifications with small number of processes. A larger certification (with, e.g, over a dozen sub-processes) would imply a significant amount of time to be implemented and debugged, just to name some phases of the development process.

In fact, this situation would further deteriorate if we were considering integrating in our framework more advanced scripting features. We could, for example, be interested in time-outing the processes independently to ensure

they do not go past a certain time frame, in controlling better the input and output information from processes (checking, for example, if input information is able to be processed though `STDIN`, i.e, respects the specific implementations on different programming languages and environments[1], etc) or in ensuring that, anytime an error occurs, the script actually creates a small report that is integrated in the final certification instead of just showing information through the standard `STDERR`.

We believe, however, that following an AG-based approach similar to our own would facilitate the integration of these features in structured and simple ways as one-of tasks that once achieved become automatically available for any certification, old and new.

Furthermore, because tree nodes are modular units of an AG implementation, it is even easier to upgrade small parts of the script as needed. For example, implementing timeout features on parallel processes would imply changing only the corresponding attribute on the desired tree nodes.

Generating scripts automatically presents several difficulties that are orthogonal to any generation mechanism, including to our own AG-based setting. Code translation is challenged by the usual concerns of assuring that the result is both syntactically and semantically perfect, and that all constructors/primitives/declarations of the target language are correctly declared and used. Implementing multiple processes, for example, implies a tight control on the variables that carry their results and their inputs.

In a chain of processes from `A` to `B`, the variable that stores the information produced by `A` must be the one that feeds `B`, and all the variables must have different names (and if they do not, they must be used in different execution contexts). Also, this mechanism is even harder to implement within parallel processing, where all the outputs are aggregated into one single process (remember the data type $Processing_{Tree}$ on page 159, where a parallel tree node has always a processing list and a component that aggregates information).

After defining the scope rules for variables, the script generation via the attribution is easy to perform, once again thanks to our AG-based mechanism. The code generation follows the syntactic rules of the target language

[1] `http://en.wikipedia.org/wiki/Here_document` [Accessed in 25 March, 2012]

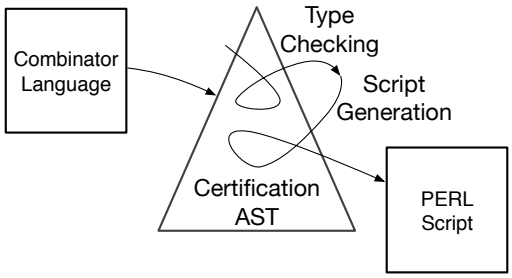


Figure 9.2: Processing the combinator DSL.

(in this case `Perl`) and ensures that the constructors/primitives/parenthesis are written and in the correct form.

9.2.4 Overview

Overall, the machinery for the embedding a combinator DSL we have presented is resumed in Figure 9.2. We have the typical behavior when processing languages using an AG: we create an AST, that in this case comes from the `Haskell` combinators we have defined. We then attribute this AST to perform the needed semantics.

Implemented in an AG environment, our type checking system automatically guarantees that the input and output types of each sub-process are right to the definition of a process work flow and of a certification, and do not break such definitions, while also managing to automatically create low-level implementations of certifications in the form of `Perl` scripts.

9.3 Portal

Using the combinator language presented in the previous section, we developed a web portal for the analysis and certification of OSS that aims at improving on these three issues. The portal works as a repository for tools that analyze source code.

Several projects have focused on the analysis and assessment of software, being the `Squale` project [Squale, 2014], QSOS [QSOS, 2014] and the `Alitheia Core` [Core, 2014] important examples of this. In comparison with

our work, we believe that potential users of these systems see their extensibility and improvement limited by custom schemas of information or domain-specific languages for plug-ins development. This is either because these projects are based on assessment models for OSS, or because they create unified storage systems or even because they imply the usage of frames of reference to create an evaluation that often depends on axis of criteria.

Our solution allows a wide range of tools based on different programming languages and techniques to be imported into our portal, taken that such tools are capable of running as bash tools and that they receive information through the standard `STDIN` and `STDOUT` Unix's streams. We believe this includes a significant amount of already existing potential tools.

What is more, through the use of our DSL, virtually any tool in our portal can be connected to other tools to create a flow of information (as long as the input and output types of two chained tools match), easily allowing the introduction of software assessments and the extension of such of assessments.

