First-class Attribute Grammars

Oege de Moor[1], Kevin Backhouse[1] & S. Doaitse Swierstra[2]

1: Programming Research Group
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{oege,kevinb}@comlab.ox.ac.uk
2: Department of Computer Science
PO Box 80.089, 3508 TB Utrecht, The Netherlands
doaitse@cs.uu.nl

1. Introduction

This paper is a contribution to the ongoing quest for modular descriptions of language processors, with the specific aim of rapidly prototyping domain-specific languages [21]. Some might argue that this problem was solved in the eighties, with the development of a proliferation of language processors based on attribute grammars [11, 15, 22]. Others might argue that functional programming languages such as ML are adequate for the purpose, without any further extensions. We believe that functional programming languages do not offer enough specialised support for implementing compilers. However, attribute grammars are not in widespread use, despite their many advantages. This may be due to restrictions imposed by attribute definition languages, which are often less flexible than general purpose functional programming languages. Such general languages tend to yield descriptions that are compact, but they lack the dedicated structuring mechanisms of attribute grammars.

In this paper we initiate a systematic study of such structuring mechanisms, by giving them a compositional semantics. The semantics is expressed in the vocabulary of functional programming. Our semantics thus opens the way towards combining the powerful structuring mechanisms for attribute grammars with the flexibility of a general purpose programming language. In particular, it is easy to define new structuring operators in our semantics. Furthermore, because the semantics is a functional program, one immediately obtains a prototype for experimenting with newly defined features. Naturally the results of this paper do not stand on their own, and many of the ideas have been gleaned from the attribute grammar literature, in particular [5, 6, 16, 19, 20, 23, 26]. Especially the thesis by Stephen Adams [1] has been an inspiration for this work.

Attribute grammars and functional programming There exists a well-known encoding of attribute grammars into programming languages that have lazy evaluation [14, 18]. This encoding has been dismissed by others on the following grounds:

- Lazy evaluation is inherently inefficient, and therefore an attribute evaluator based on it must be inefficient.
- The resulting programs are highly convoluted and much less modular than standard attribute grammars.

The first objection has been refuted by the work of Augusteijn, who has built an attribute grammar evaluator based on lazy evaluation: he reports that its performance is on a par with other systems that do a sophisticated analysis of dependencies, and produce a schedule for the attribute computations based on that analysis. Augusteijn's system, named *Elegant*, has been widely used within Philips for implementing domain-specific languages [2]. Because our primary objective is a compositional semantics, the efficiency issue is not really important in the present context.

The second objection remains valid, however, and indeed the Elegant system suffers from this problem. Essentially, all attribute definitions have to be grouped by production. It is thus not possible to group all definitions for a single attribute in one place, and then specify how each rule contributes to the behaviour of a production. One cannot reuse the same set of attribution rules, and make them contribute to different productions. Elegant is particular in this respect: many other attribute grammar systems do allow such groupings, but only at a syntactic and not at a semantic level. If we wanted to provide the same functionality in a general purpose programming language, so that rules, productions and grammars are all first-class citizens of the language, we would have to give a type to each re-usable component.

The purpose of types is to guarantee the absence of certain run-time errors. In choosing an appropriate type system for composing attribute grammars from smaller components, we need to decide what run-time errors we wish to avoid. There are a number of common errors that are typically caught by attribute grammar systems: a mismatch between productions as used in the attribute definitions and in the context-free grammar, the use of an attribute that has not been defined, a cyclic dependency between attribute definitions, and the use of an attribute in a context that does not match its type. In this paper, we only aim to avoid the last kind of error.

The idea of embedding domain-specific languages directly into a more general *host language* is a buoyant area of research. Recent examples include languages for pretty-printing [13], reactive animation [9], and musical composition [12]. This paper adds the example of attribute grammars to that list. All these works, including our own, can be seen as providing an executable semantics for a domain-specific language. While studying semantics, one is not concerned with matters of concrete syntax, and indeed we shall defer the choice of appropriate notations to later work.

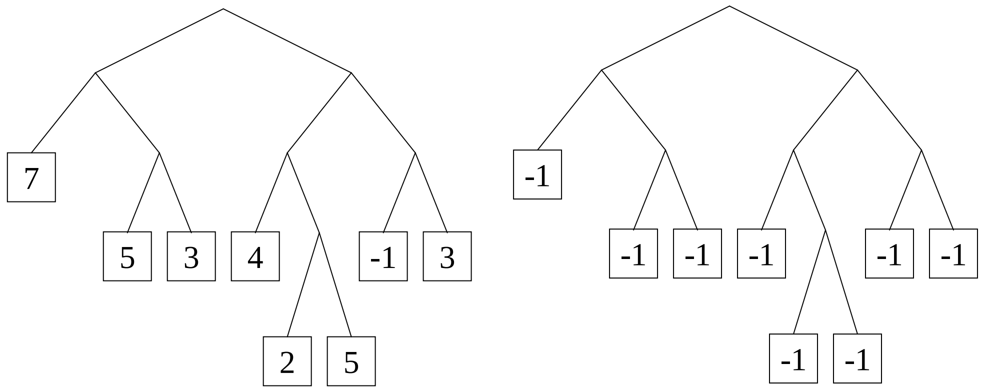
As argued by Swierstra *et al.* in [25], some of the above examples of embedded domain-specific languages could be more nicely structured in an attribute grammar style. In that paper, an attribute grammar preprocessor is used for achieving the desired structure. The present paper provides a semantics of that preprocessor.

Overview The structure of the paper is as follows. First we introduce a small attribute grammar example that we shall use throughout to illustrate the ideas. We show how we might simplify the attribute grammar by using "aspects". Next, we introduce our notation, which is loosely based on the lazy functional programming language Haskell [4]. The notation is illustrated by defining the basic types of trees, productions and attributes. They provide the preliminaries for discussing *families*, *rules* and *aspects*: the building blocks of our semantics. To illustrate these building blocks in a practical setting, we revisit our introductory example. We then show how easily they can be mapped into an executable implementation. Finally, we discuss directions for future work, in particular how we can provide more sophisticated static checks on the composed attribute grammar.

It is assumed that the reader has some degree of familiarity with a modern functional programming language, as well as the basic concepts of attribute grammars. The traditional encoding of attribute grammars in a lazy functional language is described by [14]. A passing acquaintance with this encoding will be helpful, but is not necessary. A good introduction to the style of functional definition in this paper can be found in [4].

2. An example: *repmin*

Consider binary trees, whose internal *nodes* are unlabelled, and whose *leaves* are labelled with integer values. We aim to replace all leaf values by the minimum leaf value. An example of this is given in



The input tree *The output tree*

Figure 1: An example of the use of *repmin*

Figure 1. This is known as the *repmin* problem, and it was first introduced by [3]. As noted by [18], the *repmin* problem is easily expressed as an attribute grammar, which we will now present in an anonymous but hopefully self-explanatory notation.

As a first step we introduce two synthesised attributes, named *ntree* (for *new tree*), and *locmin* (for *local minimum*). Furthermore there is one inherited attribute, named *gmin* (for *global minimum*). The strategy is to recursively compute the local minimum on all nodes. The global minimum equals the local minimum of the root. This value is broadcast to all the leaves. The new tree is then built recursively.

