Attribute Grammars Fly First-Class

How to do Aspect Oriented Programming in Haskell

Marcos Viera
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy

S. Doaitse Swierstra
Department of Computer Science
Utrecht University
Utrecht, The Netherlands
doaitse@cs.uu.nl

Wouter Swierstra
Chalmers University of Technology
Göteborg, Sweden
wouter@chalmers.se

Abstract

Attribute Grammars (AGs), a general-purpose formalism for describing recursive computations over data types, avoid the trade-off which arises when building software incrementally: should it be easy to add new data types and data type alternatives or to add new operations on existing data types? However, AGs are usually implemented as a pre-processor, leaving e.g. type checking to later processing phases and making interactive development, proper error reporting and debugging difficult. Embedding AG into Haskell as a combinator library solves these problems.

Previous attempts at embedding AGs as a domain-specific language were based on extensible records and thus exploiting Haskell's type system to check the well-formedness of the AG, but fell short in compactness and the possibility to abstract over oft occurring AG patterns. Other attempts used a very generic mapping for which the AG well-formedness could not be statically checked.

We present a typed embedding of AG in Haskell satisfying all these requirements. The key lies in using HList-like typed heterogeneous collections (extensible polymorphic records) and expressing AG well-formedness conditions as type-level predicates (i.e., type-class constraints). By further type-level programming we can also express common programming patterns, corresponding to the typical use cases of monads such as $Reader$, $Writer$ and $State$. The paper presents a realistic example of type-class-based type-level programming in Haskell.

Categories and Subject Descriptors D.3.3 [*Programming languages*]: Language Constructs and Features; D.1.1 [*Programming techniques*]: Applicative (Functional) Programming

General Terms Design, Languages, Performance, Standardization

Keywords Attribute Grammars, Class system, Lazy evaluation, Type-level programming, Haskell, HList

1. Introduction

Functional programs can be easily extended by defining extra functions. If however a data type is extended with a new alternative, each parameter position and each case expression where a value of this type is matched has to be inspected and modified accordingly. In object oriented programing the situation is reversed: if we implement the alternatives of a data type by sub-classing, it is easy to add a new alternative by defining a new subclass in which we define a method for each part of desired global functionality. If however we want to define a new function for a data type, we have to inspect all the existing subclasses and add a method describing the local contribution to the global computation over this data type. This problem was first noted by Reynolds (Reynolds 1975) and later referred to as "the expression problem" by Wadler (Wadler 1998). We start out by showing how the use of AGs overcomes this problem.

As running example we use the classic *repmin* function (Bird 1984); it takes a tree argument, and returns a tree of similar shape, in which the leaf values are replaced by the minimal value of the leaves in the original tree (see Figure 1). The program was originally introduced to describe so-called circular programs, i.e. programs in which part of a result of a function is again used as one of its arguments. We will use this example to show that the computation is composed of three so-called *aspects*: the computation of the minimal value as the first component of the result of sem_Tree (asp_smin), passing down the globally minimal value from the root to the leaves as the parameter $ival$ (asp_ival), and the construction of the resulting tree as the second component of the result (asp_sres).

Now suppose we want to change the function *repmin* into a function *repavg* which replaces the leaves by the average value of the leaves. Unfortunately we have to change almost every line of the program, because instead of computing the minimal value we have to compute both the sum of the leaf values and the total number of leaves. At the root level we can then divide the total sum by the total number of leaves to compute the average leaf value. However, the traversal of the tree, the passing of the value to be used in constructing the new leafs and the construction of the new tree all remain unchanged. What we are now looking for is a way to define the function *repmin* as:

$$repmin = sem\_Root\ (asp\_smin \oplus asp\_ival \oplus asp\_sres)$$

so we can easily replace the aspect asp_smin by asp_savg:

$$repavg = sem\_Root\ (asp\_savg \oplus asp\_ival \oplus asp\_sres)$$

In Figure 2 we have expressed the solution of the *repmin* problem in terms of a domain specific language, i.e., as an attribute grammar (Swierstra et al. 1999). Attributes are values associated with tree nodes. We will refer to a collection of (one or more) related attributes, with their defining rules, as an aspect. After defining the underlying data types by a few **DATA** definitions, we define the different aspects: for the two "result" aspects we

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

```
data Root = Root Tree
data Tree = Node Tree Tree
          | Leaf Int

repmin = sem_Root

sem_Root (Root tree)
  =         let (smin, sres) = (sem_Tree tree) smin
            in  (sres)

sem_Tree (Node l r)
  = λival → let (lmin, lres) = (sem_Tree l    ) ival
                (rmin, rres) = (sem_Tree r    ) ival
            in  (lmin ‘min‘ rmin, Node lres rres)

sem_Tree (Leaf i)
  = λival →     (i, Leaf ival)
```

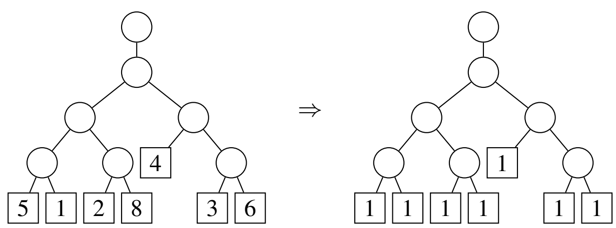

Figure 1. *repmin* replaces leaf values by their minimal value

```
DATA Root | Root tree
DATA Tree | Node l, r : Tree
          | Leaf i : {Int}
SYN Tree  [smin : Int]
SEM Tree
       | Leaf lhs .smin = @i
       | Node lhs .smin = @l.smin ‘min‘ @r.smin
INH Tree [ival : Int]
SEM Root
       | Root tree.ival  = @tree.smin
SEM Tree
       | Node l    .ival  = @lhs.ival
              r    .ival  = @lhs.ival
SYN Root Tree [sres : Tree]
SEM Root
       | Root lhs .sres  = @tree.sres
SEM Tree
       | Leaf lhs .sres  = Leaf @lhs.ival
       | Node lhs .sres  = Node @l.sres @r.sres
```