The usage of our portal is heterogeneous in that it supports the analysis of any programming language and distributed in the sense that it makes software analysis available in the web. Also, while already incorporating several predefined certifications, the portal makes it very simple for any user to re-arrange these certifications and to develop new ones: we designed and implemented a DSL that allows portal users to define, in a high level and abstract way, how certifications and software tools that analyze source code can be integrated and combined. This allows the creation of personalized analysis closely tied to the scope and nature of the necessary feedback.

The portal has been constructed out of about 320 lines of `JAVASCRIPT` and of circa 1500 lines of `HTML` + `PHP` code. In fact, from these 1500 lines, around 125 interface with a simple database for storing information related to the certifications of the portal, which itself includes 3 tables and 12 records. The DSL that the portal provides for re-arranging certifications and components was developed out of around 400 lines of `Haskell` code.

Once a certification for a particular programming language is available, users just need to upload a file in that language to analyze it. Having done so, our portal only presents as certification options for it the ones that match

its type. This means that, for example, having uploaded a `Haskell` file, users will only see the certifications that are available for `Haskell`.

More information about this portal and the supporting framework can be found in [Martins et al., 2014a] and [Martins et al., 2013].

9.4 Conclusions

We believe the advantages of the system we have presented here are two fold: first, the combinators create an intuitive and simple yet powerful environment to create not only certifications but also processes work flows in general, while ensuring their validation.

Secondly, our AG approach can be easily transformed to carry any change needed, both by the definitions of certifications or by processes and their work flows. This mechanism is modular, easily extensible and upgrades on code generation or type checking are performed of attributes, where new features and functionalities are easy to design and implement in a modular and concise way.

We also presented a portal for analyzing source code artifacts and providing information reports about them. Our portal supports various analysis scenarios and is able of dealing with programs expressed in different programming languages. The DSL for process management is implemented in our portal.

Chapter 10

Conclusions

In this thesis we have presented an embedding of modern AG extensions using a concise and elegant zipper-based implementation. The overview of what has been achieved can be seen in Figure 10.1.

We started by embedding canonical AGs and continued by defining various extensions, including combining higher order and circularity. 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.

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

We have also seen that the solution we presented varies from traditional approaches to implement multi-traversal algorithms in a functional setting. Our AG-based solution does not require extra effort to implement intermediate data types or to control complex functions scheduling, when comparing to strict programs, and does not require the lazy mechanisms of circular ones.