The production named *Root* rewrites the start symbol *Start* to *Tree*. The resulting tree of parent *Start* is the resulting tree of the child *Tree*. It is here that the global minimum is defined:

$$\begin{array}{lcl} \mathit{Root} & : & \mathit{Start} \rightarrow \mathit{Tree} \\ \mathit{Start.ntree} & = & \mathit{Tree.ntree} \\ \mathit{Tree.gmin} & = & \mathit{Tree.locmin} \end{array}$$

At each binary node, the local minimum is obtained by taking the minimum of both subtrees; the global minimum is broadcast from the parent to both children. Here and below, we use indices to refer to successive occurrences of the nonterminal *Tree*:

$$\begin{array}{lcl} \mathit{Node} & : & \mathit{Tree}_0 \rightarrow \mathit{Tree}_1 \ \mathit{Tree}_2 \\ \mathit{Tree}_0.\mathit{ntree} & = & \mathit{Node} \ \mathit{Tree}_1.\mathit{ntree} \ \mathit{Tree}_2.\mathit{ntree} \\ \mathit{Tree}_0.\mathit{locmin} & = & \mathit{min} \ \mathit{Tree}_1.\mathit{locmin} \ \mathit{Tree}_2.\mathit{locmin} \\ \mathit{Tree}_1.\mathit{gmin} & = & \mathit{Tree}_0.\mathit{gmin} \\ \mathit{Tree}_2.\mathit{gmin} & = & \mathit{Tree}_0.\mathit{gmin} \end{array}$$

Finally, the local minimum of a leaf is its value, and the new tree is a leaf with the global minimum as its value:

$$\begin{array}{lcl} \mathit{Leaf} & : & \mathit{Tree} \rightarrow \mathit{Val} \\ \mathit{Tree.ntree} & = & \mathit{Leaf} \ \mathit{Tree.gmin} \\ \mathit{Tree.locmin} & = & \mathit{Val.value} \end{array}$$

Despite the simplicity of this example, there is already quite a lot of tedious detail to take care of, most notably the copying of the *gmin* attribute from the root of the tree to the leaves. It is also a little annoying that the definition of each attribute is smeared out over several productions, making it difficult to see the flow of information at a glance. It is for that reason that practical attribute grammar systems provide better structuring mechanisms, of the kind that we shall discuss below.

Let us plod on, however, and consider how the above attribute grammar would have to be modified for a slightly different problem. Instead of replacing each leaf L by the global minimum, we aim to replace it by the number of times the global minimum occurred to the left of L in the inorder traversal of the tree. For this we introduce an additional chained attribute *count* that keeps track of that number. The new root production initialises the count to zero:

$$\begin{array}{lcl} \mathit{Root} & : & \mathit{Start} \rightarrow \mathit{Tree} \\ \mathit{Tree.count} & = & 0 \end{array}$$

At a node, we chain the count from left to right. As is often the case with chained attributes, there is some subtle punning going on with the names: the first mention of $\mathit{Tree}_0.\mathit{count}$ is the *inherited* attribute *count*, whereas its second occurrence is the *synthesised* attribute of the same name:

$$\begin{array}{lcl} \mathit{Node} & : & \mathit{Tree}_0 \rightarrow \mathit{Tree}_1\ \mathit{Tree}_2 \\ \mathit{Tree}_1.\mathit{count} & = & \mathit{Tree}_0.\mathit{count} \\ \mathit{Tree}_2.\mathit{count} & = & \mathit{Tree}_1.\mathit{count} \\ \mathit{Tree}_0.\mathit{count} & = & \mathit{Tree}_2.\mathit{count} \end{array}$$

Finally, at a leaf we compare the value to the global minimum, and if they coincide, the counter is incremented. We also redefine the computation of *ntree*:

$$\begin{array}{lcl} \mathit{Leaf} & : & \mathit{Tree} \rightarrow \mathit{Val} \\ \mathit{Tree.ntree} & = & \mathit{Leaf}\ \mathit{Tree.count} \\ \mathit{Tree.count} & = & \textbf{if}\ \mathit{Val.value} = \mathit{Tree.gmin}\ \textbf{then} \\ & & \quad \mathit{Tree.count} + 1 \\ & & \textbf{else} \\ & & \quad \mathit{Tree.count} \end{array}$$

To obtain a program for the modified *repmin* problem, we now have to paste these new definitions into the original grammar. This involves adding the new rules for *count* to each of the productions, and overriding the original definition of the *ntree* attribute in the *Leaf* production. Indeed, most attribute grammar systems treat structuring mechanisms in this syntactic way. Furthermore, they introduce syntactic abbreviations for common patterns such as chained attributes [16]. We aim to show how these structuring operations can be given a precise semantics.

In our semantic view, the only essential difference between the above two grammars is the presence of the *count* attribute: the rest of the semantics is shared. The overriding of the *ntree* attribute is modelled by making *ntree* a parameterised attribute. Furthermore, the semantics facilitates easy definitions of oft-occurring patterns (such as that of *chained attributes*, and *broadcasting of inherited attributes*). Making such patterns explicit removes a lot of the tedium involved in writing attribute grammars, and also makes them easier to read. It is the compositional semantics (of well known structuring mechanisms) that is the contribution of this paper. The fact that the semantics is an executable prototype is a pleasant side effect of expressing ourselves in a lazy functional programming language. Having an executable prototype makes it easy to experiment with new structuring operators, giving an opportunity to explore beyond the fixed vocabulary of typical attribute grammar systems.

With some syntactic sugar for increased readability, the new formulations of the *repmin* problem and its variation are as follows. First we introduce the inherited attribute *gmin*. It is introduced through a so-called *attribute aspect* that groups several definitions for an attribute together. This aspect stipulates that *gmin* is copied at the *Node* production, and a specialised definition is given at the root:

$$\begin{array}{rcl} gmins & = & \textbf{inherit}\ gmin \\ & & \textbf{copy at}\ Node \\ & & \textbf{define at}\ Root[Tree] :\ Tree.locmin \end{array}$$

Here, the notation *Root*[*Tree*] specifies that we are defining an attribute on the *Tree* nonterminal of the *Root* production.

One can also define attribute aspects for synthesised attributes. The default behaviour here is to collect multiple occurrences of the attribute from the children. In the case of the local minimum, we collect with the minimum function, and its value at a leaf is simply the original label:

$$\begin{array}{rcl} locmins & = & \textbf{synthesise}\ locmin \\ & & \textbf{collect with}\ min\ \textbf{at}\ Node \\ & & \textbf{define at}\ Leaf[Tree] :\ Val.value \end{array}$$

To cater for later variation, the construction of the new tree is parameterised by the attribute that we substitute for leaves:

$$\begin{array}{rcl} ntrees & = & \textbf{synthesise}\ ntree(a\ :\ Attribute\ Int) \\ & & \textbf{collect with}\ Node\ \textbf{at}\ Node \\ & & \textbf{define at}\ Leaf[Tree] :\ Leaf\ \ Tree.a \\ & & \qquad\qquad\quad Root[Start] :\ Tree.ntree \end{array}$$

Finally, the chained counter has special definitions in two productions. It is inherited in *Root*, and synthesised in *Leaf*:

$$\begin{array}{rcl} counts & = & \textbf{chain}\ count \\ & & \textbf{define at}\ Root[Tree] :\ 0 \\ & & \qquad\qquad\quad Leaf[Tree] :\ \textbf{if}\ Val.value\ =\ Tree.gmin\ \textbf{then} \\ & & \qquad\qquad\qquad\qquad\qquad\quad count + 1 \\ & & \qquad\qquad\qquad\qquad\qquad \textbf{else} \\ & & \qquad\qquad\qquad\qquad\qquad\quad count \end{array}$$

The solution to the original problem is now obtained by assembling the above aspects with the following Haskell expression:

$$repmin_0\ =\ compiler\ [gmins, locmins, ntrees\ gmin]\ ntree$$

The final argument indicates that we want to return the *ntree* attribute as the result of compilation. The more complicated variation of the *repmin* problem is assembled by including the counter:

$$repmin_1\ =\ compiler\ [gmins, locmins, ntrees\ count, counts]\ ntree$$

We should stress that each of the attribute aspects *gmins*, *locmins*, *ntrees* and *counts* are first-class values that can be passed as parameters and returned as results. To define precisely what those values are is the goal of the remainder of this paper.

3. Preliminaries: Trees, Productions and Attributes

To set the scene, and to introduce some Haskell vocabulary through familiar concepts, we start by defining trees, productions and attributes. Most of these definitions are extremely straightforward. It is only in our definition of attributes that we have to exercise some foresight. This will facilitate easy composition at a later stage. Readers who are familiar with Haskell may wish to skim the subsection on attributes, and then proceed to the next section, which is the core of the paper.

