Merging Idiomatic Haskell with Attribute Grammars

Arie Middelkoop[1], Jeroen Bransen[2], Atze Dijkstra[2], and S. Doaitse Swierstra[2]

[1] `amiddelk@gmail.com`
[2] Utrecht University, `{J.Bransen,atze,doaitse}@uu.nl`

Abstract. Attribute grammars with embedded Haskell code form an expressive domain specific language for tree traversals, and thereby an excellent tool for compiler construction. The Utrecht Haskell Compiler has been fully implemented with attribute grammars in a modular and aspect-oriented way of programming. However, the integration of attribute grammars with some prominent Haskell features poses challenges: conventional attribute grammars cannot be written for all Haskell data types, lazy evaluation may lead to space leaks, and the embedded code may not be monadic. In this paper we investigate these challenges, and present solutions using some general extensions to attribute grammars.

Keywords: Attribute grammars, Composition, Monads

1 Introduction

In functional programming, attribute grammars (AGs) can be seen as a declarative and compositional domain specific language for tree traversals, in particular those that can be written as a fold or catamorphism (Meijer et al., 1991). Because of the correspondence between grammars and algebraic data types, an attribute grammar describes such a traversal as a collection of rules between attributes of connected nodes of the tree, leaving the fact that we describe a catamorphism implicit. Attribute grammars are therefore well suited for the implementation of compilers.

Haskell is known to be an excellent implementation language for attribute grammars, since lazy evaluation provides a natural infrastructure for evaluating attribute grammars and advanced type system features even allows them to be expressed as an embedded domain specific language (Viera et al., 2009, 2011). In addition, rules can be given as pure embedded Haskell code (Swierstra and Alcocer, 1998) thus adding the expressiveness of Haskell to attribute grammars.

Our experiences with attribute grammars in the large project of the Utrecht Haskell Compiler (UHC) (Dijkstra et al., 2009) confirm that attribute can be used to implement a compiler in an attractive way and that Haskell is an excellent host language. However, over the years we also ran into a number of obstacles:

– Lazy evaluation is a double-edged sword. The translation of attribute grammars to Haskell results in so-called circular Haskell code which is difficult to

optimize. This code easily exhibits space leaks which are troublesome when dealing with large trees. This problem can be remedied by producing acyclic Haskell code (Bransen et al., 2012), imposing however (mild) restrictions on the grammar.
- Haskell data types, which may be parametrized over the types of their fields, are more expressive than context-free grammars. We can for example not define a context-free grammar for a data type with α as type parameter where a field of type α is represented as a nonterminal. Consequently, these fields cannot be traversed.
- The order of evaluation is completely left implicit in an attribute grammar, hence it is unclear how to integrate monadic attribute definitions, as the order of evaluation may be relevant for the result.

There is a need to solve these obstacles: lazy evaluation, polymorphism, and monads are prominent Haskell features that would be useful in combination with attribute grammars. As potential use-case, consider layout combinators for a window manager expressed by an attribute grammar that may tile windows either horizontally or vertically. This problem is closely related to pretty printing (Swierstra and Chitil, 2009). A solution for such a problem features monadic operations to query window properties such as hints, and may be parametrized over the type of state used by user hooks. We need solutions for integrating such features with attribute grammars, because these are not only obstacles for the application of attribute grammars, but also an obstacle for their adoption by Haskell programmers, who expect to be able to use such features.

In this paper we make two kinds of contributions: we show how to integrate these Haskell features with attribute grammars, and we present a novel attribute grammar extension that forms an essential ingredients. Specifically, we give a short introduction to attribute grammars (Section 2), show the translation of attribute grammars to cyclic and acyclic code (Section 3) and explain some common attribute grammar extensions (Section 4). We present the following novel attribute grammar extensions:

- Section 5 deals with data types and attributes that parametrized over types.
- Section 6 presents eager rules, which allows local assumptions to be made about the evaluation order of rules.
- Section 10 presents hooks into the attribute evaluator so that Haskell code can control the evaluator of a node (e.g. apply it repeatedly, or change attribute values), about which normally no assumptions can be made in an attribute grammar.

We show that with these extensions we can integrate the aforementioned Haskell features:

- Section 7 deals with the integration of type classes.
- Section 8 shows how to represent the fields of a data type as nonterminal that have a type that is a parameter of the data type.
- Section 9 introduces monadic rules which allows the combination of the composition mechanism of attribute grammars and monads to be combined.

```
data Root
   | Top      top : Tree
data Tree
   | Bin      l : Tree    r : Tree
   | Leaf     val :: Int
attr Tree  inh gmin :: Int
           syn lmin :: Int
           syn repl :: Tree
attr Root  syn repl :: Tree

sem Tree
   | Leaf   lhs.lmin   = @val
            lhs.repl   = Leaf @lhs.gmin
   | Bin    lhs.lmin   = @l.lmin `min` @r.lmin
            lhs.repl   = Bin @l.repl @r.repl
            l.gmin     = @lhs.gmin
            r.gmin     = @lhs.gmin
sem Root
   | Top    top.gmin   = @top.lmin
            lhs.repl   = @top.repl
```

Fig. 1: Common example of an attribute grammar: Repmin

This work is done in the context of the Utrecht University Attribute Grammar Compiler (UUAGC). The ideas are applicable for attribute grammars in general, but are explained in terms of the UUAGC syntax and implementation.

2 Attribute Grammar Tutorial: Repmin

Figure 1 shows Repmin (Bird, 1984), a typical example of an attribute grammar. Before we explain the example, we first discuss the three important syntactic elements.