We have also shown how rewrite rules can be used to specify forward

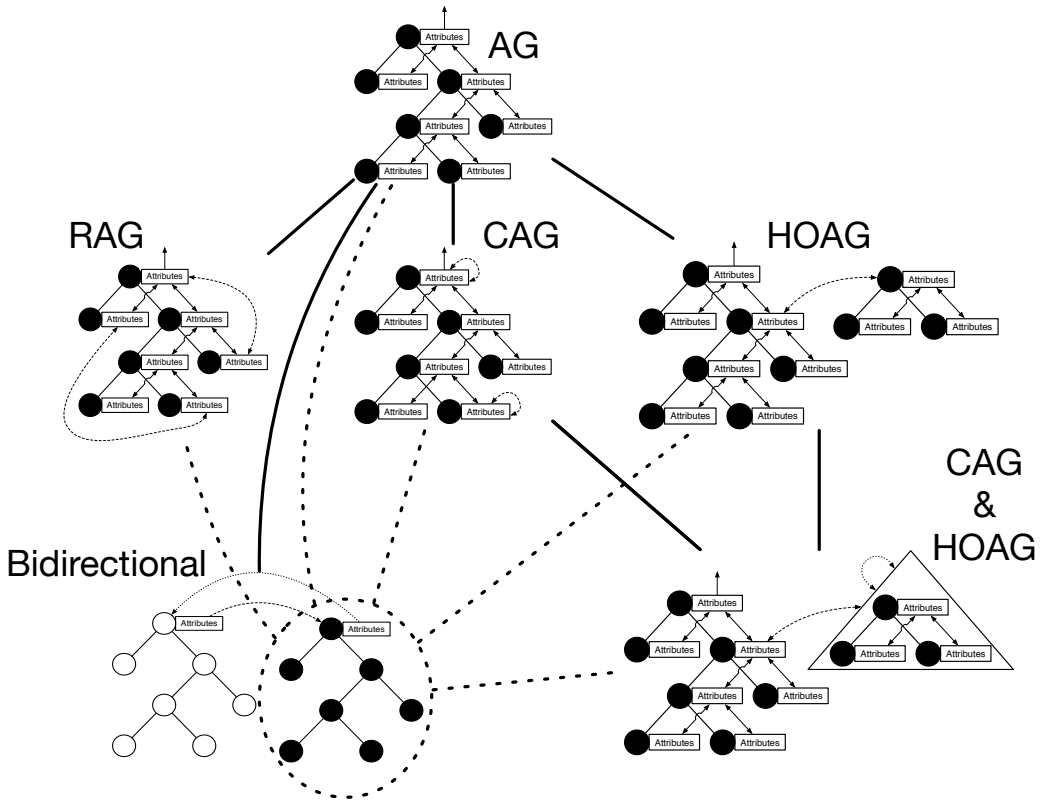


Figure 10.1: An overview of all the features we were able to embedded using our zipper-based technique.

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 bidirectional system supports are completely automatic for many applications, freeing the programmer of having to write complex attribute equations that have to perform multiple pattern matching, manage both the links back and their types or prioritizing transformations. The data types resulting from the transformations on the bidirectional system can themselves be the subjects of all the different attribute grammar techniques we have presented.

10.1 Processing LET

In this section, we would like to go back to the example that was used throughout the thesis to illustrate the involved concepts. All the semantics we have described through AGs are visible in Figure 10.2.

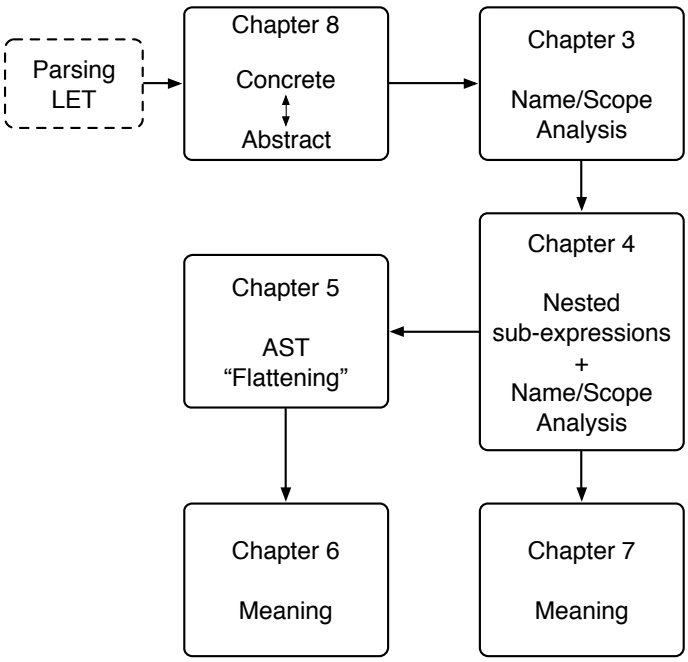


Figure 10.2: Overview of the examples presented on this thesis.

With the examples provided, we have automatic mechanisms available to transform from the concrete to the abstract representation of `LET`, we have extended the language with nested expressions, we have programs to perform name and scope analysis and we presented two different strategies to calculate the meaning of a `LET` program.

We never defined a way to parse `LET`. The biggest reason is that, since parsing is such an important step on language analysis, there are multiple techniques and tools to achieve it. For `Haskell` we have for example [Bienia et al., 2008] or [Viera et al., 2008], and a search on the `Haskell` repository `Hackage`[1] returns dozens of packages related to parsing.

[1] `https://hackage.haskell.org`

We believe this is a proof of the potential of the approach we are presenting. The simple and small examples we provided with the main aim of showcasing the techniques we have developed is enough to create an interpreter for a small programming language, that despite its simplicity creates interesting challenges.

10.2 Limitations of this Approach

Our approach has some limitations when comparing them to other AG system, being them embedded or custom AG systems.

10.2.1 References in HOAGs

One limitation of our system is regarding the use of references. We have shown in Chapter 4 that these are possible, and we even provided an example of an AG using references, but we never used references in an HOAG. For example, the symbol table from Chapter 6 could be a higher order tree with references instead of actual values. The reason we never did this has to do with the generic zipper library we use.