3.1. Trees

For simplicity, our attribute grammars will operate on a rather primitive kind of tree, whose type is independent of the underlying context-free grammar. As said in the introduction, that makes our semantic definitions simpler, but it does carry the risk of run-time errors when an attribute grammar is applied to a particular tree. A safer approach would be to define a separate type of tree for each grammar.

A tree is either a *Fork* labelled with a value of type α and a list of descendants that are also trees, or it is a *Val* labelled with a β:

$$\mathbf{data}\ \mathit{Tree}\ \alpha\ \beta\ =\ \mathit{Fork}\ \alpha\ [\mathit{Tree}\ \alpha\ \beta]\ |\ \mathit{Val}\ \beta$$

Typically, the type α represents the names of productions, and β is the type of attributions that were computed by the scanner or parser. The most common type of tree is therefore *Tree ProdName Attrs*, where *ProdName* denotes the type of names of productions, and *Attrs* that of attributions. Both of these types will be formally defined below. Sometimes it is convenient to vary the instantiations of α and β in the definition of trees, however, and that is why we abstract from the concrete type. An example where that flexibility will come in handy is the definition of a function that decorates a tree with all relevant attribute values.

In our running example, we have a grammar with three productions named *Root*, *Node* and *Leaf*. Together these names make up the data type of production names that may occur at *Fork* nodes of a tree:

$$\mathbf{data}\ \mathit{ProdName}\ =\ \mathit{Root}\ |\ \mathit{Node}\ |\ \mathit{Leaf}$$

3.2. Attributes and attributions

An *attribution* is a finite mapping from attribute names to attribute values. We shall exercise a little notational freedom when discussing finite mappings, and write $A \mapsto B$ for the set of finite maps from A to B. Accordingly, the type of attributions is defined:

$$\mathbf{type}\ \mathit{Attrs}\ =\ \mathit{AttrName}\ \mapsto\ \mathit{AttrValue}$$

Note that in contrast to previous types (which were new types, introduced with the Haskell keyword **data**) this type is merely an abbreviation for an existing type (which is indicated by using **type** in lieu of **data**). The choice to model attributions as finite maps implies that we cannot guarantee, by exploiting the type system of Haskell, that certain names are present in an attribution: such a check could have been enforced by modelling attributions through record types. Note also that all attributes map to values of the same type, namely *AttrValue*. As we shall see below, *AttrValue* is defined as the disjoint union of all possible attribute types in a particular grammar.

We shall often write $\{\,(n_0, v_0),\ (n_1, v_1),\ \ldots\ ,\ (n_{k-1}, v_{k-1})\,\}$ for the attribution that sends each name n_i to the value v_i. Strictly speaking this is not valid Haskell syntax, but it will ease the presentation of concrete examples.

It is our goal to make all concepts in our semantics composable, and that entails introducing a union, join or merge operation wherever we can. In the case of attributions, the obvious choice is the *join* of finite maps. For finite maps f and g, the join $f \oplus g$ is defined by:

$$\begin{aligned}(f\ \oplus\ g)\ x\ &=\ f\ x,\ \mathbf{if}\ x\ \in\ \mathit{domain}\ f\\ &=\ g\ x,\ \mathbf{otherwise}\end{aligned}$$

In this definition, we are again taking a notational liberty, namely writing application of finite maps as ordinary function application. In Haskell, a special operator has to be introduced. Furthermore, in

Haskell, application of a finite map to an element outside its domain will result in a run-time error. Note that the join operator is associative, and it has an identity element, namely the empty map.

While we have shown how to combine attributions, as yet we do not have a way of putting elements (*i.e.* (name,value) pairs) into an attribution. We define such embedding functions, one for each attribute, along with the corresponding projection. In fact, we take such an embedding/projection pair as the *definition* of an attribute. To wit, the type of attributes whose values are of type α is:

$$\mathbf{type}\ At\ \alpha\ =\ (\alpha\ \rightarrow\ Attrs,\ Attrs\ \rightarrow\ \alpha)$$

The first component of such a pair is the embedding, and the second is the projection:

$$\begin{aligned}
&embed &&::\ At\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ Attrs\\
&embed\ (e, p) &&=\ e\\
\\
&project &&::\ At\ \alpha\ \rightarrow\ Attrs\ \rightarrow\ \alpha\\
&project\ (e, p) &&=\ p
\end{aligned}$$

We shall ensure that for any attribute a, we have $project\ a\ \cdot\ embed\ a\ =\ id$. The opposite composition $embed\ a\ \cdot\ project\ a$ will usually not be the identity, because it always produces an attribution with only a single element.

One way to think of the expression $project\ a$ is as the function that maps a grammar symbol S to $S.a$: we project the a attribute from the attribution associated with S. Conversely, the embedding is what is used to define the attribute of a grammar symbol. Admittedly it may appear a little odd to define attributes in this way, but by encoding them as an embedding/projection pair, we avoid having to pass attribute names separately to many of the functions defined below.

Attributes are created using the function $mkAt$. It takes an attribute name, an embedding from α into the type of attribute values, and a coercion that goes in the opposite direction. The result is an attribute of type $At\ \alpha$:

$$\begin{aligned}
&mkAt &&::\ (AttrName,\ \alpha\ \rightarrow\ AttrValue,\ AttrValue\ \rightarrow\ \alpha)\ \rightarrow\ At\ \alpha\\
&mkAt\ (n, e, p) &&=\ (\ \lambda\ a\ \rightarrow\ \{(n, e\ a)\},\\
& &&\qquad \lambda\ as\ \rightarrow\ p\ (as\ n)\ )
\end{aligned}$$

That is, to embed an attribute value we wrap it in a singleton map, that only maps the name n to the value $e\ a$. Conversely, given an attribution as, we look up the corresponding value and project it to the type a.

In the running example, there are five attributes in all. First, there is the integer valued attribute of leaves – this is filled in by the scanner when the tree is read in. Furthermore, we have the local minimum, the global minimum, the newly created tree, and the counter. For each of these attributes, we introduce an identifier:

$$\mathbf{data}\ AttrName\ =\ ValueId\ |\ LocminId\ |\ GminId\ |\ NtreeId\ |\ CountId$$

The type of attribute values is the disjoint union of values for each of these five attributes. For each attribute, we have a *constructor* that embeds the value into the union, and a *destructor* that projects it out of the union. In Haskell syntax, this reads:

$$\begin{aligned}
\mathbf{data}\ AttrValue\ =\ &Value\ \{unvalue\ ::\ Int\}\ |\\
&Locmin\ \{unLocmin\ ::\ Int\}\ |\\
&Gmin\ \{unGmin\ ::\ Int\}\ |\\
&Ntree\ \{unNtree\ ::\ Tree\ ProdName\ Attrs\}\ |\\
&Count\ \{unCount\ ::\ Int\}
\end{aligned}$$

Note the type of the attribute *ntree*: it is a tree whose *Fork* nodes are labelled with names of productions, and whose value nodes carry an attribution. Using the above constructors and destructors for *AttrValue*, we can now define the five attributes using the *mkAt* function:

$$
\begin{array}{lcl}
\mathit{value} & = & \mathit{mkAt}\,(\mathit{ValueId},\ \mathit{Value},\ \mathit{unvalue}) \\
\mathit{locmin} & = & \mathit{mkAt}\,(\mathit{LocminId},\ \mathit{Locmin},\ \mathit{unlocmin}) \\
\mathit{gmin} & = & \mathit{mkAt}\,(\mathit{GminId},\ \mathit{Gmin},\ \mathit{ungmin}) \\
\mathit{ntree} & = & \mathit{mkAt}\,(\mathit{NtreeId},\ \mathit{Ntree},\ \mathit{unntree}) \\
\mathit{count} & = & \mathit{mkAt}\,(\mathit{CountId},\ \mathit{Count},\ \mathit{uncount})
\end{array}
$$

We note once again that our use of embedding/projection pairs neatly hides the internal structure of an attribute, namely its name and its type. In an early version of this paper, we did not do so, and consequently we had to pass the triples of (name, constructor, destructor) around in many functions. That is rather clumsy, and it breaks the abstraction of an 'attribute' — we wish to hide the implementation detail as much as possible. The practical benefit is that the Haskell type system guarantees that each attribute can only be assigned values of the appropriate type.