Figure 2. AG specification of repmin

introduce synthesized attributes (**SYN** *smin* and **SYN** *sres*), and for the "parameter" aspect we introduce an inherited attribute (**INH** *ival*).

Note that attributes are introduced separately, and that for each attribute/alternative pair we have a separate piece of code describing what to compute in a **SEM** rule; the defining expressions at the right hand side of the =-signs are all written in Haskell, using minimal syntactic extensions to refer to attribute values (the identifiers starting with a @). These expressions are copied directly into the generated program: only the attribute references are replaced by references to values defined in the generated program. The attribute grammar system only checks whether for all attributes a definition has been given. Type checking of the defining expressions is left to the Haskell compiler when compiling the generated program (given in Figure 1).

As a consequence type errors are reported in terms of the generated program. Although this works reasonably well in practice, the question arises whether we we can define a set of combinators which enables us to embed the AG formalism directly in Haskell, thus making the separate generation step uncalled for and immediately profiting from Haskell's type checker and getting error messages referring to the original source code.

A first approach to such an embedded attribute grammar notation was made by de Moor et al. (de Moor et al. 2000b). Unfortunately this approach, which is based on extensible records (Gaster and Jones 1996), necessitates the introduction of a large set of combinators, which encode positions of children-trees explicitly. Furthermore combinators are indexed by a number which indicates the number of children a node has where the combinator is to be applied. The *first contribution* of this paper is that we show how to overcome this shortcoming by making use of the Haskell class system.

The *second contribution* is that we show how to express the previous solution in terms of heterogeneous collections, thus avoiding the use of Hugs-style extensible records are not supported by the main Haskell compilers.

Attribute grammars exhibit typical design patterns; an example of such a pattern is the inherited attribute *ival*, which is distributed to all the children of a node, and so on recursively. Other examples are attributes which thread a value through the tree, or collect information from all the children which have a specific attribute and combine this into a synthesized attribute of the father node. In normal Haskell programming this would be done by introducing a collection of monads (*Reader*, *State* and *Writer* monad respectively), and by using monad transformers to combine these in to a single monadic computation. Unfortunately this approach breaks down once too many attributes have to be dealt with, when the data flows backwards, and especially if we have a non-uniform grammar, i.e., a grammar which has several different non-terminals each with a different collection of attributes. In the latter case a single monad will no longer be sufficient.

One way of making such computational patterns first-class is by going to a universal representation for all the attributes, and packing and unpacking them whenever we need to perform a computation. In this way all attributes have the same type at the attribute grammar level, and non-terminals can now be seen as functions which map dictionaries to dictionaries, where such dictionaries are tables mapping *String*s representing attribute names to universal attribute values (de Moor et al. 2000a). Although this provides us with a powerful mechanism for describing attribute flows by Haskell functions, this comes at a huge price; all attributes have to be unpacked before the contents can be accesses, and to be repacked before they can be passed on. Worse still, the check that verifies that all attributes are completely defined, is no longer a static check, but rather something which is implicitly done at run-time by the evaluator, as a side-effect of looking up attributes in the dictionaries. The *third contribution* of this paper is that we show how patterns corresponding to the mentioned monadic constructs can be described, again using the Haskell class mechanism.

The *fourth contribution* of this paper is that it presents yet another large example of how to do type-level programming in Haskell, and what can be achieved with it. In our conclusions we will come back to this.

Before going into the technical details we want to give an impression of what our embedded Domain Specific Language (DSL)

```
data Root = Root{tree :: Tree}
          deriving Show
data Tree = Node{l :: Tree, r :: Tree}
          | Leaf{i :: Int}
          deriving Show

$(deriveAG '' Root)
$(attLabels ["smin", "ival", "sres"])

asp_smin = synthesize smin       at {Tree}
           use        min  0     at {Node}
           define at  Leaf = i
asp_ival = inherit    ival       at {Tree}
           copy   at  {Node}
           define at  Root.tree = tree.smin
asp_sres = synthesize sres              at {Root, Tree}
           use        Node (Leaf 0)     at {Node}
           define at  Root = tree.sres
                      Leaf = Leaf lhs.ival

asp_repmin = asp_smin ⊕ asp_sres ⊕ asp_ival
repmin t = select sres from compute asp_repmin t
```

Figure 3. *repmin* in our embedded DSL

looks like. In Figure 3 we give our definition of the *repmin* problem in a lightly sugared notation.

To completely implement the *repmin* function the user of our library[1] needs to undertake the following steps (Figure 3):

- define the Haskell data types involved;
- optionally, generate some boiler-plate code using calls to Template Haskell;
- define the aspects, by specifying whether the attribute is inherited or synthesized, with which non-terminals it is associated, how to compute its value if no explicit definition is given (i.e., which computational pattern it follows), and providing definitions for the attribute at the various data type constructors (productions in grammar terms) for which it needs to be defined, resulting in *asp_repmin*;
- composing the aspects into a single large aspect *asp_repmin*
- define the function *repmin* that takes a *tree*, executes the semantic function for the tree and the aspect *asp_repmin*, and selects the synthesized attribute *sres* from the result.