The zipper library in [Adams, 2010] allows wrapping of structures in `Haskell` as long as they are instantiable by the classes *Data* and *Typeable*, but the zipper themselves are not. In practice, what this means is that when trying to write something like:

$$\begin{array}{lll} \textbf{let } \mathit{zipper} & = \mathit{to}_{\mathit{Zipper}}\ 4 :: \mathit{Zipper\ Integer} & \text{-- OK} \\ \textbf{let } \mathit{wrap\_zipper} & = \mathit{to}_{\mathit{Zipper}}\ \mathit{zipper} & \text{-- ERROR} \end{array}$$

the second line will produce an error. While an Integer (in this case 4) is instantiated by `Data` and `Typeable`, which means we can wrap it inside a zipper with $\mathit{to}_{\mathit{Zipper}}$, creating something of type *Zipper Integer*, the same is not true for *zipper*, which means $\mathit{to}_{\mathit{Zipper}}\ \mathit{zipper}$ will fail.

This has limitations in our approach, as it means higher order attributes cannot contain references (which if the reader recalls are themselves zippers).

10.2.2 Repetitive Attribute Evaluation

Another drawback of our approach is that when defining AGs, attributes are calculated every time they are requested, which degrades the performance of our system.

Most systems have mechanism to deal with repetitive attribute evaluation. In `JastAdd` attributes are evaluated on demand, and if the user declares them as "lazy", their values are cached. In `Kiama` attributes are evaluated when they are demanded for a specific node. The most common kind of attributes are cached so their value will not be recalculated, but there is the option of having uncached attributes. `Silver` has similar technology built in. Creating a specific AG system has the advantage of customizing attribute evaluators and define their precise behavior.

In our embedding memoization is not easy to achieve, since we are expressing attribute grammars in a lazy setting. The combination of laziness and memoization is still an open problem, and modern compilers such as `GHC` still lacks support for it.

This embedding inherits the disadvantage of being recalculated as often as they are needed, but inherits the advantage of being calculated lazily without any additional encoding, as this is the functional nature of `Haskell`. In practice this means that if an attribute is defined but is not needed by the grammar, it is never calculated.

Other embeddings in `Haskell` ([de Moor et al., 2000a] and [Viera and Swierstra, 2012]) do not have recalculation of attributes as a problem, because their AGs are ultimately expressed as circular lazy programs [Bird, 1984]. However, this means they can not have some extensions, such as circular AGs.

As we have stated in the introduction to this work, embedding a language in a host language makes it inherit its advantages and disadvantages, and this is a perfect example of this happening.

10.2.3 Language Extensions

In our setting we use data types in `Haskell` to define grammars (and subsequently, languages), on which we define attribute computations as functions. The attribute computations point to children through their numeric position in the tree node. For example, for the data type:

$$\textbf{data}\ \mathit{Let} = \mathit{Let}\ \mathit{Dcls}\ \mathit{Expr}$$

we define an attribute that pretty prints the AST as:

$$\begin{aligned}&\mathit{p\_print}\ \mathit{ag} = \textbf{case}\ (\mathit{constructor}\ \mathit{ag})\ \mathit{as}\\&\quad \texttt{"Let"} = (\mathit{p\_print}\ \mathit{ag}\ .\$\ 1) +\!\!+ (\mathit{result}\ \mathit{ag}\ .\$\ 2)\end{aligned}$$

where we define the *p_print* attribute as being the result of its computation on the first child plus the result of the second child. However, if we want to insert type information, as in:

$$\textbf{data}\ \mathit{Let} = \mathit{Let}\ \mathit{Type}\ \mathit{Dcls}\ \mathit{Expr}$$

the previously defined attribute is automatically invalid, because now we are asking for *p_print* on a *Type* (first child) and *result* on a *Dcls* (second child). Generalizing, this means that insertion of symbols in a grammar, ans subsequent adaptation of the corresponding data type may make all the attributes specified for that production invalid.

10.3 Future Work

We believe the embedding for AGs we have presented closely resembles an attribute grammar and as the expressive power of embedding AG systems. However there are still a few points we would like to address.

First, we would like to improve attribute definition by referencing non-terminals instead of (numeric) positions on the right-hand side of productions. This would make attributes easier to write, and extending productions would be simplified as long as the names of the left-hand side symbols are maintained the same.

Currently, in our setting, attributes are recalculated every time they are necessary during the computations of an AG. Memoization would be a great solution for this problem. The `Haskell` compiler GHC does not support it, but `Haskell` packages such as `Memotrie`[2] gives us hope of being capable of achieving this.