The **data** declarations resemble data declarations from Haskell. They describe the context-free grammar: type constructors are nonterminals, value constructors are productions, and constructor fields are symbols in the right hand side of productions. An instance of such a type is a tree where each node was produced by a constructor. The fields of the constructors are explicitly named and come in two variations: terminal symbols (with a double colon) and nonterminal symbols (with a single colon).

Three categories of attributes can be declared for a nonterminal: inherited attributes of a child are defined by the parent and can be used by the child, and synthesized attributes are defined by the child and can be used by the parent. Finally, a chained attribute is a shorthand for a pair consisting of an inherited and a synthesized attribute with the same name.

Rules define how an attribute is to be computed in terms of other attributes. A **sem** block specifies a collection of rules per production. A rule is of the form $p = e$ where p is a pattern defining attributes, and e is a Haskell expression that may refer to an attribute or terminal by prefixing it with the @ symbol. We refer to an attribute using the notation $c.a$ where a is the name of the attribute, and c is either the name of a child, or the keyword **lhs** (left hand symbol) when referring to an attribute of the parent.

Attribute grammars thus offer a programming model where each node is decorated with attributes, and rules specify data dependencies between attributes. The idea of attribute evaluation is to compute values for attributes according to the data dependencies.

```
semTree (Leaf val) = semLeaf val
semTree (Bin l r)  = semBin (semTree l)
                            (semTree r)
semRoot (Top top) = semTop (semTree top)

semTop top =
  let (top_lmin, top_repl) = top top_gmin
      top_gmin = top_lmin
      lhs_repl  = top_repl
  in lhs_repl

semLeaf val = λlhs_gmin →
  let lhs_lmin = val
      lhs_repl  = Leaf lhs_gmin
  in (lhs_lmin, lhs_repl)

semBin l r = λlhs_gmin →
  let (l_lmin, l_repl)  = l lhs_gmin
      (r_lmin, r_repl) = r lhs_gmin
      lhs_lmin = l_lmin `min` r_lmin
      lhs_repl  = Bin l_repl r_repl
  in (lhs_lmin, lhs_repl)
```

Fig. 2: Translation of Repmin to lazy Haskell code.

Figure 1 shows how to compute a transformed tree as synthesized attribute *repl* where the value in each leaf is replaced by the minimal value in the original tree. For this purpose, the synthesized attribute *lmin* represents the local minimum of the tree. At the root of the tree, the inherited attribute *gmin* is defined as the global minimum by taking the local minimum associated with thee node at the *top* of the tree. This minimum value is passed down unchanged from each parent to its children.

Using attribute grammars is advantageous over writing Haskell functions by hand:

- The rules and attributes can be given in any order.
- The navigation over the tree structure is implicit in comparison to e.g. zippers.

These two advantages allow an attribute grammar to be composed out of several individual fragments, which facilitates separation of concerns.

3 Evaluation Algorithm

This sections mentions two translations to Haskell, which are both implemented in UUAGC. This section serves two goals: to provide the reader with a better understanding of the semantics (which we do not specify formally), and to prepare for later sections on the various grammar extensions and their implementation.

3.1 Translation to Lazy Haskell Code

In Swierstra and Alcocer (1998) a translation to lazy Haskell code is presented. The translation uses catamorphisms to map each node of the tree to its *evaluator*, which is a function that takes values of the node's inherited attributes as its arguments and computes the values of the node's synthesized attributes. For the Repmin example of the previous section, the evaluator is thus a function that takes *gmin* and produces *lmin* and *repl*.

Figure 2 shows the catamorphisms for *Tree* and *Root*, and the *semantic functions* that comprise the algebra, one for each production. A semantic function takes the evaluators for the children of a node as parameter and produces the evaluator for itself. The body of the function consists of calls to the evaluators of the children, and their inputs and outputs are tied together by straightforward translations of rules.

Note that *semTop* has a cyclic definition because the argument *top_gmin* to function *top* depends on the result *top_lmin* of *top*. This is not a problem because the runtime data dependencies are acyclic: *top_lmin* can be computed without needing *top_gmin*. Lazy evaluation provides the appropriate attribute scheduling. However, this requires that the function is not strict in its arguments, reducing the opportunity for optimizations.

3.2 Translation to Acyclic Haskell Code

The definition in Figure 1 is cyclic because the evaluator of the root needs to pass *gmin* to *top* before it gets *lmin*. However, what if the evaluator does not evaluate a tree in one (lazy) step, but as a sequence of smaller steps? If the evaluator of *top* can produce *lmin* in the first step, and only in the second step would need *gmin* to produce *repl*, then the definition is no longer cyclic!

It is possible to avoid the cyclic definition in Figure 2 when the grammar is *absolutely noncircular*. This can be verified using a static analysis given by Knuth (1968). In Bransen et al. (2012) we describe our approach for generating acyclic Haskell code based on attribute scheduling, using the algorithm from Kennedy and Warren (1976).

4 Common Attribute Grammar Extensions

This section covers a number of common language-independent attribute grammar extensions that we need in later sections.

4.1 Local Attributes

Local attributes are a simple but useful extension for sharing an intermediate result per node among rules. Rules may refer to an attribute of the form **loc**.x when it is defined by a rule. Local attributes resemble let-bindings, and examples of their use are given in later sections.

4.2 Copy Rules

Many rules copy values simply up and down the tree. These rules occur often in standard patterns and can be derived automatically based on the name equality of attributes. Such a derivable rule is called a *copy rule*. Copy rules may be omitted by the programmer, which has significant benefits in larger grammars where the majority of rules are copy rules.

The following are common patterns for which rules are automatically derivable:

Topdown When the inherited attribute $c.a$ of a child is not defined, but $\mathbf{lhs}.a$ exists, the copy rule $c.a = @\mathbf{lhs}.a$ is derived.