Together these rules define for each of the productions a so-called Data Dependency Graph (DDG). A DDG is basically a data-flow graph (Figure 4), with as incoming values has the inherited attributes of the father node and the synthesized attributes of the children nodes (indicated by closed arrows), and as outputs the inherited attributes of the children nodes and the synthesized attributes of the father node (open arrows). The semantics of our DSL is defined as the data-flow graph which results from composing all the DDGs corresponding to the individual nodes of the abstract syntax tree. Note that the semantics of a tree is thus represented by a function which maps the inherited attributes of the root node onto its synthesized attributes.

The main result of this paper is a combinator based implementation of attribute grammars in Haskell; it has statically type checked

[1] Available as *AspectAG* in Hackage.

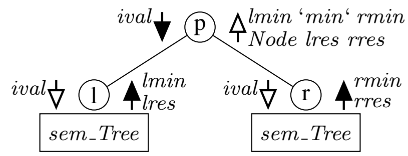

Figure 4. The DDG for *Node*

semantic functions, it is statically checked for correctness at the attribute grammar level, and high-level attribute evaluation patterns can be described.

In Section 2 we introduce the heterogeneous collections, which are used to combine a collection of inherited or synthesised attributes into a single value. In Section 3 we show how individual attribute grammar rules are represented. In Section 4 we introduce the aforementioned ⊕ operator which combines the aspects. In Section 5 we introduce a function *knit* which takes the DDG associated with the production used at the root of a tree and the mappings (*sem_...* functions) from inherited to synthesised attributes for its children (i.e. the data flow over the children trees) and out of this constructs a data flow computation over the combined tree. In Section 6 we show how the common patterns can be encoded in our library, and in Section 7 we show how default aspects can be defined. In Section 8 we discuss related work, and in Section 9 we conclude.

2. HList

The library HList (Kiselyov et al. 2004) implements typeful heterogeneous collections (lists, records, ...), using techniques for dependently typed programming in Haskell (Hallgren 2001; McBride 2002) which in turn make use of Haskell 98 extensions for multi-parameter classes (Peyton Jones et al. 1997) and functional dependencies (Jones 2000). The idea of *type-level programming* is based on the use of types to represent type-level values, and classes to represent type-level types and functions.

In order to be self-contained we start out with a small introduction. To represent Boolean values at the type level we define a new type for each of the Boolean values. The class *HBool* represents the type-level type of Booleans. We may read the instance definitions as “the type-level values *HTrue* and *HFalse* have the type-level type *HBool*”:

```
class    HBool x
data HTrue ; hTrue  = ⊥ :: HTrue
data HFalse; hFalse = ⊥ :: HFalse
instance HBool HTrue
instance HBool HFalse
```

Since we are only interested in type-level computation, we defined *HTrue* and *HFalse* as empty types. By defining an inhabitant for each value we can, by writing expressions at the value level, construct values at the type-level by referring to the types of such expressions.

Multi-parameter classes can be used to describe type-level *relations*, whereas functional dependencies restrict such relations to functions. As an example we define the class *HOr* for type-level disjunction:

```
class (HBool t, HBool t', HBool t'')
   ⇒ HOr t t' t'' | t t' → t''
  where hOr :: t → t' → t''
```

The context $(HBool\ t, HBool\ t', HBool\ t'')$ expresses that the types t, t' and t'' have to be type-level values of the type-level type $HBool$. The functional dependency $t\ t' \rightarrow t''$ expresses that the parameters t and t' uniquely determine the parameter t''. This implies that once t and t' are instantiated, the instance of t'' must be uniquely inferable by the type-system, and that thus we are defining a type-level function from t and t' to t''. The type-level function itself is defined by the following non-overlapping instance declarations:

$$\begin{aligned}&\textbf{instance}\ \mathit{HOr}\ \mathit{HFalse}\ \mathit{HFalse}\ \mathit{HFalse}\\ &\quad\textbf{where}\ \mathit{hOr}\ \_\ \_ = \mathit{hFalse}\\ &\textbf{instance}\ \mathit{HOr}\ \mathit{HTrue}\ \mathit{HFalse}\ \mathit{HTrue}\\ &\quad\textbf{where}\ \mathit{hOr}\ \_\ \_ = \mathit{hTrue}\\ &\textbf{instance}\ \mathit{HOr}\ \mathit{HFalse}\ \mathit{HTrue}\ \mathit{HTrue}\\ &\quad\textbf{where}\ \mathit{hOr}\ \_\ \_ = \mathit{hTrue}\\ &\textbf{instance}\ \mathit{HOr}\ \mathit{HTrue}\ \mathit{HTrue}\ \mathit{HTrue}\\ &\quad\textbf{where}\ \mathit{hOr}\ \_\ \_ = \mathit{hTrue}\end{aligned}$$

If we write $(\mathit{hOr}\ \mathit{hTrue}\ \mathit{hFalse})$, we know that t and t' are HTrue and HFalse, respectively. So, the second instance is chosen to select hOr from and thus t'' is inferred to be HTrue.