Regarding the bidirectional system we have presented, there are few areas we would want to address. The first would be the generalization of this work to other attribute grammar systems besides `Silver` and our embedding. We would like to generate AG specifications for other systems such as `Kiama`, `LRC` or `JastAdd` so as to provide these techniques to a wider audience in the attribute grammar community. The bidirectional approach should also be evaluated on a number of mainstream syntactically rich languages.

We would also like to, wherever possible, benchmark our embedding against other AG embeddings and systems. This test is hard to do, as it is difficult to directly compare systems as wide as the ones we have shown. For example, we have specific AG systems such as `Silver`, `Kiama`, `LRC` or `UUAG` [Swierstra et al., 2004], embeddings in object-oriented environments such as `Kiama` or embeddings in functional environments, such as [de Moor et al., 2000a] or the embedding we are presenting in this work. To make it harder, not all these systems support the same extensions on AGs.

[2] `http://hackage.haskell.org/package/MemoTrie`

Bibliography

Adams, M. D. (2010). Scrap Your Zippers: A Generic Zipper for Heterogeneous Types. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 13–24, New York, USA. Association for Computing Machinery (ACM).

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.

Alblas, H. (1991). Introduction to attribute grammars. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag.

Badouel, E., Fotsing, B., and Tchougong, R. (2007). Yet another implementation of attribute evaluation. Research Report RR-6315, INRIA.

Badouel, E., Fotsing, B., and Tchougong, R. (2011). Attribute Grammars As Recursion Schemes over Cyclic Representations of Zippers. *Electronic Notes in Theoretical Computer Science*, 229(5):39–56.

Badouel, E., Tchougong, R., Nkuimi-Jugnia, C., and Fotsing, B. (2013). Attribute Grammars As Tree Transducers over Cyclic Representations of Infinite Trees and Their Descriptional Composition. *Theory on Computer Science*, 480:1–25.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and*

Compilation Techniques, PACT '08, pages 72–81, New York, USA. Association for Computing Machinery (ACM).

Bird, R. (1998). *Introduction to Functional Programming using Haskell (2nd Edition)*. Prentice-Hall.

Bird, R. S. (1984). Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250.

Bohannon, A., Pierce, B. C., and Vaughan, J. A. (2006). Relational Lenses: A Language for Updatable Views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, USA. Association for Computing Machinery (ACM).

Boyland, J. T. (2005). Remote Attribute Grammars. *Magazine Communications of the ACM*, 52(4):627–687.

Bransen, J., Dijkstra, A., and Swierstra, S. D. (2014). Lazy Stateless Incremental Evaluation Machinery for Attribute Grammars. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 145–156, New York, USA. Association for Computing Machinery (ACM).

Brown, D., Levine, J., and Mason, T. (1992). *lex & yacc (2nd Edition)*. O'Reilly Media.

Campos, M. D. and Barbosa, L. S. (2009). Implementation of an Orchestration Language as a Haskell Domain Specific Language. *Electronic Notes Theoretic Compututer Science*, 255:45–64.

Carle, A. and Pollock, L. (1996). On the Optimality of Change Propagation for Incremental Evaluation of Hierarchical Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29.

Chirica, L. M. and Martin, D. F. (1979). An order-algebraic definition of knuthian semantics. *Mathematical systems theory*, 13(1):1–27.

Core, A. (May 2014). `http://www.sqo-oss.org`.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional Transformations: A Cross-Discipline Perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 260–283. Springer-Verlag.

Czarnecki, K. and Helsen, S. (2006). Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal - Model-driven software development*, 45(3):621–645.

de Moor, O., Backhouse, K., and Swierstra, S. D. (2000a). First-class Attribute Grammars. *Informatica (Slovenia)*, 24(3).

de Moor, O., Peyton-Jones, S., and Van Wyk, E. (2000b). Aspect-Oriented Compilers. In Czarnecki, K. and Eisenecker, U. W., editors, *Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 121–133. Springer-Verlag.

Dijkstra, A., Fokker, J., and Swierstra, S. D. (2009). The Architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 93–104, New York, USA. Association for Computing Machinery (ACM).

Doets, K. and van Eijck, J. (2004). *The Haskell Road to Logic, Maths and Programming (2nd Edition)*. College Publications.

Ekman, T. and Hedin, G. (2006). Modular Name Analysis for Java Using Jastadd. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 422–436, Berlin, Heidelberg. Springer-Verlag.