Bottom-up When the synthesized attribute $\mathbf{lhs}.a$ is not defined, either the rule $\mathbf{lhs}.a = @c_0.a$ ‘*mappend*‘ ... ‘*mappend*‘ $@c_k.a$ is derived for the subsequence c_0 ... c_k of the children which have a synthesized attribute a, or the rule $\mathbf{lhs}.a = \mathit{mempty}$ is derived when no such child exists. The functions *mappend* and *mempty* come from the monoid class and can be overridden with the **use** construct.

Chained When an inherited attribute $c.a$ of a child c is missing, the rule $c.a = @k.a$ is derived if $k.a$ exists, where k is either the nearest preceding child that has a as synthesized attribute or otherwise **lhs** if a is an inherited attribute. Also, when the synthesized attribute $\mathbf{lhs}.a$ is missing, $\mathbf{lhs}.a = @k.a$ is derived.

When an attribute has a **use** declaration, copy rules are generated according to the topdown and bottom-up pattern, and otherwise the chained pattern. Note that we do not omit copy rules in our examples for didactic reasons. However, we mention copy rules in later arguments, hence this explanation.

4.3 Higher-Order Children

A production declares the children that a node has at the start of attribute evaluation. An extension, higher-order attribute grammars (Vogt et al., 1989), allows additional children to be declared that become part of the tree during attribute evaluation. This is one of the most important and versatile attribute grammar extensions, and we will use it later in several examples.

A higher-order child $c : M$ (where c is the name of the child and M the nonterminal) is a tree defined by an attribute $\mathbf{inst}.c$ of its parent node. We say that the child c is *instantiated* with the value of attribute $\mathbf{inst}.c$. Such an attribute is also known as a higher-order attribute.

The declaration of the child is prefixed with **inst** to differentiate it from a conventional child declaration. So, to define some child $c : M$ as the result of expression e, the child must be declared and the attribute $\mathbf{inst}.c$ must be defined by some rule:

$$\begin{array}{lll} \mathbf{data}\ N & \mid P & \mathbf{inst}.c : M \\ \mathbf{sem}\ N & \mid P & \mathbf{inst}.c = e \end{array}$$

Equivalently, child declarations may also be given in the semantics block.

Furthermore, we must define the inherited attributes of c and may use the synthesized attributes of c. Its synthesized attributes additionally depend on the definition of $\mathbf{inst}.c$, because part of the tree must be known before synthesized attributes can be computed for it. Otherwise, a higher-order child is indistinguishable from a conventional child.

We define a nonterminal to represent a counter dispenser:

```
data Uniq | Next
attr Uniq   chn counter :: Int
            syn value   :: Int
sem Uniq
   | Next   lhs.value   = @lhs.counter
            lhs.counter = @lhs.counter + 1
```

Copy rules can be used to chain the counter through the tree, and an attribute @$u.value$ is obtained with:

```
inst.u : Uniq
inst.u = Next
```

Fig. 3: A unique number mechanism.

We define a nonterminal to represent a local attribute:

```
data Loc @α | Loc
attr  Loc chn value :: α
sem   Loc
   |  Loc lhs.value = @lhs.value
```

To desugar a local attribute x, we introduce:

```
inst.x : Loc
inst.x = Loc
```

and then replace each occurrence of **loc**.x with $x.value$.

Fig. 4: Local attributes as higher-order children.

The code generated for a production gets the evaluator for conventional children as parameter, but not for higher-order children. Instead, the evaluator is obtained by applying the catamorphism *semM* to the tree constructed for attribute **inst**.c.

Later sections make heavy use of higher-order children to expose Haskell functions as a flat tree to exploit the AG's composition mechanism. Since this pattern is important we give now two examples:

– Figure 3 shows a tree *Uniq* as abstraction for a dispenser of unique numbers. The tree itself is just a plain node *Next*. The required information is in the attributes.
– Figure 4 shows how to implement local attributes with higher-order children.

4.4 Proxy Nonterminals

We present a common pattern for adding a nonterminal to the grammar that serves as an alias for another nonterminal. Similar to type aliases in Haskell, this pattern can be used to statically distinguish certain occurrences of nonterminals.

A common pattern is to introduce a nonterminal that serves as an observable alias for another nonterminal, which we will call *proxy nonterminals*. A proxy nonterminal *Proxy* for N is a nonterminal *Proxy* with the same attributes declarations as N and is defined as **data** *Proxy* | $P\ n : N$ with its semantics trivially defined by copy rules. It thus has a single production P, containing one child n which is the nonterminal symbol N. We can thus substitute *Proxy* for

```
data List α @β                           attr List  inh f :: α → β  syn r :: List β
  | Nil                                  sem List
  | Cons   hd :: α   tl : List α           | Nil   lhs.r = Nil
attr List syn length :: Int                | Cons lhs.r = Cons (@lhs.f @hd) @tl.r
sem List
  | Nil   lhs.length = 0
  | Cons lhs.length = 1 + @tl.length
```

Fig. 5: Parametric polymorphism in an attribute grammar.

occurrences of N (in productions other than P) without changing the attribute computations.

Proxy nonterminals can be added to the grammar by changing the original description, e.g. the data declarations. This transformation is not transparent. Code that produces the tree (e.g. a parser) must also generate the proxy nodes at the appropriate places in the tree.