To illustrate the above definitions, let us consider an example attribution from the *repmin* problem. An internal node might have the following inherited attribution:

$$(\mathit{embed}\ \mathit{gmin}\ 4\ \oplus\ \mathit{embed}\ \mathit{count}\ 5) =\ \{\,(\mathit{GminId}, \mathit{Gmin}\ 4),\ (\mathit{CountId}, \mathit{Count}\ 5)\,\}$$

Using the material presented so far we define the following tree construction functions for the *repmin* example:

$$
\begin{array}{ll}
\mathit{leaf}\ a & = \mathit{Fork}\ \mathit{Leaf}\ [\mathit{Val}\,(\mathit{embed}\ \mathit{value}\ a)] \\
\mathit{node}\ \mathit{ss} & = \mathit{Fork}\ \mathit{Node}\ \mathit{ss} \\
\mathit{root}\ t & = \mathit{Fork}\ \mathit{Root}\ [t]
\end{array}
$$

Note how the leaf data is stored in the *value* attribute. Now, one can construct an example tree thus:

$$
\begin{array}{lcl}
\mathit{example} & :: & \mathit{Tree}\ \mathit{ProdName}\ \mathit{Attrs} \\
\mathit{example} & = & \mathit{root}\,(\mathit{node}\,[\,\mathit{node}\,[\mathit{leaf}\ 3, \mathit{leaf}\ 1], \\
 & & \qquad\qquad\quad \mathit{node}\,[\mathit{leaf}\ 4, \mathit{node}\,[\mathit{leaf}\ 1,\ \mathit{leaf}\ 2]]])
\end{array}
$$

It is worthwhile to reflect for a moment which parts of the semantics so far are dependent on the particular example at hand. The types of production names, attribute names and attribute values are specific. To get the semantics for other examples, new definitions have to be substituted.

4. Composing attribute grammars

What are the building blocks of an attribute grammar? In their purest form, they are composed only of productions, and for each production, all attributes are defined simultaneously. Many attribute grammar systems also allow one to group definitions by *aspect*, where a number of related attributes are defined together, but not necessarily all attributes for each production. We have seen several examples of aspects in our running example. These aspects are however special in the sense that each defines only a single attribute. Aspects can be *woven* together to form a pure attribute grammar. Naturally one could see this as a syntactic operation, performed by a preprocessor, that simply collects all attribute definitions for each production from all aspects. We believe that it is beneficial to give a semantics to aspects, so that they are first-class values that can be returned as the result of

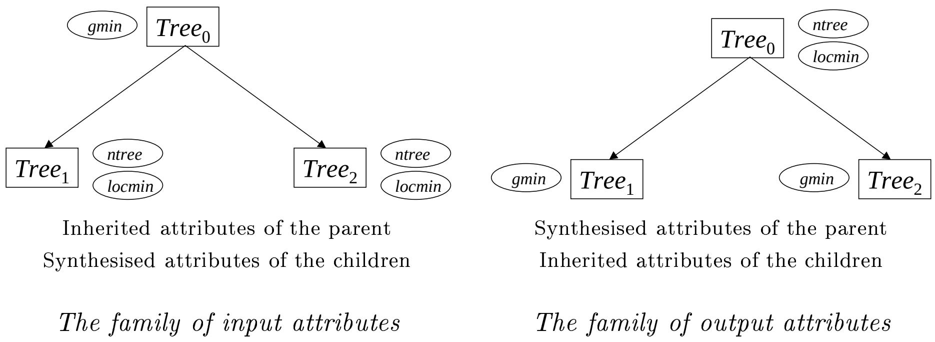


Figure 2: Input and Output Attribute Families

functions, and passed as arguments. This section describes such a semantics. Experienced functional programmers may wish to glance ahead at Figure 3, which gives a summary of the types introduced in this section.

The first step towards defining a semantics is to use Haskell functions to model attribute definitions. Let us recall the traditional form of attribute definitions. The following code was used in our *repmin* example:

$$\begin{array}{lcl} \mathit{Node} & : & \mathit{Tree}_0 \rightarrow \mathit{Tree}_1\ \ \mathit{Tree}_2 \\ \mathit{Tree}_0.\mathit{ntree} & = & \mathit{Node}\ \ \mathit{Tree}_1.\mathit{ntree}\ \ \mathit{Tree}_2.\mathit{ntree} \\ \mathit{Tree}_0.\mathit{locmin} & = & \mathit{min}\ \ \mathit{Tree}_1.\mathit{locmin}\ \ \mathit{Tree}_2.\mathit{locmin} \\ \mathit{Tree}_1.\mathit{gmin} & = & \mathit{Tree}_0.\mathit{gmin} \\ \mathit{Tree}_2.\mathit{gmin} & = & \mathit{Tree}_0.\mathit{gmin} \end{array}$$

There are two ways of viewing this code. The first view is that there are four semantic functions that each define a single attribute. The second view is that there is one semantic rule that defines a set of attributes (containing four elements). We prefer the second view because it will allow us to easily define the *composition* of two semantic rules. So we shall define semantic rules to be functions from the set of *input attributes* to a subset of the *output attributes*. Input attributes are attributes such as $\mathit{Tree}_1.\mathit{locmin}$ above. They are the attributes that we are allowed to read from. The output attributes are the attributes that we are defining. This is illustrated in Figure 2.

It is useful to have a name for the sets of attributes used above. We shall call them families. The first subsection below formally defines families, and operators for composing them. Next, we turn to the definition of rules. A rule is a mapping between families, namely from the input attributes of a production to some of its output attributes. Once rules are defined, it is possible to formalise the notion of an aspect. An aspect assigns rules to a number of production names. The remainder of this section shows how aspects can be built and combined in various ways, including those that were illustrated in the introduction.

4.1. Families

Families are used to model sets of input attributes or sets of output attributes. Therefore, a *family* consists of an attribution for a parent node, and an attribution for each of its children. That is, it is a pair that consists of an attribution, and a list of attributions:

$$\textbf{type}\ \mathit{Fam}\ =\ (\mathit{Attrs}, [\mathit{Attrs}])$$

For concreteness, let us consider the family of input attributes associated with the *Node* production in *repmin*. These attributes are depicted in Figure 2. Below is an instantiation of the family, in which values have been assigned to all the attributes:

$$\begin{array}{l} (\{\,(\mathit{GminId},\ \mathit{Gmin}\ 2)\,\},\ [\,\{\,(\mathit{NtreeId},\ \mathit{Ntree}\ (\mathit{Fork}\ \ldots)),\ (\mathit{LocminId},\ \mathit{Locmin}\ 3)\,\}, \\ \qquad\qquad\qquad\qquad\quad\ \{\,(\mathit{NtreeId},\ \mathit{Ntree}\ (\mathit{Val}\ \ldots)),\ (\mathit{LocminId},\ \mathit{Locmin}\ 5)\,\}\,]) \end{array}$$

Following our design principle that each new concept should have a corresponding join operation, we now define the empty family, and joining of families. Not surprisingly, we can do so by lifting the earlier definitions on attributions in an appropriate way.

The simplest family of all has empty attributions, and an infinite number of children:

$$\begin{array}{l} \emptyset\ ::\ \mathit{Fam} \\ \emptyset\ =\ (\emptyset,\ \mathit{repeat}\ \emptyset) \end{array}$$

The function $\mathit{repeat} :: \alpha \rightarrow [\alpha]$ generates an infinite list of copies of its argument. Again we are taking a minor notational liberty here, by overloading the notation for the empty map to also apply to the empty family. In Haskell, the two would have to be separated, or overloaded via a so-called *type class*. We shall use the same illicit overloading in the definition of the join operator on families.