Despite the fact that is looks like a computation at the value level, its actual purpose is to express a computation at the type-level; no interesting value level computation is taking place at all. If we had defined HTrue and HFalse in the following way:

$$\begin{aligned}&\textbf{data}\ \mathit{HTrue}\ = \mathit{HTrue}\,; \mathit{hTrue} = \mathit{HTrue}\ :: \mathit{Htrue}\\ &\textbf{data}\ \mathit{HFalse} = \mathit{HFalse}; \mathit{hFalse} = \mathit{HFalse} :: \mathit{HFalse}\end{aligned}$$

then the same computation would also be performed at the value level, resulting in the value HTrue of type HTrue.

2.1 Heterogeneous Lists

Heterogeneous lists are represented with the data types HNil and HCons, which model the structure of a normal list both at the value and type level:

$$\begin{aligned}&\textbf{data}\ \mathit{HNil} = \mathit{HNil}\\ &\textbf{data}\ \mathit{HCons}\ e\ l = \mathit{HCons}\ e\ l\end{aligned}$$

The sequence $\mathit{HCons}\ \mathit{True}\ (\mathit{HCons}\ \texttt{"bla"}\ \mathit{HNil})$ is a correct heterogeneous list with type $\mathit{HCons}\ \mathit{Bool}\ (\mathit{HCons}\ \mathit{String}\ \mathit{HNil})$. Since we want to prevent that an expression $\mathit{HCons}\ \mathit{True}\ \mathit{False}$ represents a correct heterogeneous list (the second HCons argument is not a type-level list) we introduce the classes HList and its instances,and express express this constraint by adding a context condition to the $\mathit{HCons}...$ instance:

$$\begin{aligned}&\textbf{class}\ \mathit{HList}\ l\\ &\textbf{instance}\ \mathit{HList}\ \mathit{HNil}\\ &\textbf{instance}\ \mathit{HList}\ l \Rightarrow \mathit{HList}\ (\mathit{HCons}\ e\ l)\end{aligned}$$

The library includes a multi-parameter class $\mathit{HExtend}$ to model the extension of heterogeneous collections.

$$\begin{aligned}&\textbf{class}\ \mathit{HExtend}\ e\ l\ l' \mid e\ l \rightarrow l', l' \rightarrow e\ l\\ &\quad\textbf{where}\ \mathit{hExtend} :: e \rightarrow l \rightarrow l'\end{aligned}$$

The functional dependency $e\ l \rightarrow l'$ makes that $\mathit{HExtend}$ is a type-level function, instead of a relation: once e and l are fixed l' is uniquely determined. It fixes the type l' of a collection, resulting from extending a collection of type l with an element of type e. The member $\mathit{hExtend}$ performs the same computation at the level of values. The instance of $\mathit{HExtend}$ for heterogeneous lists includes the well-formedness condition:

$$\begin{aligned}&\textbf{instance}\ \mathit{HList}\ l \Rightarrow \mathit{HExtend}\ e\ l\ (\mathit{HCons}\ e\ l)\\ &\quad\textbf{where}\ \mathit{hExtend} = \mathit{HCons}\end{aligned}$$

The main reason for introducing the class $\mathit{HExtend}$ is to make it possible to encode constraints on the things which can be HCons-ed; here we have expressed that the second parameter should be a list again. In the next subsection we will see how to make use of this facility.

2.2 Extensible Records

In our code we will make heavy use of non-homogeneous collections: grammars are a collection of productions, and nodes have a collection of attributes and a collection of children nodes. Such collections, which can be extended and shrunk, map typed labels to values and are modeled by an HList containing a heterogeneous list of fields, marked with the data type Record. We will refer to them as records from now on:

$$\textbf{newtype}\ \mathit{Record}\ r = \mathit{Record}\ r$$

An empty record is a Record containing an empty heterogeneous list:

$$\begin{aligned}&\mathit{emptyRecord} :: \mathit{Record}\ \mathit{HNil}\\ &\mathit{emptyRecord} = \mathit{Record}\ \mathit{HNil}\end{aligned}$$

A field with label l (a phantom type (Hinze 2003)) and value of type v is represented by the type:

$$\textbf{newtype}\ \mathit{LVPair}\ l\ v = \mathit{LVPair}\{\mathit{valueLVPair} :: v\}$$

Labels are now almost first-class objects, and can be used as type-level values. We can retrieve the label value using the function $\mathit{labelLVPair}$, which exposes the phantom type parameter:

$$\begin{aligned}&\mathit{labelLVPair} :: \mathit{LVPair}\ l\ v \rightarrow l\\ &\mathit{labelLVPair} = \bot\end{aligned}$$

Since we need to represent many labels, we introduce a polymorphic type Proxy to represent them; by choosing a different phantom type for each label to be represented we can distinguish them:

$$\textbf{data}\ \mathit{Proxy}\ e; \mathit{proxy} = \bot :: \mathit{Proxy}\ e$$

Thus, the following declarations define a record (myR) with two elements, labelled by $\mathit{Label1}$ and $\mathit{Label2}$:

$$\begin{aligned}&\textbf{data}\ \mathit{Label1}; \mathit{label1} = \mathit{proxy} :: \mathit{Proxy}\ \mathit{Label1}\\ &\textbf{data}\ \mathit{Label2}; \mathit{label2} = \mathit{proxy} :: \mathit{Proxy}\ \mathit{Label2}\\ &\mathit{field1} = \mathit{LVPair}\ \mathit{True}\ \ :: \mathit{LVPair}\ (\mathit{Proxy}\ \mathit{Label1})\ \mathit{Bool}\\ &\mathit{field2} = \mathit{LVPair}\ \texttt{"bla"} :: \mathit{LVPair}\ (\mathit{Proxy}\ \mathit{Label2})\ [\mathit{Char}]\\ &\mathit{myR} = \mathit{Record}\ (\mathit{HCons}\ \mathit{field1}\ (\mathit{HCons}\ \mathit{field2}\ \mathit{HNil}))\end{aligned}$$