A transparent approach is possible, which we will use in Section 8, using an extension of higher-order attributes. Instead of *defining* a child with an attribute, we allow the *redefinition* of a child via an attribute that contains a function that *transforms* the evaluator of the child. The following example demonstrates a transformation of a child $n : N$ in production *Root* to a child $n : \mathit{Proxy}$:

```
data Root  | Root   n : N
sem Root   | Root   inst.n : Proxy
                    inst.n = λevalN → semP evalN
```

The *Root* production declares a child $n : N$. The **inst**.n attributes defines an attribute that is actually a function that takes the original evaluator of n as parameter *evalN* and transforms it so that it becomes an evaluator that fits nonterminal *Proxy*. In this case, we accomplish this by adding a P node on top of it. Note that the function *semP*, which is the part of the algebra that corresponds to production P, is exactly doing that. This transformation possible when the definition of **inst**.n does not depend on any of the synthesized attributes of n.

5 Parametric Polymorphism

The ability to abstract over types plays a major role in obtaining code reuse in strongly typed functional languages, and also in the form of generics in imperative languages. This section shows an attribute grammar extension for parametrizing nonterminals to abstract over the types of terminals, and for abstracting over the types of attributes.

Figure 5 shows that by parametrizing the nonterminal *List* with the type α of the terminal *hd*, it is possible to define the synthesized attribute *length* for lists containing elements of any type.

Similarly, by parametrizing the attributes over a type β, we can implement a functor: a transformation that maps each element @*hd* of the list to @**lhs**.f @*hd* where @**lhs**.f is an arbitrary function from a (the type the items in the list) to some arbitrary type β.

There is an essential difference between type variables α and β. Type variables declared with the prefix @ may appear only in the types of attributes, but may not appear in the types of terminals, and are not part of the generated data type. Thus, the data type is parametrized with α and the evaluator with α and β.

The implementation of this extension consists of printing the type variables at the appropriate places in type signatures.

6 Feature: Eager rules

The data dependencies between rules form a partial order, which suffices for attribute grammars because rules are encouraged to be pure. It may sometimes be useful to augment the data dependencies to locally prioritize certain rules over other rules in a production. This can for example be useful for debugging, efficiency, and other reasons that appear in later sections.

By taking the order of appearance of rules in the source files into account, it is possible to obtain a total order among rules. This approach impairs the composability of attribute grammars, but may still be useful for specifying an order among strongly correlated rules. Other canonical total orders are far from obvious. A total order would also leave little freedom to attribute scheduling, hence we are looking for a less ad-hoc mechanism.

A rule is *eager* when it is described with the notation p \$= e with a pattern p and expression e. The idea is to schedule these rules so that they are computed as soon as their dependencies are available, in contrast to conventional rules that are scheduled when an attribute that depends on it needs to be computed.

This is a challenging problem. To prioritize a rule, it is also necessary to prioritize the dependencies of that rule. This interacts globally with rules of other productions, and it is not clear which one has more priority. Such global consequences are undesirable when all that we want is a bit more local priority. We therefore propose to prioritize only the attributes that involve themselves only with eager rules, and leave the scheduling of the other attributes up to their original data dependencies.

An attribute a is *eager* when each rule r that depends on a is by itself eager, or r depends on another eager attribute. These are global properties of a grammar that are easily derived from the grammar with a static analysis similar to cycle analysis.

Given an inherited eager attribute a of some nonterminal N, we can determine the set of synthesized eager attributes that depend on a. We call these the *collaborators* of a. Furthermore, we can determine the set of non-eager inherited attributes that the collaborators depend on, which we call the *opposition* of a. The idea is to prioritize the computation of eager inherited attributes of a child as soon as their opposition has been computed.

```
data Root α | Top   root : Tree α
data Tree α | Bin   left : Tree α   right : Tree α
            | Leaf  val :: α
attr Tree   inh gmin :: α   syn lmin :: α   syn repl :: Tree α
```

Fig. 6: Repmin with type classes (see also Figure 1)

The scheduling algorithm of Section 3.2 starts from the demanded synthesized attributes of the parent for a visit to determine which rules and child visits to schedule. We change this algorithm to realize the above idea. For each eager inherited attribute $n.a$ of a child n, if the opposition of $n.a$ can be computed, we add the collaborators of $n.a$ that can be computed to the set of attributes to be computed. Recall that an attribute of a child can be computed if the inherited attributes of the parent it indirectly depends on have been computed.

Then, to deal with ordering the eager rules scheduled to a particular visit, we repeatedly take the unscheduled eager rules that do not depend on any other unscheduled eager rules, and schedule their non-eager dependencies and then the rules themselves in the order of appearance. See (Middelkoop, 2012, Section 3.5.2) for a detailed algorithm.

The approach is sound because it only adds additional scheduling constraints. The approach is also complete: if a schedule can be computed for a grammar than a schedule can also be computed when rules are changed into eager rules. The scheduling is also locally predictable: an eager rule is guaranteed to be scheduled before an independent non-eager rule that depends on a superset of the non-eager inherited attributes that the eager rule depends.

Conventional rules are scheduled based on the dependencies of the attributes that they define. Eager rules have the additional property that they also get scheduled if the inherited attributes that they depend on become available. We make use of this property in several later sections.

7 Integration: Type Classes

Haskell programmers make heavy use of type classes, and thus expect to combine them with attribute grammars. A typical use arises when some part of the tree or some of the attributes are abstracted over some type. When an overloaded function is applied to the value of such an attribute, a dictionary is required that provides the implementation of the overloaded function. The construction and passing of dictionaries is normally handled implicitly by the Haskell compiler, and the question arises how this integrates with attribute grammars.

Figure 6 shows a variation on Repmin of Figure 1 which works for trees containing comparable values of any type α. We omitted the rules as these are the same as the original definition. The attributes are polymorphic in the type α, and in the rules we are using *min* from the class *Ord*, so the generated code can only be used when the type α is in the *Ord* class and when the corresponding dictionary is brought in scope of the code that is generated from the attribute definition that uses *min*.