Two families can be *joined* by joining their parents, and joining their children position-wise. Informally, we have:

$$\begin{array}{l} (s, [cs_0, cs_1, \ldots])\ \oplus\ (t, [ct_0, ct_1, \ldots]) \\ = \\ (s\ \oplus\ t,\ [cs_0\ \oplus\ ct_0,\ cs_1\ \oplus\ ct_1,\ \ldots]) \end{array}$$

In Haskell, this is achieved via the function call $\mathit{zipWith}\ f\ xs\ ys$ which applies the function f to corresponding elements of the lists xs and ys. Furthermore, the length of the result of $\mathit{zipWith}\ f$ is the minimum of the length of its arguments. We have:

$$\begin{array}{lcl} (\oplus) & :: & \mathit{Fam} \rightarrow \mathit{Fam} \rightarrow \mathit{Fam} \\ (s, cs)\ \oplus\ (t, ct) & = & (s\ \oplus\ t,\ \mathit{zipWith}\ (\oplus)\ cs\ ct) \end{array}$$

Again the operator $(\oplus)$ is associative, and has unit $\emptyset$.

4.2. Rules

As we discussed earlier, a rule is a mapping from the *input attributes* of a production to some of its *output attributes*. Since both input and output attributes can be modelled as families, we define the type of attribute definition rules as:

$$\mathbf{type}\ \mathit{Rule}\ =\ \mathit{Fam}\ \rightarrow\ \mathit{Fam}$$

To illustrate, consider the rule that defines the *locmin* attribute in the *Node* production of *repmin*. In the traditional notation we employed in the introduction, it reads:

$$\mathit{Tree}_0.\mathit{locmin}\ =\ \mathit{min}\ \mathit{Tree}_1.\mathit{locmin}\ \mathit{Tree}_2.\mathit{locmin}$$

Encoded as an element of the above type, it becomes the function:

$$\begin{array}{l} \lambda\,(\mathit{tree}_0, [\mathit{tree}_1, \mathit{tree}_2])\ \rightarrow\ (\mathit{embed}\ \mathit{locmin}\ (\mathit{min}\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_1)\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_2)), \\ \qquad\qquad\qquad\qquad\quad\ [\emptyset, \emptyset]) \end{array}$$

Note that in the resulting family, both children have empty attributions. If we instead encode two rules simultaneously, say both of

$$\begin{aligned} &\mathit{Tree}_0.\mathit{locmin} &&= \mathit{min}\ \mathit{Tree}_1.\mathit{locmin}\ \mathit{Tree}_2.\mathit{locmin} \\ &\mathit{Tree}_2.\mathit{gmin} &&= \mathit{Tree}_0.\mathit{gmin} \end{aligned}$$

we would have a non-empty attribution for the second child:

$$\begin{aligned} \lambda\,(\mathit{tree}_0, [\mathit{tree}_1, \mathit{tree}_2]) \rightarrow\ &(\mathit{embed}\ \mathit{locmin}\ (\mathit{min}\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_1)\ (\mathit{project}\ \mathit{locmin}\ \mathit{tree}_2)), \\ &\ [\emptyset, \mathit{embed}\ \mathit{gmin}\ (\mathit{project}\ \mathit{gmin}\ \mathit{tree}_0)]) \end{aligned}$$

Note that we have chosen suggestive identifiers in the argument family, but these are merely local names. In the rule itself, no knowledge of the nonterminals of the underlying context free grammar has been encoded. This has certain advantages, in particular that one can give rules that are independent of the precise form of the production that they will be associated to. The main disadvantage is that the notation can be a little hairy to use in practice: although we already named descendants in a production, those same names have to be repeated in each rule associated with the production.

Let us now consider some operations for manipulating rules. By lifting the corresponding operations on families, we get an empty rule (that does not define any attributes) and a join operator:

$$\begin{aligned} &\emptyset &&:: \mathit{Rule} \\ &\emptyset\, f &&= \emptyset \\[1em] &(\oplus) &&:: \mathit{Rule} \rightarrow \mathit{Rule} \rightarrow \mathit{Rule} \\ &(r_1 \oplus r_2)\, f &&= (r_1\, f) \oplus (r_2\, f) \end{aligned}$$

It is at this point that we can start introducing some shorthands for common vocabulary in attribute definitions. For example, here is an operator that generates a copy rule, which simply copies an inherited attribute from the parent to all the children:

$$\begin{aligned} &\mathit{copyRule} &&:: \mathit{At}\,\alpha \rightarrow \mathit{Rule} \\ &\mathit{copyRule}\,(e, p)\,(\mathit{inhp}, \mathit{syncs}) &&= (\emptyset,\ \mathit{repeat}\ (e\ (p\ \mathit{inhp}))) \end{aligned}$$

Another common design pattern is to collect synthesised attributes off all the children. Here we need a function *collect* that maps a list of attribute values to a single value:

$$\begin{aligned} &\mathit{collectRule} &&:: \mathit{At}\,\alpha \rightarrow ([\alpha] \rightarrow \alpha) \rightarrow \mathit{Rule} \\ &\mathit{collectRule}\,(e, p)\ \mathit{collect}\,(\mathit{inhp}, \mathit{syncs}) &&= (e\,(\mathit{collect}\,(\mathit{map}\ p\ \mathit{syncs})),\ \mathit{repeat}\ \emptyset) \end{aligned}$$

The function $\mathit{map}\ p\ \mathit{syncs}$ applies the projection p to each of the synthesised attributions of the children. We are assuming, therefore, that each of the children does indeed possess the attribute in question.

Finally, here is a formulation of the notion of *chain rule*. It takes an attribute, and it returns a rule that threads the attribute from left to right, before defining the synthesised occurrence at the parent:

$$\begin{aligned} &\mathit{chainRule} &&:: \mathit{At}\,\alpha \rightarrow \mathit{Rule} \\ &\mathit{chainRule}\,(e, p)\,(\mathit{inhp}, \mathit{syncs}) &&= (\mathit{last}\ \mathit{output},\ \mathit{init}\ \mathit{output}) \\ & && \quad \textbf{where}\ \mathit{output} = \mathit{map}\ (e \cdot p)\ \mathit{input} \\ & && \qquad\qquad \mathit{input} = \mathit{inhp} : \mathit{syncs} \end{aligned}$$

First we take all the input attributions as a list, by prefixing the synthesised attributions of the children by the inherited attribution of the parent: this yields the list named *input*. We then apply the composite function $e \cdot p$ to each of the elements of *input*: this yields the list *output*. Finally we return the last element of *output* as the synthesised attribution of the parent, and all but the last element as the inherited attributions of the children. Note that this definition is completely independent of how many children there are. In particular, if there was no child at all ($syncs = [\,]$), the attribute is copied unchanged from the inherited attribution to the synthesised attribution.

Undoubtedly some readers will prefer subtly different definitions of these common patterns: hopefully they will be encouraged by the simplicity of our choices to try and formulate their own in the present framework.

4.3. Aspects

Often we wish to group together rules that define related attributes, across multiple productions. For instance, we might wish to group together all attribute definitions that relate to type checking, or to a particular data flow analysis. Such a group of related rules is called an *aspect*, following terminology in object-oriented design [17]. Formally, we define:

$$\textbf{type}\ \mathit{Aspect} = \mathit{ProdName} \mapsto \mathit{Rule}$$

In words, an aspect maps production names to rules. It is not necessary for an aspect to map every production name in a grammar to a rule: it can be a partial function. We have already discussed several aspects in the introduction to the *repmin* problem, and we shall see shortly how these can be expressed as elements of the above type.

Again we can lift the empty and join operators to operate on aspects, and again we shall write $\emptyset$ for the empty aspect, and $\oplus$ for join. The empty aspect is simply the empty finite map. The join operator is a little more subtle than before, due to the fact that aspects may be partial:

$$\begin{aligned}(f \oplus g)\, x &= f\, x,\ \textbf{if}\ x \notin \mathit{domain}\ g\\ &= g\, x,\ \textbf{if}\ x \notin \mathit{domain}\ f\\ &= f\, x \oplus g\, x,\ \textbf{otherwise}\end{aligned}$$

An aspect is often defined by giving a default rule for most productions (for instance a copy rule for an inherited attribute), supplemented by specific rules for only a handful of the productions. To build the default aspect, we have the operator:

$$\begin{aligned}&\mathit{defaultAspect} &&:: \mathit{Rule} \to [\mathit{ProdName}] \to \mathit{Aspect}\\ &\mathit{defaultAspect}\ r\ \mathit{ls} &&= \{\,(l, r) \mid l \leftarrow \mathit{ls}\}\end{aligned}$$

It maps each production name l (of type *ProdName*) to the same rule r. Strictly speaking the above is not valid Haskell, as we have made up the set comprehension notation for finite maps, for increased readability.