Ekman, T. and Hedin, G. (2007). The Jastadd Extensible Java Compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 1–18, New York, USA. Association for Computing Machinery (ACM).

Farrow, R. (1986). Automatic Generation of Fixed-point-finding Evaluators for Circular, but Well-defined, Attribute Grammars. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 85–98, New York, USA. Association for Computing Machinery (ACM).

Fernandes, J. P. (2004). Generalized LR Parsing in Haskell. Research report, Universidade do Minho.

Fernandes, J. P. and Saraiva, J. (2007). 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, pages 102–111. Association for Computing Machinery (ACM).

Fernandes, J. P., Saraiva, J., Seidel, D., and Voigtländer, J. (2011). Strictification of Circular Programs. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 131–140, New York, USA. Association for Computing Machinery (ACM).

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Transactions on Programming Languages and Systems*, 29(3).

Gill, A. (2014). Domain-specific Languages and Code Synthesis Using Haskell. *Magazine Communications of the ACM*, 57(6):42–49.

Gray, R. W., Levi, S. P., Heuring, V. P., Sloane, A. M., and Waite, W. M. (1992). Eli: A Complete, Flexible Compiler Construction System. *Magazine Communications of the ACM*, 35(2):121–130.

Hedin, G. (1999). Reference Attributed Grammars. In *Proceedings of the 2nd Workshop on Attribute Grammars and their Applications*, WAGA '99, pages 153–172. INRIA Rocquencourt.

Hedin, G. (2011). An Introductory Tutorial on JastAdd Attribute Grammars. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, pages 166–200, Berlin, Heidelberg. Springer-Verlag.

Hibberd, M., Lawley, M., and Raymond, K. (2007). Forensic Debugging of Model Transformations. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, MODELS'07, pages 589–604. Springer-Verlag.

Hoover, R. and Teitelbaum, T. (1986). Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 39–50, New York, USA. Association for Computing Machinery (ACM).

Hu, Z., Mu, S.-C., and Takeichi, M. (2004). A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 178–189, New York, USA. Association for Computing Machinery (ACM).

Hudak, P. (1996). Building Domain-specific Embedded Languages. *ACM Computer Surveys*, 28(4es).

Hudak, P. (2000). *The Haskell School of Expression - Learning Functional Programming through Multimedia*. Cambridge University Press.

Huet, G. (1997). The Zipper. *Journal of Functional Programming*, 7(5):549–554.

Johnsson, T. (1987). Attribute grammars as a functional programming paradigm. In Kahn, G., editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag.

Jones, L. G. (1990). Efficient Evaluation of Circular Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462.

Jones, S. P., Gordon, A., and Finne, S. (1996). Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308, New York, USA. Association for Computing Machinery (ACM).

Jones, S. P., Hughes, J., et al. (1999). *Report on the Programming Language Haskell 98*. `http://www.haskell.org/definition/haskell98-report.pdf`.

Jouault, F. and Kurtev, I. (2006). Transforming Models with ATL. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138. Springer-Verlag.

Jourdan, M., Parigot, D., Julié, C., Durin, O., and Bellec, C. L. (1990). Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 209–222, New York, USA. ACM.

Kaiser, G. E. and Kaplan, S. M. (1993). Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multiuser Software Development Environments. *ACM Transactions on Software Engineering and Methodology*, 2(1):47–92.

Kaminski, T. and Van Wyk, E. (2012). Integrating Attribute Grammar and Functional Programming Language Features. In *Proceedings of the 4th International Conference on Software Language Engineering*, SLE '11, pages 263–282, Berlin, Heidelberg. Springer-Verlag.

Kastens, U. (1980). Ordered attributed grammars. *Acta Informatica*, 13(3):229–256.

Kastens, U. and Schmidt, C. (2002). VL-Eli:: A Generator for Visual Languages. *Electronic Notes in Theoretical Computer Science*, 65(3):139 – 143. Second Workshop on Language Descriptions, Tools and Applications (Satellite Event of {ETAPS} 2002).

Kitchin, D., Quark, A., Cook, W., and Misra, J. (2009). The Orc Programming Language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg. Springer-Verlag.

Knuth, D. E. (1968). Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145. Corrections in **5**(1971) pp. 95-96.

Kuiper, M. and Swierstra, S. D. (1987). Using Attribute Grammars to Derive Efficient Functional Programs. In *Computing Science in the Netherlands*.