Since our lists will represent collections of attributes we want to express statically that we do not have more than a single definition for each attribute occurrence, and so the labels in a record should be all different. This constraint is represented by requiring an instance of the class $\mathit{HRLabelSet}$ to be available when defining extendability for records:

$$\begin{aligned}&\textbf{instance}\ \mathit{HRLabelSet}\ (\mathit{HCons}\ (\mathit{LVPair}\ l\ v)\ r)\\ &\quad\Rightarrow \mathit{HExtend}\ (\mathit{LVPair}\ l\ v)\ (\mathit{Record}\ r)\\ &\qquad\qquad (\mathit{Record}\ (\mathit{HCons}\ (\mathit{LVPair}\ l\ v)\ r))\\ &\quad\textbf{where}\ \mathit{hExtend}\ f\ (\mathit{Record}\ r) = \mathit{Record}\ (\mathit{HCons}\ f\ r)\end{aligned}$$

The class $\mathit{HasField}$ is used to retrieve the value part corresponding to a specific label from a record:

$$\begin{aligned}&\textbf{class}\ \mathit{HasField}\ l\ r\ v \mid l\ r \rightarrow v\ \textbf{where}\\ &\quad\mathit{hLookupByLabel} :: l \rightarrow r \rightarrow v\end{aligned}$$

At the type-level it is statically checked that the record r indeed has a field with label l associated with a value of the type v. At value-level the member $\mathit{hLookupByLabel}$ returns the value of type v. So, the following expression returns the string "bla":

$$\mathit{hLookupByLabel}\ \mathit{label2}\ \mathit{myR}$$

The possibility to update an element in a record at a given label position is provided by:

$$\textbf{class}\ \mathit{HUpdateAtLabel}\ l\ v\ r\ r'\ |\ l\ v\ r \rightarrow r'\ \textbf{where}$$
$$\mathit{hUpdateAtLabel} :: l \rightarrow v \rightarrow r \rightarrow r'$$

In order to keep our programs readable we introduce infix operators for some of the previous functions:

$$(.*.) \quad = \mathit{hExtend}$$
$$\_\ .=.\ v = \mathit{LVPair}\ v$$
$$r\ \#\ l\ = \mathit{hLookupByLabel}\ l\ r$$

Furthermore we will use the following syntactic sugar to denote lists and records in the rest of the paper:

- $\{\ v1, ..., vn\ \}$ for $(v1\ .*.\ ...\ .*.\ vn\ .*.\ \mathit{HNil})$
- $\{\{\ v1, ..., vn\ \}\}$ for $(v1\ .*.\ ...\ .*.\ vn\ .*.\ \mathit{emptyRecord})$

So, for example the definition of myR can now be written as:

$$\mathit{myR} = \{\{\ \mathit{label1}\ .=.\ \mathit{True}, \mathit{label2}\ .=.\ \texttt{"bla"}\ \}\}$$

3. Rules

In this subsection we show how attributes and their defining rules are represented. An *attribution* is a finite mapping from attribute names to attribute values, represented as a *Record*, in which each field represents the name and value of an attribute.

$$\textbf{type}\ \mathit{Att}\ \mathit{att}\ \mathit{val} = \mathit{LVPair}\ \mathit{att}\ \mathit{val}$$

The labels[2] (attribute names) for the attributes of the example are:

$$\textbf{data}\ \mathit{Att\_smin}; \mathit{smin} = \mathit{proxy} :: \mathit{Proxy}\ \mathit{Att\_smin}$$
$$\textbf{data}\ \mathit{Att\_ival};\ \ \mathit{ival}\ \ = \mathit{proxy} :: \mathit{Proxy}\ \mathit{Att\_ival}$$
$$\textbf{data}\ \mathit{Att\_sres};\ \ \mathit{sres}\ \ = \mathit{proxy} :: \mathit{Proxy}\ \mathit{Att\_sres}$$

When inspecting what happens at a production we see that information flows from the inherited attribute of the parent and the synthesized attributes of the children (henceforth called in the *input* family) to the synthesized attributes of the parent and the inherited attributes of the children (together called the *output* family from now on). Both the input and the output attribute family is represented by an instance of:

$$\textbf{data}\ \mathit{Fam}\ c\ p = \mathit{Fam}\ c\ p$$

A *Fam* contains a single attribution for the parent and a collection of attributions for the children. Thus the type p will always be a *Record* with fields of type *Att*, and the type c a *Record* with fields of the type:

$$\textbf{type}\ \mathit{Chi}\ \mathit{ch}\ \mathit{atts} = \mathit{LVPair}\ \mathit{ch}\ \mathit{atts}$$

where ch is a label that represents the name of that child and atts is again a *Record* with the fields of type *Att* associated with this particular child. In our example the *Root* production has a single child *Ch_tree* of type *Tree*, the *Node* production has two children labelled by *Ch_l* and *Ch_r* of type *Tree*, and the *Leaf* production has a single child called *Ch_i* of type *Int*. Thus we generate, using template Haskell:

$$\textbf{data}\ \mathit{Ch\_tree}; \mathit{ch\_tree} = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch\_tree}, \mathit{Tree})$$
$$\textbf{data}\ \mathit{Ch\_r};\quad \mathit{ch\_r}\quad = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch\_r}, \mathit{Tree})$$
$$\textbf{data}\ \mathit{Ch\_l};\quad \mathit{ch\_l}\quad = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch\_l}, \mathit{Tree})$$
$$\textbf{data}\ \mathit{Ch\_i};\quad \mathit{ch\_i}\quad = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch\_i}, \mathit{Int})$$

Note that we encode both the name and the type of the child in the type representing the label.