In practice it is somewhat inconvenient to specify rules and aspects directly. Therefore, to make the interface of this library for composing attribute grammars a little less forbidding, we introduce the notion of *attribute aspects*. An attribute aspect is like an ordinary aspect, but it defines values only for a single attribute of type α. Formally, it is a finite map from production names to functions of type $\mathit{Fam} \to \alpha$ (ordinary aspects have a result of type $\mathit{Rule} = \mathit{Fam} \to \mathit{Fam}$):

$$\textbf{type}\ \mathit{AtAspect}\ \alpha = \mathit{ProdName} \mapsto (\mathit{Fam} \to \alpha)$$

An attribute aspect can be converted into a proper aspect by applying the function *inh* (for inherited attributes) or *synth* (for synthesised attributes). In the case of an inherited attribute, the attribute aspect defines a list of values, one for each descendant:

$$\begin{array}{lcl}
inh & :: & At\ \alpha \rightarrow AtAspect\ [\alpha] \rightarrow Aspect \\
inh\ a\ atAspect\ pname\ f & = & (\emptyset,\ map\ (embed\ a)\ (atAspect\ pname\ f)) \\
 & & \\
synth & :: & At\ \alpha \rightarrow AtAspect\ \alpha \rightarrow Aspect \\
synth\ a\ atAspect\ pname\ f & = & (embed\ a\ (atAspect\ pname\ f), repeat\ \emptyset)
\end{array}$$

(In these definitions, we are again taking the liberty of mixing the notation of ordinary functions with that for finite maps.) Using the above operators, we can define a primitive that defines an inherited attribute that is mostly copied, except in a few productions that are specified as an attribute aspect:

$$\begin{array}{lcl}
inherit & :: & At\ \alpha \rightarrow [ProdName] \rightarrow AtAspect\ [\alpha] \rightarrow Aspect \\
inherit\ a\ pnames\ atAspect & = & inh\ a\ atAspect\ \oplus\ defaultAspect\ (copyRule\ a)\ pnames
\end{array}$$

Astute readers will recognise this as desugared version of the *inherit* construct that was introduced earlier in this paper:

inherit ⟨attribute *a*⟩
copy at ⟨list of production names *pnames*⟩
define at ⟨attribute aspect *atAspect*⟩

The only difference is that the list of values was more conveniently specified in the introduction, by listing symbol occurrences in the relevant production. Of course such syntactic sugar is easily added by a simple preprocessor.

Similarly, one obtains the semantic counterpart for the *synthesise* construct in the introduction. There a synthesised attribute is collected in a specified list of productions, and defined elsewhere through an attribute aspect:

$$\begin{array}{lcl}
synthesise & :: & At\ \alpha \rightarrow ([\alpha] \rightarrow \alpha) \rightarrow [ProdName] \rightarrow AtAspect\ \alpha \\
 & & \rightarrow Aspect \\
synthesise\ a\ coll\ pnames\ atAspect & = & synth\ a\ atAspect \\
 & & \oplus \\
 & & defaultAspect\ (collectRule\ a\ coll)\ pnames
\end{array}$$

Finally, a chained attribute is defined by specifying two attribute aspects. The first gives the initialisations (which are inherited) and the second gives the update rules (which are synthesised):

$$\begin{array}{lcl}
chain & :: & At\ \alpha \rightarrow [ProdName] \rightarrow AtAspect\ [\alpha] \rightarrow AtAspect\ \alpha \\
 & & \rightarrow Aspect \\
chain\ a\ pnames\ atAspect_0\ atAspect_1 & = & inh\ a\ atAspect_0 \\
 & & \oplus \\
 & & synth\ a\ atAspect_1 \\
 & & \oplus \\
 & & defaultAspect\ (chainRule\ a)\ pnames
\end{array}$$

The definition of commonly occurring patterns such as *inherit*, *synthesise* and *chain* as first-class values was our original motivation for introducing the notion of aspects. For reasons of exposition, we have chosen the simplest possible definitions of these aspects, and not the most general ones. It should furthermore be noted that aspects do not necessarily define a single attribute, and so one can also define more complex patterns involving multiple attributes. Kastens and Waite [16] discuss techniques for encoding common attribution patterns in much greater detail. Many of our examples (in particular the *chain* and *synthesise* functions) were borrowed from their paper.

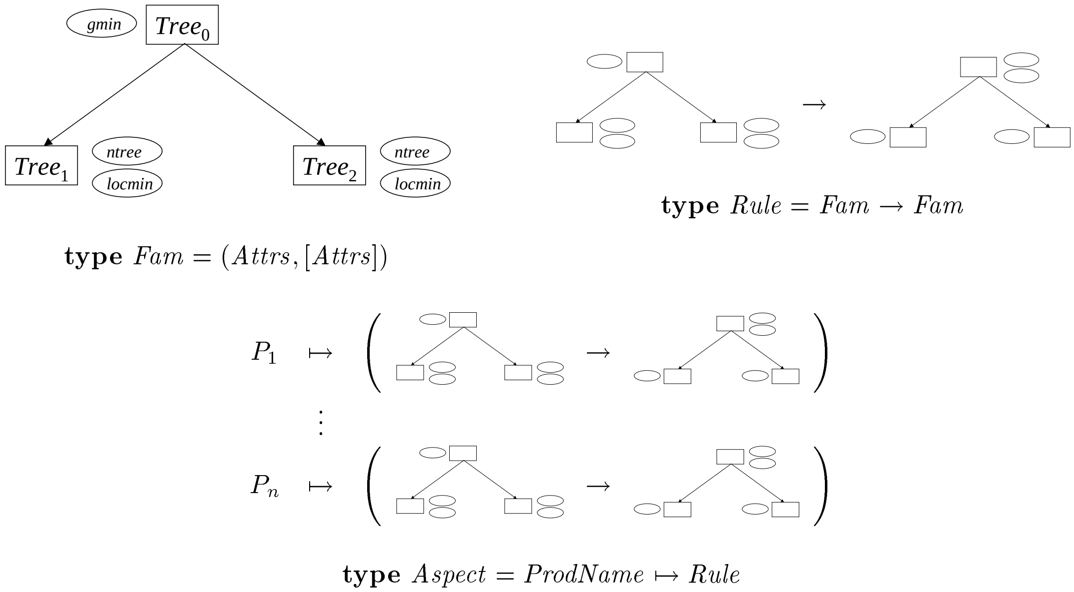


Figure 3: A summary of Section 4

5. The *repmin* example revisited

Using the definitions of the previous section, we can return to the example introduced at the beginning of this paper. This will illustrate the use of families, rules and aspects in practice. The reader may find it helpful to refer to Figure 3, which summarises the definitions of the previous section.