Kuiper, M. F. and Saraiva, J. (1998). Lrc - A Generator for Incremental Language-Oriented Tools. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, pages 298–301, London, UK. Springer-Verlag.

Lämmel, R. and Jones, S. P. (2003). Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, USA. Association for Computing Machinery (ACM).

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press.

Luković, I., ao Varanda Pereira, M. J., Oliveira, N., da Cruz, D., and Henriques, P. R. (2011). A DSL for PIM Specifications: Design and Attribute Grammar based Implementation. *Computer Science and Information Systems*, 8(2):379–403.

Magnusson, E. and Hedin, G. (2000). Program Visualization Using Reference Attributed Grammars. *Nordic Journal of Computing*, 7(2):67–86.

Martins, P., Carvalho, N., Fernandes, J. P., Almeida, J. J., and Saraiva, J. (2013). A Framework for Modular and Customizable Software Analysis. In *Proceedings of the 13th International Conference on Computational Science and Its Applications*, volume 7972 of *ICCSA '13*, pages 443–458. Springer-Verlag.

Martins, P., Fernandes, J. P., and Saraiva, J. (2012). A Purely Functional Combinator Language for Software Quality Assessment. In *Proceedings of the Symposium on Languages, Applications and Technologies*, volume 21 of *SLATE '12*, pages 51–69. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Martins, P., Fernandes, J. P., and Saraiva, J. (2014a). A Web Portal for the Certification of Open Source Software. In *Proceedings of the 6th International Workshop on Foundations and Techniques for Open Source Software Certification*, volume 7991 of *OPENCERT '12*, pages 244–260. Springer-Verlag.

Martins, P., Saraiva, J., Fernandes, J. P., and Van Wyk, E. (2014b). Generating Attribute Grammar-based Bidirectional Transformations from Rewrite Rules. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 63–70. Association for Computing Machinery (ACM).

Matsuda, K., Hu, Z., Nakano, K., Hamana, M., and Takeichi, M. (2007). Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 47–58, New York, USA. Association for Computing Machinery (ACM).

Mena, A. S. (2014). *Beginning Haskell: A Project-Based Approach*. Apress.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4).

Mernik, M., Lenic, M., Avdicausevic, E., and Zumer, V. (2000). Multiple Attribute Grammar Inheritance. *Informatica (Slovenia)*, 24(3).

Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2010). Iterative Type Inference with Attribute Grammars. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 43–52, New York, USA. Association for Computing Machinery (ACM).

Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2011). Stepwise Evaluation of Attribute Grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 5:1–5:8, New York, USA. Association for Computing Machinery (ACM).

Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML - Revised*. The MIT Press.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA.

O'Sullivan, B., Goerzen, J., and Stewart, D. (2008). *Real World Haskell*. O'Reilly Media.

Paakki, J. (1995). Attribute Grammar Paradigms&Mdash;a High-level Methodology in Language Implementation. *ACM Compututer Surveys*, 27(2):196–255.

Parr, T. (2013). *The Definitive ANTLR 4 Reference (2nd Edition)*. Pragmatic Bookshelf.

Pennings, M., Swierstra, S. D., and Vogt, H. (1992). Using Cached Functions and Constructors for Incremental Attribute Evaluation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, PLILP '92, pages 130–144, London, UK. Springer-Verlag.

Pennings, M. C. (1994). *Generating incremental attribute evaluators*. PhD thesis, Computer Science, Utrecht University.

QSOS (May 2014). `http://www.qsos.org`.

Reps, T. and Demers, A. (1987). Sublinear-space Evaluation Algorithms for Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 9(3):408–440.

Reps, T. W. and Teitelbaum, T. (1989). *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag.

Rodeh, M. and Sagiv, M. (1999). Finding Circular Attributes in Attribute Grammars. *Journal of the ACM*, 46(4):556–ff.

Saraiva, J. (1999). *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University.

Saraiva, J. (2002). Component-Based Programming for Higher-Order Attribute Grammars. In Batory, D., Consel, C., and Taha, W., editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 268–282. Springer-Verlag.

Saraiva, J., Kuiper, M., and Swierstra, S. D. (1997). Specializing Trees for Efficient Functional Decoration. In Leuschel, M., editor, *Workshop on Specialization of Declarative Programs and its Applications*, ILPS '97, pages 63–72. (Also available as Technical Report CW 255, Department of Computer Science, Katholieke Universiteit Leuven, Belgium).