The way we generate code allows the Haskell compiler to handle type classes automatically if we do not generate type signatures. Note that type signatures are particularly important to aid error reporting, hence we are not willing to leave them out. Fortunately, the impact on type signatures is rather small, because only the types of the generated fold and algebra functions need to specify the used dictionaries in their body as class predicates, which can be manually specified by the programmer with a bit of additional syntax:

```
attr Ord α ⇒ Root Tree   inh gmin :: α   lmin :: α
```

This notation expresses that the *Ord* α class constraint is added to the catamorphisms and semantic functions generated for *Root* and *Tree*.

This construction is undesirable for several reasons:

- In a context where not all synthesized attributes are needed, the rule using the dictionary may not be scheduled, and the dictionary not needed, resulting in ambiguous overloading.
- It requires a language-specific extension, while there might be solutions more native to attribute grammars.

8 Integration: Abstraction over Nonterminals

Section 5 showed that data types may have fields that have a type that the data type takes as parameter, and that we treat these fields as terminals. But what about nonterminals? For example, when some meta information such as source locations occurs at many places in different types of trees, it is common to factor it out into a separate nonterminal:

```
data Info t | Label  tree  :: t          line :: Int
data Stmt   | If     guard : Info Expr   body : Info Stmt
data Expr   | App    fun   : Info Expr   arg  : Info Expr
```

We would like to change terminal *tree* into a nonterminal so that *Info* becomes polymorphic in the nonterminal *t* chosen for *tree*, but it is unclear how to deal with such a grammar: what are the attributes of *Info*? This likely depends on what attributes are defined on *t* (which is not known) and the *line* likely influences them or requires additional attributes. This issue becomes even more difficult when a production has multiple such children.

Saraiva and Swierstra (1999a) deal with nonterminals parametrized over nonterminals by specifying which attributes will be present. This is not a solution in this case because it *Stmt* and *Expr* may not have the same attributes. Instead, we propose a simpler approach: we virtualize the tree. The observation is that higher-up there must be a place where it is known which attributes to expect:

either because the instantiation of the type variables is known or because the attributes are independent of it. For example, we can assume that we know the attributes of a tree of type *Info Expr*.

For this concrete type, it is possible to derive some suitable representation that does not involve nonterminals as parameters, for example by specializing the original data definition to the known type arguments:

```
data InfoExpr  | Rep    expr : Expr    line :: Int
```

We can thus define the required attributes and rules on *InfoExpr* instead, provided that we transform a tree of type *Info Expr* to *InfoExpr*. We first introduce a proxy nonterminal for *InfoExpr*, which will take care of the conversion.

```
data ExprProxy  | Proxy   orig : Info Expr
sem Stmt        | If      inst.guard : ExprProxy
                          inst.guard = Proxy
```

For the conversion, we compute the representation from *Label*, passing down as additional information that *t* is a *Stmt* in this context, and using a higher-order child to make the representation part of the tree:

```
attr Info   inh eqExpr :: t ∼ Expr      syn repExpr :: InfoExpr
sem Info          | Label   lhs.repExpr  = case@lhs.eqExpr of
                                              Refl → Rep @tree @line
sem ExprProxy | Proxy   orig.eqExpr  = Refl
                        inst.rep     :  InfoExpr
                        inst.rep     = @orig.repExpr
```

Similarly, a representation for *Info Stmt* can be added, with corresponding attributes *eqStmt* and *repStmt* for *Info*. The *orig.eqExpr* attribute can only be defined in *ExprProxy* and vice versa for *orig.eqStmt*. Thus, by making these nonterminals start symbols of the grammar, these inherited attributes need only be defined for the appropriate proxies (Section 2).

The above approach for specializing the types of nonterminals can be automated with some preprocessing. On the other hand, this approach makes it also possible to use a more abstract representation (e.g. using sums of products (Magalhães et al., 2010)) to obtain generic code.

9 Integration: Monads

Monads are a typical abstraction that Haskell programmers use when implementing tree traversals. They are often considered as an alternative to attribute grammars. Indeed, Schrijvers and Oliveira (2011) show how to deal with stacks of reader, write and state monads to obtain a similar composability that comes naturally with attribute grammars.

However, attribute grammars and monads are different composition mechanisms but they are not mutually exclusive. In fac, they are different composition mechanisms that are fruitful to combine (Meijer and Jeuring, 1995). Section 9.1 shows the embedding of pure monadic computations that use reader, writer, state functionality as abstraction (e.g. the RWS monad), and Section 9.2 shows how we can represent the AG as a monad to incorporate impure operations.

9.1 Integration: Pure Monadic Code in Rules

When using an attribute grammar there seems no need to use reader, writer or state (RWS) monads, because attributes provide a more general facility when combined with copy rules (Section 4.2). However, as rules may contain arbitrary Haskell code, that code can involve (pure) monads, and this may certainly be appropriate when constructing complex values.

When the monad can be evaluated as a pure Haskell function, which is the case for RWS monads, monadic code is not different from conventional code, and can be used without a need for special attribute grammar facilities (otherwise, see Section 9.2). However, the use of monadic code gives rise to a particular pattern for which we can introduce an abstraction, which we discuss in the remainder of this section.

Example. The following grammar on lists of integers defines a synthesized attribute r. Given such a list L, the attribute r of L is also a list of integers but with consecutive elements and so that there are as many elements as the total sum of the elements of L. The grammar implements this behavior by concatenating lists **loc**.*es* that are present for each cons-node of L, where the size of **loc**.*es* is equal to the integer *fld hd* of the cons-node. The consecutive numbers are obtained by taking them from the inherited attribute s that is threaded to the end of the list. The computation that defines **loc**.*es* is given as a monadic expression **loc**.m:

```
data IntList  | Nil
              | Cons   hd :: Int    tl : IntList
attr IntList   inh s :: Int    syn r :: IntList
sem IntList
   | Nil    lhs.r          = [ ]
   | Cons lhs.r            = @loc.es ++ @tl.r
           (loc.es, tl.s)  = runState @loc.m @lhs.s
           loc.m           = replicateM @hd $ do
                                 e ← get
                                 modify (+1)
                                 return e
```

The state monad takes the initial counter, produces the result **loc**.*es* and an updated counter, which is subsequently passed on to the tail of the list as *tl*.*c*.