The first aspect is that of the global minimum. The global minimum is an inherited attribute that is broadcast to all nodes through a copy rule. It follows that we only have to define its value at the root. There it equals the local minimum of the immediate descendant:

$$\begin{array}{lcl} gmins & :: & Aspect \\ gmins & = & inherit\ gmin\ [Node] \\ & & \{\ (Root,\ \lambda\,(start, [tree])\ \to\ [project\ locmin\ tree]\ )\ \} \end{array}$$

By contrast, the local minimum is a synthesised attribute that is collected from all descendants using the function *minlist* that returns the minimum of a list of integers. For leaves, the local minimum is defined to be the value of the single child:

$$\begin{array}{lcl} locmins & :: & Aspect \\ locmins & = & synthesise\ locmin\ minlist\ [Node] \\ & & \{\ (Leaf,\ \lambda\,(leaf, [val])\ \to\ project\ value\ val)\ \} \end{array}$$

It remains to define the aspect that produces new trees. Recall that we had two versions of the example problem, which differed in the value that had to be substituted for leaves. To cater for that difference, we parameterise the aspect by the attribute that is the value to substitute at leaves. At the root, and at ordinary nodes, we collect new trees of the descendants using the *node* constructor. At the leaves, we substitute the argument attribute:

$$\begin{array}{lcl} ntrees & :: & At\ Int\ \to\ Aspect \\ ntrees\ a & = & synthesise\ ntree\ node\ [Node] \\ & & \{\ (Leaf,\ \lambda\,(leaf, [val])\ \to\ leaf\ (project\ a\ leaf)), \\ & & \ \ (Root,\ \lambda\,(start, [tree])\ \to\ project\ ntree\ tree)\ \} \end{array}$$

The first and simplest version of the example can now be assembled into a compiler, by passing the global minimum attribute to the *ntrees* aspect: (The Haskell function *compiler* that we use here is explained in the next section.)

$$\begin{array}{lll} repmin_0 & :: & Tree\ ProdName\ Attrs \ \rightarrow \ Tree\ ProdName\ Attrs \\ repmin_0 & = & compiler\ [gmins, locmins, ntrees\ gmin]\ ntree \end{array}$$

The more complicated version of the example required that we replace each leaf L by the number of times the global minimum occurs to the left of L. To program that variant, we first introduce an aspect for the counter:

$$\begin{array}{lll} counts & :: & Aspect \\ counts & = & chain\ count\ [Node] \\ & & \{\ (Root,\ \lambda\,(start, [tree])\ \rightarrow\ [0])\ \} \\ & & \{\ (Leaf,\ \lambda\,(leaf, [val])\ \rightarrow\ \mathbf{if}\ (project\ value\ val)\ ==\ (project\ gmin\ leaf)\ \mathbf{then}\ ) \\ & & \qquad\qquad\qquad\qquad\qquad\quad project\ count\ leaf\ +\ 1 \\ & & \qquad\qquad\qquad\qquad\qquad \mathbf{else} \\ & & \qquad\qquad\qquad\qquad\qquad\quad project\ count\ leaf\ \} \end{array}$$

The new compiler is similar to the old one, except that we now weave in the counter aspect, and we pass the *count* attribute to the *ntrees* aspect:

$$\begin{array}{lll} repmin_1 & :: & Tree\ ProdName\ Attrs \ \rightarrow \ Tree\ ProdName\ Attrs \\ repmin_1 & = & compiler\ [gmins, locmins, ntrees\ count, counts]\ ntree \end{array}$$

6. Mapping Aspects to Compilers

In this section we shall explain how *families*, *rules* and *aspects* can be mapped to an executable implementation. We shall use the well known method of encoding attribute grammars as lazy functional programs [14, 18]. We shall give a brief introduction to this encoding, but the reader will benefit from an acquaintance with the work of Johnsson [14] and Swierstra [18]. A more recent paper by Swierstra [25] approaches the problem from a wider perspective and gives some non-trivial examples.

6.1. The Encoding

The method of encoding attribute grammars as lazy functional programs is based on the following observation: the semantics of a tree can be modelled as a *function*. This function is parameterised by the inherited attributes of the root of the tree and computes the synthesised attributes of the root. In other words, it is a function of type $Attrs\ \rightarrow\ Attrs$. This observation is valid for the following reason: if we instantiate the inherited attributes of the root of the tree, then the attribution rules tell us how to fully decorate the tree. Therefore, the synthesised attributes of the root depend functionally on the inherited attributes. Figure 4 gives an illustration of this.

In this section we shall frequently be manipulating functions of type $Attrs \rightarrow Attrs$. These functions represent the semantics of a tree, so we shall define the following shorthand:

$$\mathbf{type}\ SemTree\ =\ Attrs\ \rightarrow\ Attrs$$

A production is a tree constructor: it takes a list of trees (the children) and constructs a new tree. We said above that each of the children is modelled by a function of type *SemTree*. Therefore, the semantics of the production can be modelled by a function with the following type:

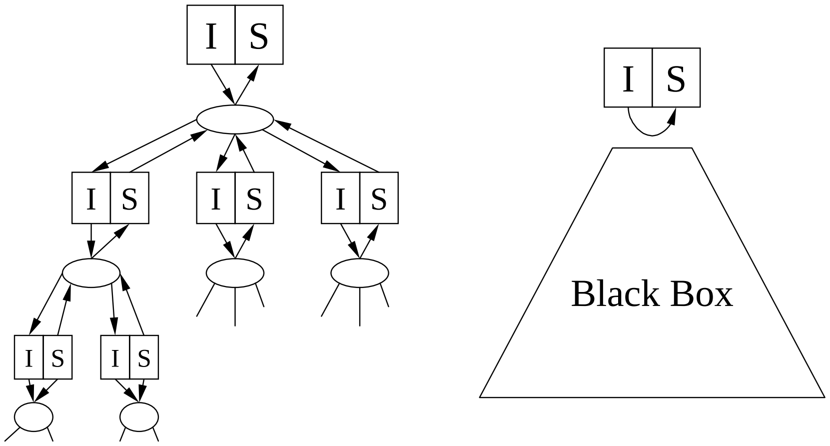


Figure 4: Modelling tree semantics as a function

$$\textbf{type}\ \mathit{SemProd}\ =\ [\mathit{SemTree}]\ \rightarrow\ \mathit{SemTree}$$

Once we have modelled the productions of an attribute grammar with functions of type *SemProd*, an evaluator for the grammar is constructed as follows: the evaluator recursively applies the semantic productions to the input tree. The result is a function of type *SemTree*, which represents the semantics of the tree. Below, we shall explain how rules can be mapped to semantic productions. Then we shall explain how aspects can be mapped to evaluators.

6.2. Mapping Rules to Semantic Productions

The conversion of rules to semantic productions is performed by the operation *knit*. Given a rule r and the semantics of the children *fs*, it should map the inherited attribution of the root to its synthesised attribution. To obtain the synthesised attributes of the root, as well as the inherited attributes of the children, we can simply apply the rule r. It remains to compute the synthesised attributes of the children: this we do by applying, for each child, the semantics to the inherited attributes. In sum, the definition of *knit* reads:

$$\begin{array}{lcl}
\mathit{knit} & :: & \mathit{Rule}\ \rightarrow\ \mathit{SemProd} \\
\mathit{knit}\ r\ \mathit{fs}\ \mathit{inhroot} & = & \mathit{synroot} \\
 & & \quad \textbf{where}\ (\mathit{synroot},\ \mathit{inhcs})\ = r\ (\mathit{inhroot},\ \mathit{syncs}) \\
 & & \qquad\qquad \mathit{syncs}\qquad\qquad\ \ = \mathit{applyList}\ \mathit{fs}\ \mathit{inhcs}
\end{array}$$

Note the cyclic definition of *synRoot*. Here we are relying on the lazy semantics of Haskell, so a similar definition would not work directly in a strict language such as ML. The function *applyList* takes a list of functions and applies them pointwise to a list of values. Its definition is as follows:[1]

$$\begin{array}{lcl}
\mathit{applyList}\ []\ \mathit{xs} & = & [] \\
\mathit{applyList}\ (f : \mathit{fs})\ \sim\!(x : \mathit{xs}) & = & f\ x\ :\ \mathit{applyList}\ \mathit{fs}\ \mathit{xs}
\end{array}$$

[1] *(For Haskell connoisseurs)* This definition contains a strictness annotation, which makes the function strict only in its first argument. This slightly simplifies the use of *knit* in practice, as it relieves users of the duty to be careful about strictness in defining attribution rules.

The use of laziness is crucial to the success of this implementation. Without it, we would need to analyse each rule and determine an evaluation order for the attributes. This would prevent us from defining a single *knit* function that can be applied to any rule. Using laziness, the evaluation order is determined at runtime [24].