Saraiva, J. and Swierstra, S. D. (1999a). Data structure free compilation. In *Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 1–16. Springer-Verlag.

Saraiva, J. and Swierstra, S. D. (1999b). Data Structure Free Compilation. In Jähnichen, S., editor, *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag.

Saraiva, J. and Swierstra, S. D. (1999c). Generic Attribute Grammars. In Parigot, D. and Mernik, M., editors, *Proceedings of the 2nd Workshop on*

Attribute Grammars and their Applications, WAGA '99, pages 185–204. INRIA Rocquencourt.

Saraiva, J. and Swierstra, S. D. (2003). Generating Spreadsheet-like Tools from Strong Attribute Grammars. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 307–323, New York, USA. Springer-Verlag.

Sasaki, A. and Sassa, M. (2004). Circular attribute grammars with remote attribute references and their evaluators. *New Generation Computing*, 22(1):37–60.

Schäfer, M., Ekman, T., and Moor, O. (2009). Formalising and Verifying Reference Attribute Grammars in Coq. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 143–159, Berlin, Heidelberg. Springer-Verlag.

Sloane, A. M., Kats, L. C. L., and Visser, E. (2010). A Pure Object-Oriented Embedding of Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 253(7):205–219.

Söderberg, E. (2012). *Contributions to the Construction of Extensible Semantic Editors*. PhD thesis, Lund University.

Söderberg, E. and Hedin, G. (2013). Circular Higher-Order Reference Attribute Grammars. In Erwig, M., Paige, R. F., and Van Wyk, E., editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 302–321. Springer-Verlag.

Squale (May 2014). `http://www.squale.org`.

Stallman, R. M. and Community, G. D. (2009). *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace Independent Publishing Platform.

Stevens, P. (2008). A Landscape of Bidirectional Model Transformations. In Lämmel, R., Visser, J., and Saraiva, J., editors, *Generative and Transformational Techniques in Software Engineering II*, pages 408–424. Springer-Verlag, Berlin, Heidelberg.

Swierstra, S. D., Alcocer, P. R. A., and Saraiva, J. (1999). Designing and Implementing Combinator Languages. In Swierstra, S. D., Oliveira, J. N., and Henriques, P. R., editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer-Verlag.

Swierstra, S. D., Baars, A., and Löh, A. (2004). The UU-AG Attribute Grammar System. `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`.

Teitelbaum, T. and Chapman, R. (1990). Higher-order Attribute Grammars and Editing Environments. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 197–208, New York, USA. Association for Computing Machinery (ACM).

Terry, P. D. (2005). *Compiling with C# and Java*. Addison-Wesley.

Uustalu, T. and Vene, V. (2005). Comonadic functional attribute evaluation. In *Trends in Functional Programming*, pages 145–162. Intellect Books.

van den Brand, M. G. J., Scheerder, J., Vinju, J. J., and Visser, E. (2002). Disambiguation Filters for Scannerless Generalized LR Parsers. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 143–158, London, UK. Springer-Verlag.

Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2008). Silver: An Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116.

Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in Attribute Grammars for Modular Language Design. In *Pro-*

ceedings of the 11th International Conference on Compiler Construction, CC '02, pages 128–142, London, UK. Springer-Verlag.

Viera, M. (2013). *First Class Syntax, Semantics, and Their Composition*. PhD thesis, Utrecht University.

Viera, M. and Swierstra, S. D. (2012). Attribute Grammar Macros. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 150–164, Berlin, Heidelberg. Springer-Verlag.

Viera, M., Swierstra, S. D., and Lempsink, E. (2008). Haskell, Do You Read Me?: Constructing and Composing Efficient Top-down Parsers at Runtime. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 63–74, New York, USA. Association for Computing Machinery (ACM).

Viera, M., Swierstra, S. D., and Swierstra, W. (2009). Attribute Grammars Fly First-class: How to Do Aspect Oriented Programming in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 245–256, New York, USA. Association for Computing Machinery (ACM).

Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 131–145, New York, NY, USA. Association for Computing Machinery (ACM).

Yakushev, A. R., Holdermans, S., Löh, A., and Jeuring, J. (2009). Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 233–244, New York, USA. Association for Computing Machinery (ACM).

Yellin, D. M. (1988). *Attribute Grammar Inversion and Source-to-source Translation*. Springer-Verlag.

Index