Concerns. This simple example demonstrates the use of monads in rules. It also shows that attributes have to be threaded into and out of the monad (e.g. via *runState*). Such rules that interface with the monad are tedious to write because they mention all attributes that play a role in the monad. This becomes more of an issue when multiple of these rules occur in a production, because of the threading of the attributes between rules and children. Thus, such a construction impairs the ability to describe rules for attributes separately and thus affects the composability of the description.

Code as the above is also prone to mistakes in attribute names that lead to accidental cycles in the threading of attributes, e.g.:

$$\begin{array}{lll}
(..., \mathbf{loc}.s_1) & = ...\ @\mathbf{lhs}.s & \\
(..., \mathbf{loc}.s_2) & = ...\ @\mathbf{loc}.s_2 & \text{-- cycle: should have been } s_1 \\
(..., \mathit{tl}.s) & = ...\ @\mathbf{loc}.s_2 &
\end{array}$$

Fortunately, this classical mistake is caught by the static dependency analysis of attribute grammars. It would otherwise lead to hard to find cases of non termination.

Composable descriptions. As a solution to the composability issues, we show another use of higher-order children (Section 4.3). First we introduce a nonterminal $M\ \phi\ \alpha$ with a single production *Do* that represents a monadic computation that it obtains as inherited attribute *expr* of type $\mathit{State}\ \phi\ \alpha$, where ϕ is the type of the state and α is the result type:

$$\begin{array}{llll}
\mathbf{data}\ M\ @\phi\ @\alpha\ \mid \mathit{Do} & & & \\
\mathbf{attr}\ M & \mathbf{inh}\ \mathit{expr} & :: \mathit{State}\ \phi\ \alpha & \\
 & \mathbf{chn}\ s & :: \phi & \\
 & \mathbf{syn}\ a & :: \alpha & \\
\mathbf{sem}\ M\ \mid \mathit{Do} & (\mathbf{lhs}.a, \mathbf{lhs}.s) = \mathit{runState}\ @\mathbf{lhs}.\mathit{expr}\ @\mathbf{lhs}.s & &
\end{array}$$

Given a tree $M\ \phi\ \alpha$, we can obtain the result of the monadic computation as attribute a, and also have the input and output state as chained attribute s. We can construct such a tree using the constructor *Do*, but how to integrate it with the actual tree?

This is where higher-order children come in again. The following example shows how to declare a higher-order child m_1, its definition and the threading of the attributes:

$$\begin{array}{lll}
\mathbf{sem}\ \mathit{IntList}\ \mid \mathit{Cons} & \mathbf{inst}.m_1 & : M \\
 & \mathbf{inst}.m_1 & = \mathit{Do} \\
 & m_1.\mathit{expr} & = @\mathbf{loc}.m \\
 & \mathbf{loc}.\mathit{es} & = @m_1.a \\
 & m_1.s & = \mathbf{lhs}.s \\
 & \mathit{tl}.s & = m_1.s
\end{array}$$

Inlining these definitions gives actually the same code as the former example. The difference is the ability to specify the threading of the attributes separately and factoring out the wrapping code of the monads. In addition, copy rules (Section 4.2) may take care of the threading rules altogether.

9.2 Integration: Attribute Grammars as Monads

The previous section showed rules containing monadic RWS operations. Dealing with impure monadic operations is more involving, as we discuss in this section. Of particular interest are *IO* and *ST* operations. The ability to e.g. update auxiliary data while processing a tree opens up a whole range of applications.

At first glance, monadic operations may not appear as quite a challenge because attribute grammars can be mapped to a sequential computation (Section 3.2) and the resulting computation can be represented as a monadic computation so that rules can be an arbitrary monadic expression. However, a declarative formalism is a double-edged sword in this setting. The evaluation of rules depends only on data dependencies, which gives little guarantees with respect to when rules are evaluated, if at all. To be able to use monadic operations, we need to provide stronger guarantees, e.g. that monadic effects are always performed and at most once.

Example. To introduce monadic rules, we give a variant of the unique number dispenser of Figure 3. When there is only the requirement that the produced numbers are unique but not that they are sequential, we can pass a reference to a shared counter as an inherited attribute and use monadic code to fetch-and-increment it:

```
attr Uniq   inh hCounter :: TVar Int   syn value :: Int
sem Uniq
   | Next   lhs.value ← atomically $ do
                          c ← readTVar @lhs.hCounter
                          writeTVar $! c + 1
                          return c
```

This example features a monadic rule, which is a rule of the form $p \leftarrow m$ where p is a pattern and m a monadic expression. It has the expected semantics: it is translated to $m' \gg\!= \lambda p' \rightarrow r$, where m' and p' are the respective translations of m and p, and r is the remainder of the computation that is scheduled after the rule. Monadic rules are scheduled as eager rules, and in addition are evaluated even when there is no data dependency on the left-hand side.

10 Feature: Inversion of Control

A common pattern that appears when writing tree computations is to first perform some initial computation over the tree (e.g. spreading environments), followed by an iterative computation (e.g. computing some fixpoint), followed by a resulting computation (e.g. producing a transformed tree and collecting error messages). This section provides a construction for expressing this pattern, and as it turns out use it to encode the monadic rules of the previous section.