[2] These and all needed labels can be generated automatically by Template Haskell functions available in the library

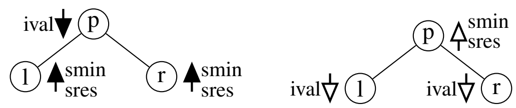

Figure 5. Repmin's input and output families for Node

Families are used to model the input and output attributes of attribute computations. For example, Figure 5 shows the input (black arrows) and output (white arrows) attribute families of the repmin problem for the production Node. We now give the attributions associated with the output family of the *Node* production, which are the synthesized attributes of the parent (*SP*) and the inherited attributions for the left and right child (*IL* and *IR*):

$$\textbf{type}\ \mathit{SP} = \mathit{Record}\ (\mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att\_smin})\ \mathit{Int})$$
$$\qquad \mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att\_sres})\ \ \mathit{Tree})$$
$$\qquad \mathit{HNil})$$
$$\textbf{type}\ \mathit{IL}\ = \mathit{Record}\ (\mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att\_ival})\ \ \mathit{Int})$$
$$\qquad \mathit{HNil})$$
$$\textbf{type}\ \mathit{IR}\ = \mathit{Record}\ (\mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att\_ival})\ \ \mathit{Int})$$
$$\qquad \mathit{HNil})$$

The next type collects the last two children attributions into a single record:

$$\textbf{type}\ \mathit{IC} = \mathit{Record}\ (\mathit{HCons}\ (\mathit{Chi}\ (\mathit{Proxy}\ (\mathit{Ch\_l}, \mathit{Tree})\ \ \mathit{IL})$$
$$\qquad \mathit{HCons}\ (\mathit{Chi}\ (\mathit{Proxy}\ (\mathit{Ch\_r}, \mathit{Tree})\ \mathit{IR})$$
$$\qquad \mathit{HNil})$$

We now have all the ingredients to define the output family for *Node*-s.

$$\textbf{type}\ \mathit{Output\_Node} = \mathit{Fam}\ \mathit{IC}\ \mathit{SP}$$

Attribute computations are defined in terms of *rules*. As defined by (de Moor et al. 2000a), a rule is a mapping from an input family to an output family. In order to make rules composable we define a rule as a mapping from input attributes to a function which extends a family of output attributes with the new elements defined by this rule:

$$\textbf{type}\ \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}'\ \mathit{sp}'$$
$$= \mathit{Fam}\ \mathit{sc}\ \mathit{ip} \rightarrow \mathit{Fam}\ \mathit{ic}\ \mathit{sp} \rightarrow \mathit{Fam}\ \mathit{ic}'\ \mathit{sp}'$$

Thus, the type *Rule* states that a rule takes as input the synthesized attributes of the children sc and the inherited attributes of the parent ip and returns a function from the output constructed thus far (inherited attributes of the children ic and synthesized attributes of the parent sp) to the extended output.

The composition of two rules is the composition of the two functions after applying each of them to the input family first:

$$\mathit{ext} :: \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}'\ \mathit{sp}'\ \mathit{ic}''\ \mathit{sp}'' \rightarrow \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}'\ \mathit{sp}'$$
$$\rightarrow \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}''\ \mathit{sp}''$$
$$(f\ \text{`}\mathit{ext}\text{`}\ g)\ \mathit{input} = f\ \mathit{input}.g\ \mathit{input}$$

3.1 Rule Definition

We now introduce the functions syndef and inhdef, which are used to define primitive rules which define a synthesized or an inherited attribute respectively. Figure 6 lists all the rule definitions for our running example. The naming convention is such that a rule with name $\mathit{prod\_att}$ defines the attribute att for the production prod. Without trying to completely understand the definitions we suggest the reader to compare them with their respective **SEM** specifications in Figure 2.

```
leaf_smin  (Fam chi par)
   = syndef smin (chi # ch_i)
node_smin (Fam chi par)
   = syndef smin (((chi # ch_l) # smin)
                    `min`
                   ((chi # ch_r) # smin))
root_ival   (Fam chi par)
   = inhdef ival  { nt_Tree }
                  {{ ch_tree
                       .=. (chi # ch_tree) # smin }}
node_ival   (Fam chi par)
   = inhdef ival  { nt_Tree }
                  {{ ch_l .=. par # ival
                   ,  ch_r .=. par # ival }}
root_sres   (Fam chi par)
   = syndef sres  ((chi # ch_tree) # sres)
leaf_sres   (Fam chi par)
   = syndef sres  (Leaf  (par # ival))
node_sres   (Fam chi par)
   = syndef sres  (Node ((chi # ch_l) # sres)
                        ((chi # ch_r) # sres))
```