6.3. Mapping Aspects to Evaluators

We define an *attribute grammar* to be a finite map from production names to production semantics:

$$\mathbf{type}\ AG = ProdName \mapsto SemProd$$

To convert an aspect to an attribute grammar, all that needs to be done is to knit each rule in its range. This is achieved by composing the aspect with the knit function:

$$\begin{array}{lcl} knitAspect & :: & Aspect \rightarrow AG \\ knitAspect\ as & = & knit \cdot as \end{array}$$

The evaluator is a function that recursively applies the attribute grammar to the input tree.

Attribute grammars define translators, which take a tree and an inherited attribution, and which produce a synthesised attribution. To translate a *Fork* node, we translate its descendants, and apply the semantics of the relevant production. To translate a *Val* node, we return its attribution:

$$\begin{array}{lcl} trans & :: & AG \rightarrow Tree\ ProdName\ Attrs \rightarrow Attrs \rightarrow Attrs \\ trans\ ag\ (Fork\ l\ ts)\ inh & = & ag\ l\ (map\ (trans\ ag)\ ts)\ inh \\ trans\ ag\ (Val\ a)\ inh & = & a \end{array}$$

Naturally we are usually interested only in the value of a single attribute; furthermore the compiler is often specified as a set of aspects. That common vocabulary is captured by the definition:

$$\begin{array}{lcl} compiler & :: & [Aspect] \rightarrow At\ \alpha \rightarrow Tree\ ProdName\ Attrs \rightarrow \alpha \\ compiler\ ass\ a\ t & = & project\ a\ (trans\ (knitAspect\ as)\ t\ \emptyset) \\ & & \mathbf{where}\ as = foldr\ (\oplus)\ \emptyset\ ass \end{array}$$

In words, we take a list of aspects ass, an attribute a, and a tree t. The aim is to produce the synthesised value of attribute a at the root of t. To that end, we first join all the aspects in $ass = [as_0, as_1, \ldots, as_{k-1}]$ to obtain a single aspect $as = as_0 \oplus (as_1 \oplus \ldots (as_{k-1} \oplus \emptyset))$. We then apply the corresponding translator to the tree t, giving it the empty attribution to start with. That produces the synthesised attribution of of the root; projecting on a gives the desired result.

It is sometimes handy to decorate the tree with all its attributions, both inherited and synthesised. Doing so is in fact no more difficult than the above compiler. Afficionadoes of the traditional encoding of attribute grammars in the functional paradigm (which foregoes the notion of an aspect) may wish to contemplate whether this operation can be written with the same efficiency as the one below:

$$\begin{array}{lcl} scan & :: & Aspect \rightarrow Tree\ ProdName\ Attrs \rightarrow Attrs \\ & & \quad \rightarrow Tree\ (ProdName, Attrs, Attrs)\ Attrs \\ scan\ as\ (Fork\ l\ ts)\ i & = & Fork\ (l, i, s)\ ts' \\ & & \quad \mathbf{where}\ (s, ics) = as\ l\ (i, map\ syn\ ts') \\ & & \quad\quad ts' = applyList\ (map\ (scan\ as)\ ts)\ ics \\ & & \quad\quad syn\ (Fork\ (l, inh, s)\ ts) = s \\ scan\ as\ (Val\ a)\ i & = & Val\ a \end{array}$$

7. Conclusion

We have presented a semantic view of attribute grammars, embedded as first-class values in the lazy functional programming language Haskell. Naturally we regard it as a benefit that our definitions are executable, but perhaps the more important contribution is the compositional semantics that we have given to the attribute grammar paradigm. It is hoped that this compositional semantics will yield further insight into making attribute grammars more flexible, encouraging the reuse of existing code where possible.

The utility of the semantics as an executable prototype is severely marred by the absence of static checks such as closure (each attribute that is used is also defined), and the circularity check (definitions do not depend on each other in a cyclic way). It is not difficult to add these checks, however, namely by providing an abstract interpretation of attribute values, and of the semantic functions. One can use this approach to compute the dependencies for each production separately, or even to generate the text of the composed attribute grammar, which could then be presented to a traditional attribute evaluator. Full details can be found in the literate Haskell program that accompanies this paper [7].

In earlier work we presented some of the same ideas via an encoding in a Rémy-style record calculus [8]. That encoding has the advantage that one can check for closure of the definitions through type inference. We found, however, that the approach was too restrictive, and made the definition of a number of important operations (such as a combinator for introducing chained attributes) exceedingly cumbersome. It is conceivable, however, that a more appropriate type system can be found, which offers the same guarantees (in particular that each attribute is defined precisely once), without the restrictions. We are however pessimistic that such a type system will allow full type inference, and that the types will be of manageable size. Very recently, Azero and Swierstra have succeeded in simplifying our original approach through the use of novel mechanisms for resolving overloading in Haskell — but the basic drawbacks of the approach remain. While preparing the present paper, we learned that the idea to model the semantics of attribute grammars through record calculus is not new: it was first suggested by Gondow and Katayama in the Japanese literature [10].

The examples of aspects given in this paper do not demonstrate the full potential of *production names* being first-class values. Every production that an aspect annotates is explicitly listed. For example, the *locmins* aspect individually lists the *Node* and *Leaf* productions. In larger grammars it would often be useful to work with sets of productions. For example, we could compute the set of productions that might appear on a path from nonterminal X to nonterminal Y. We could then annotate every production in that set with a default computation. Production names are first-class values, so we can easily define functions that manipulate them in this way.

Acknowledgements

We have much benefited from discussions with Pablo Azero on the topic of this paper. Tony Hoare's comments on an earlier paper [8] inspired some of the improvements presented here. We would also like to thank Atze Dijkstra, Jeremy Gibbons and João Saraiva for their helpful comments on the paper. We are grateful to the Programming Tools Group at Oxford for many enjoyable research meetings where these ideas took shape. Kevin Backhouse is supported by a studentship from Microsoft Research.

Bibliography

[1] S. Adams. *Modular Attribute Grammars for Programming Language Prototyping.* Ph.D. thesis, University of Southampton, 1991.

[2] A. Augusteijn. *Functional programming, program transformations and compiler construction.* PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1993. See also: `http://www.research.philips.com/generalinfo/special/elegant/elegant.html`.

[3] R. S Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[4] R. S Bird. *Introduction to Functional Programming in Haskell.* International Series in Computer Science. Prentice Hall, 1998.

[5] L. Correnson, E. Duris, and D. Parigot. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 353–369. Springer Verlag, 1999.

[6] L. Correnson, E. Duris, and D. Parigot. Equational semantics. In A. Cortesi and G. Filé, editors, *Symposium on Static Analysis SAS '99*, volume 1694 of *Lecture Notes in Computer Science*, pages 264–283. Springer Verlag, 1999.

[7] O. De Moor, K. Backhouse, and Swierstra S. D. First-class attribute grammars: Haskell code. Available from: `http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/waga.zip`, 2000.

[8] O. De Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented compilers. In *First International Symposium on Generative and Component-based Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[9] C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*, volume 1490 of *Lecture Notes in Computer Science*, pages 284–299, 1998.

[10] K. Gondow and T. Katayama. Attribute grammars as record calculus. Technical report 93TR-0047, Tokyo Institute of Technology, 1993.

[11] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, 1992.

[12] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation — an algebra of music. *Journal of Functional Programming*, 6:465–484, 1996.

[13] R. J. M. Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, volume 1995 of *Lecture Notes in Computer Science*, pages 53–96, 1995.

[14] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.

[15] M. Jourdan, D. Parigot, Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Languages Design and Implementation*, pages 209–222, 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[16] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[17] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996.

[18] M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz`.

[19] J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementations. *ACM Computing Surveys*, 27(2):226–255, 1995.

[20] M. Pennings. *Generating incremental evaluators.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1994. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/`.

[21] C. Ramming (editor). *Proceedings of the Usenix conference on Domain-Specific Languages '97.* USENIX, 1997.

[22] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[23] J. Saraiva. *Purely functional implementation of attribute grammars.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1999. Available from: `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[24] J. Saraiva and S. D. Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction - ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[25] S. D. Swierstra, P. Azero Alcocer, and P. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, editor, *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer-Verlag, 1998.

[26] H. Vogt. *Higher order attribute grammars.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1989.