Iteration. There are several ways to incorporate iterative or fixpoint computations in attribute grammars (Farrow, 1986). Using Haskell, lazy evaluation can be exploited to obtain iteration by lifting attributes to lists and giving a collection of cyclic attribute definitions that define the value of index i in the list in terms of values in the list of attributes at indices $j<i$ (preferably $j = i-1$). However, it is tedious to write these equations especially when different attributes are involved in the cycle. Moreover, the rule ordering cannot be expressed this way.

We present a different solution that extends cycle-free attribute grammars with an inversion of control construction that can be used to express iteration. The general idea is that we can obtain from a child a function f that represents the computation of a subset of its attributes, and can replace it with another function. To this end, we need additional syntax to specify which attributes are involved and to specify a transformation function of the function that computes these attributes.

Syntax and semantics. We define the *explicit attribute set* (EAS) of a nonterminal as a subset of the attributes of the nonterminal. The syntax to declare it is similar to attribute declarations, except that it uses the keyword **expl** and the attribute types are omitted.

Declaring an EAS has the following consequences. For each nonterminal N with an EAS, a production containing a child $n : N$ must define an attribute **expl**.n. This function gets as parameter the evaluator for the attributes in the EAS, and must give such an evaluator as result. Consequently, we can influence the application of the evaluator for a particular subset of the attributes.

The identity transformation is obtained by defining **expl**.$n = id$, and more complex transformations are obtained by exploiting that the evaluator is a monadic function that takes a record containing values of the inherited attributes in the EAS and a monadic continuation that receives a record containing values of the synthesized attributes in the EAS.

With the current construction, only one EAS can be specified per nonterminal. This is not a limitation as the constructions are composable by introducing proxy nonterminals (Section 4.4). This is also a good practice when inversion of control is not required for each occurrence of a nonterminal symbol.

Static dependencies. To ensure that we can obtain an evaluator that takes the inherited attributes in one go we impose the static restriction that the inherited attributes in the EAS may not depend on any of the synthesized attributes in the EAS, and that each synthesized attribute in the EAS depends on each inherited attribute. This additionally ensures that the evaluation occurs only in the child, and does not require evaluation at a parent node. Furthermore, to have the **expl**.n attribute available for such a node n when computing the attributes in the EAS set, it needs to be an additional dependency of $n.a$ for all attributes a in the EAS.

```
data P @α          -- placeholder for a monadic computation
  | Nop
attr P          chn st       :: StateToken
                syn mbVal :: Maybe α
expl P          chn st    syn value
sem P | Nop     lhs.st         $= @lhs.st
                lhs.mbVal  = Nothing
sem M | M       inst.act  : P
                inst.act  = Nop
                expl.act = λf i k → f i $ λs →
                   @lhs.expr ≫= λa → k s {mbVal = Just v}
                lhs.value = fromJust @act.mbVal
```

Fig. 7: Monadic operations via inversion of control.

Implementation. The implementation of this feature is surprisingly straightforward. When we schedule a visit v to a node to compute a synthesized attribute mentioned in the EAS, then it needs to schedule all the attributes in the EAS. We precede v with an additional visit u that can take care of other attributes that may be involved that are not in the EAS. With this approach, when scheduling a visit v for some node n, we simply call the function defined by **expl**.n (which will be in scope) with the evaluator for v (which will also be in scope) instead of calling the evaluator for v directly.

We desire a least number of computations in v to prevent duplicate work when re-applying the evaluator. Eager rules aside, the strategy of evaluating only the rules that are needed for producing the values for the synthesized attributes scheduled to v ensures that we do not compute additional results that are discarded when reinvoking the evaluator. The purpose of u is to compute all attributes that are dependencies of synthesized attributes in the EAS but that do not depend on inherited attributes in the EAS. This requires a similar enhancement to the scheduler as discussed in Section 6: when we schedule a visit, we can specify additionally a set of synthesized attributes for which the scheduler schedules all dependencies that can be scheduled, e.g. which depend only on inherited attributes that are available so far.

Expressiveness. The construction in this section is expressive:

- Figure 7 shows how to encode the monadic actions of Section 9.2 with it. The nonterminal P serves as a placeholder that computes *Nothing*, but exhibits the desired scheduling constraints. Its evaluation is transformed to execute the monadic action and update the result with it. We can thus eliminate the language-specific monadic rules with the more general and language-independent construction shown in this section.
- Figure 8 shows exception handling and backtracking. Suppose that N is a nonterminal that provides two ways for computing the synthesized attributes

```
attr N        inh e :: Maybe BacktrackException
expl N        inh e
data M | P    c : N
sem  M | P    c.e     = Nothing
              expl.c = λf i k → catch (f i k) (λex → f i {e = Just ex} k)
```

Fig. 8: Example of exception handling and backtracking.

depending on an inherited attribute e. If some exception occurs during the first way, we want it to take the alternative way, which we accomplish by running the evaluator with a different value for e. In generel, this construction makes it possible to integrate Iteratees (Kiselyov, 2012) and stepwise evaluation (Middelkoop et al., 2011).

11 Related Work

Background. Attribute grammars where introduced by Knuth (1968) to define the semantics of context free languages, and have since found their application in compiler generation. The circularity of attribute grammars is a prominent topic in related literature. Bird (1984) provided the basis for attribute grammars as circular functional programs (Johnsson, 1987). Swierstra and Alcocer (1998) give the corresponding translation to Haskell, and show the advantages of embedded Haskell code in rules.