Figure 6. Rule definitions for repmin

The function *syndef* adds the definition of a synthesized attribute. It takes a label *att* representing the name of the new attribute, a value *val* to be assigned to this attribute, and it builds a function which updates the output constructed thus far.

```
syndef :: HExtend (Att att val) sp sp'
       ⇒ att → val → (Fam ic sp → Fam ic sp')
syndef att val (Fam ic sp) = Fam ic (att .=. val .*. sp)
```

The record *sp* with the synthesized attributes of the parent is extended with a field with name *att* and value *val*, as shown in Figure 7. If we look at the type of the function, the check that we have not already defined this attribute is done by the constraint *HExtend* (*Att att val*) *sp* sp', meaning that sp' is the result of adding the field (*Att att val*) to *sp*, which cannot have any field with name *att*. Thus we are statically preventing duplicated attribute definitions.

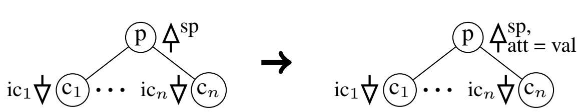

Figure 7. Synthesized attribute definition

Let us take a look at the rule definition *node_smin* of the attribute *smin* for the production *Node* in Figure 6. The children *ch_l* and *ch_r* are retrieved from the input family so we can subsequently retrieve the attribute *smin* from these attributions, and construct the computation of the synthesized attribute *smin*. This process is demonstrated in Figure 8. The attribute *smin* is required (underlined) in the children *l* and *r* of the input, and the parent of the output is extended with *smin*.
If we take a look at the type which is inferred for *node_sres* we find back all the constraints which are normally checked by an off-line attribute grammar system, i.e., an attribute *smin* is made available

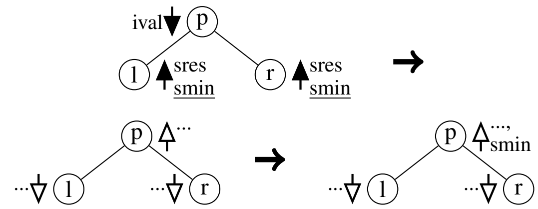

Figure 8. Rule *node_sres*

by each child and an attribute *smin* can be safely added to the current synthesized attribution of the parent: [3]

```
node_sres :: (HasField (Proxy (Ch_l, Tree)) sc scl
             , HasField (Proxy Att_smin) scl Int
             , HasField (Proxy (Ch_r, Tree)) sc scr
             , HasField (Proxy Att_smin) scr Int
             , HExtend (Att (Proxy Att_smin) Int)
                    sp sp')
          ⇒ Rule sc ip ic sp ic sp'
```

The function *inhdef* introduces a new inherited attribute for a collection of non-terminals. It takes the following parameters:

att the attribute which is being defined;

nts the non-terminals with which this attribute is being associated;

vals a record labelled with child names and containing values, describing how to compute the attribute being defined at each of the applicable child positions.

The parameter *nts* takes over the role of the **INH** declaration in Figure 2. Here this extra parameter seems to be superfluous, since its value can be inferred, but adds an additional restriction to be checked (yielding to better errors) and it will be used in the introduction of default rules later. The names for the non-terminals of our example are:

```
nt_Root = proxy :: Proxy Root
nt_Tree = proxy :: Proxy Tree
```

The result of *inhdef* again is a function which updates the output constructed thus far.

```
inhdef :: Defs att nts vals ic ic'
       ⇒ att → nts → vals → (Fam ic sp → Fam ic' sp)
inhdef att nts vals (Fam ic sp) =
       Fam (defs att nts vals ic) sp
```

The class *Def* is defined by induction over the record *vals* containing the new definitions. The function *defs* inserts each definition into the attribution of the corresponding child.

```
class Defs att nts vals ic ic' | vals ic → ic' where
  defs :: att → nts → vals → ic → ic'
```

We start out with the base case, where we have no more definitions to add. In this case the inherited attributes of the children are returned unchanged.

```
instance Defs att nts (Record HNil) ic ic where
  defs _ _ _ ic = ic
```

The instance for *HCons* given below first recursively processes the rest of the definitions by updating the collection of collections of inherited attributes of the children *ic* into ic'. A helper type level

[3] In order to keep the explanation simple we will suppose that *min* is not overloaded, and takes *Int*'s as parameter.

function $SingleDef$ (and its corresponding value level function $singledef$) is used to incorporate the single definition (pch) into ic', resulting in a new set ic''. The type level functions $HasLabel$ and $HMember$ are used to statically check whether the child being defined (lch) exists in ic' and if its type (t) belongs to the non-terminals nts, respectively. The result of both functions are $HBool$s (either $HTrue$ or $HFalse$) which are passed as parameters to $SingleDef$. We are now ready to give the definition for the non-empty case:

$$
\begin{array}{l}
\textbf{instance}\ (Defs\ att\ nts\ (Record\ vs)\ ic\ ic' \\
\qquad\quad ,HasLabel\ (Proxy\ (lch, t))\ ic'\ mch \\
\qquad\quad ,HMember\ (Proxy\ t)\ nts\ mnts \\
\qquad\quad ,SingleDef\ mch\ mnts\ att \\
\qquad\qquad (Chi\ (Proxy\ (lch, t))\ vch) \\
\qquad\qquad ic'\ ic'') \\
\quad \Rightarrow Defs\ att\ nts \\
\qquad\qquad (Record\ (HCons\ (Chi\ (Proxy\ (lch, t))\ vch)\ vs)) \\
\qquad\qquad ic\ ic'' \\
\quad \textbf{where} \\
\quad defs\ att\ nts\ \sim(Record\ (HCons\ pch\ vs))\ ic = \\
\qquad singledef\ mch\ mnts\ att\ pch\ ic' \\
\qquad \textbf{where}\ ic' \ = defs\ att\ nts\ (Record\ vs)\ ic \\
\qquad\qquad\quad lch \ = labelLVPair\ pch \\
\qquad\qquad\quad mch \ = hasLabel\ lch\ ic' \\
\qquad\qquad\quad mnts = hMember\ (sndProxy\ lch)\ nts
\end{array}
$$