In a different setting, Kennedy and Warren (1976) gave an abstract interpretation of acyclic attribute grammars for the generation of efficient evaluators, but may require the evaluator to support a number of visit sequences that are exponential in the number of attributes. Kastens (1980) showed an approach that is incomplete but requires only a single visit sequence. Saraiva and Swierstra (1999b) showed a continuation-based translation to strict functional programs for this case. Bransen et al. (2012) report that Kastens' approach is too restrictive in the context of UHC, and propose a functional implementation of the Kennedy-Warren approach instead which does not exhibit exponential behavior in practice.

UUAGC. The Utrecht University Attribute Grammar Compiler (UUAGC) is the source of inspiration for this paper. The requests for the features discussed in this paper originated from the UHC project (Dijkstra et al., 2009) and from students taking a course on program analysis. UUAGC supports higher-order children, demand-driven and statically ordered attribute evaluators, and polymorphism and overloading. It offers various forms of code generation, including monadic code that it can additionally exploit for generating a parallel evaluator.

The idea related to eager rules originates from a research project (Middelkoop, 2012) and corresponding prototype implementation (Middelkoop et al., 2010). We made this idea suitable for attribute grammars (this paper) and are integrating it into UUAGC.

Functional programming. Besides attribute grammar preprocessors such as the UUAGC and Happy, there are also deep embeddings (de Moor et al., 2000; Viera et al., 2009). The deep embeddings integrate well with the type system, and the preprocessors usually leave type checking to Haskell. Recently, Kaminski and Van Wyk (2011) showed the inverse direction: how to incorporate functional programming features into attribute grammars, including type inference, polymorphic types, and pattern matching.

12 Conclusion

Purely functional programming languages and attribute grammars fit well together, because purity gives the necessary freedom for scheduling attribute computations. Previous work has shown that Haskell is in particular a good host language because its lazy evaluation provides most of the machinery needed to implement attribute grammars.

Some desirable Haskell features raise challenges when combined with attribute grammars, and this paper presented solutions to these challenges. These challenges included the support of data types with higher kinds and monadic effects. Our solutions relied on two general attribute grammar techniques that we used throughout the paper: higher-order children and static attribute scheduling. On top of these extensions, we proposed eager rules to influence the static scheduling.

Some of the addressed challenges are strictly spoken not unique to Haskell, but do show up more prominently when using Haskell. The attribute grammar extension that we propose is however not language specific and thus offers a general solution that is useful for other languages as well.

This paper can therefore also be seen as motivation for investing the effort of incorporating extensions such as higher-order children into an attribute grammar system. This paper also showed the need for static attribute scheduling, and the question remains how we can further exploit it. In contrast to higher-order children, the attribute scheduling is not so easily implemented and clashes with some extensions that are of a dynamic nature. This potentially asks for approaches to combine demand driven and statically ordered attribute evaluation.

Bibliography

Bird, R. S. (1984). Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250.

Bransen, J., Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2012). The Kennedy-Warren Algorithm Revisited: Ordering Attribute Grammars. In *PADL '12*, pages 183–197.

de Moor, O., Backhouse, K., and Swierstra, S. D. (2000). First-class Attribute Grammars. *Informatica*, 24(3).

Dijkstra, A., Fokker, J., and Swierstra, S. D. (2009). The Architecture of the Utrecht Haskell Compiler. In *Haskell Symposium*, pages 93–104.

Farrow, R. (1986). Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. In *CC '86*, pages 85–98.

Johnsson, T. (1987). Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173.

Kaminski, T. and Van Wyk, E. (2011). Integrating Attribute Grammar and Functional Programming Language Features. In *SLE*, pages 263–282.

Kastens, U. (1980). Ordered Attributed Grammars. *Acta Informatica*, 13:229–256.

Kennedy, K. and Warren, S. K. (1976). Automatic Generation of Efficient Evaluators for Attribute Grammars. In *POPL '76*, pages 32–49.

Kiselyov, O. (2012). Iteratees. In *FLOPS*, pages 166–181.

Knuth, D. E. (1968). Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145.

Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A Generic Deriving Mechanism for Haskell. In *Haskell*, pages 37–48.

Meijer, E., Fokkinga, M. M., and Paterson, R. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA*, pages 124–144.

Meijer, E. and Jeuring, J. (1995). Merging Monads and Folds for Functional Programming. In *AFP*, volume 925, pages 228–266.

Middelkoop, A. (2012). *Inference of Program Properties with Attribute Grammars, Revisited*. PhD thesis, Universiteit Utrecht.

Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2010). Iterative Type Inference with Attribute Grammars. In *GPCE '10*, pages 43–52.

Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2011). Stepwise Evaluation of Attribute Grammars. In *LDTA*, page 5.

Saraiva, J. and Swierstra, S. D. (1999a). Generic Attribute Grammars.

Saraiva, J. and Swierstra, S. D. (1999b). Purely Functional Implementation of Attribute Grammars. Technical report, Universiteit Utrecht.

Schrijvers, T. and Oliveira, B. C. d. S. (2011). Monads, Zippers and Views: Virtualizing the Monad Stack. In *ICFP*, pages 32–44.

Swierstra, S. D. and Alcocer, P. R. A. (1998). Attribute Grammars in the Functional Style. In *Systems Implementation 2000*, pages 180–193.

Swierstra, S. D. and Chitil, O. (2009). Linear, Bounded, Functional Pretty-Printing. *JFP*, 19(1):1–16.

Viera, M., Swierstra, D., and Middelkoop, A. (2011). UUAG Meets AspectAG - How to make Attribute Grammars First-Class. Technical Report UU-CS-2011-029, Universiteit Utrecht.

Viera, M., Swierstra, S. D., and Swierstra, W. (2009). Attribute Grammars Fly First-Class: how to do Aspect Oriented Programming in Haskell. In *ICFP '09*, pages 245–256.

Vogt, H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher-Order Attribute Grammars. In *PLDI '89*, pages 131–145.