The class $Haslabel$ can be encoded straightforwardly, together with a function which retrieves part of a phantom type:

$$
\begin{array}{l}
\textbf{class}\ HBool\ b \Rightarrow HasLabel\ l\ r\ b \mid l\ r \to b \\
\textbf{instance}\ HasLabel\ l\ r\ b \Rightarrow HasLabel\ l\ (Record\ r)\ b \\
\textbf{instance}\ (HEq\ l\ lp\ b, HasLabel\ l\ r\ b', HOr\ b\ b'\ b'') \\
\quad \Rightarrow HasLabel\ l\ (HCons\ (LVPair\ lp\ vp)\ r)\ b'' \\
\textbf{instance}\ HasLabel\ l\ HNil\ HFalse \\
hasLabel :: HasLabel\ l\ r\ b \Rightarrow l \to r \to b \\
hasLabel = \bot \\
sndProxy :: Proxy\ (a, b) \to Proxy\ b \\
sndProxy\ \_ = \bot
\end{array}
$$

We only show the instance with both mch and $mnts$ equal to $HTrue$, which is the case we expect to apply in a correct attribute grammar definition: we do not refer to children which do not exist, and this child has the type we expect.[4]

$$
\begin{array}{l}
\textbf{class}\ SingleDef\ mch\ mnts\ att\ pv\ ic\ ic' \\
\qquad \mid mch\ mnts\ pv\ ic \to ic' \\
\quad \textbf{where}\ singledef :: mch \to mnts \to att \to pv \to ic \to ic' \\
\textbf{instance}\ (HasField\ lch\ ic\ och \\
\qquad\quad ,HExtend\ (Att\ att\ vch)\ och\ och' \\
\qquad\quad ,HUpdateAtLabel\ lch\ och'\ ic\ ic') \\
\quad \Rightarrow SingleDef\ HTrue\ HTrue\ att\ (Chi\ lch\ vch)\ ic\ ic' \\
\quad \textbf{where}\ singledef\ \_\ \_\ att\ pch\ ic = \\
\qquad hUpdateAtLabel\ lch\ (att\ .=.\ vch\ .*.\ och)\ ic \\
\qquad \textbf{where}\ lch\ = labelLVPair\ \ pch \\
\qquad\qquad\quad vch = valueLVPair\ pch \\
\qquad\qquad\quad och = hLookupByLabel\ lch\ ic
\end{array}
$$

We will guarantee that the collection of attributions ic (inherited attributes of the children) contains an attribution och for the child lch, and so we can use $hUpdateAtlabel$ to extend the attribution for this child with a field ($Att\ att\ vch$), thus binding attribute att to value vch. The type system checks, thanks to the presence of $HExtend$, that the attribute att was not defined before in och.

4. Aspects

We represent aspects as records which contain for each production a rule field.

$$\textbf{type}\ Prd\ prd\ rule = LVPair\ prd\ rule$$

For our example we thus introduce fresh labels to refer to repmin's productions:

$$
\begin{array}{l}
\textbf{data}\ P\_Root;\ p\_Root\ = proxy :: Proxy\ P\_Root \\
\textbf{data}\ P\_Node; p\_Node = proxy :: Proxy\ P\_Node \\
\textbf{data}\ P\_Leaf;\ p\_Leaf\ \ = proxy :: Proxy\ P\_Leaf
\end{array}
$$

We now can define the aspects of repmin as records with the rules of Figure 6.[5]

$$
\begin{array}{l}
asp\_smin = \{\{\ p\_Leaf\ \ .=.\ leaf\_smin \\
\qquad\qquad\quad ,\ \ p\_Node\ .=.\ node\_smin\ \}\} \\
asp\_ival\ \ = \{\{\ p\_Root\ \ .=.\ root\_ival \\
\qquad\qquad\quad ,\ \ p\_Node\ .=.\ node\_ival\ \}\} \\
asp\_sres\ \ = \{\{\ p\_Root\ \ .=.\ root\_sres \\
\qquad\qquad\quad ,\ \ p\_Node\ .=.\ node\_sres \\
\qquad\qquad\quad ,\ \ p\_Leaf\ \ .=.\ leaf\_sres\ \}\}
\end{array}
$$

4.1 Aspects Combination

We define the class Com which will provide the instances we need for combining aspects:

$$
\begin{array}{l}
\textbf{class}\ Com\ r\ r'\ r'' \mid r\ r' \to r'' \\
\quad \textbf{where}\ (\ \oplus\ ) :: r \to r' \to r''
\end{array}
$$

With this operator we can now combine the three aspects which together make up the repmin problem:

$$asp\_repmin = asp\_smin \oplus asp\_ival \oplus asp\_sres$$

Combination of aspects is a sort of union of records where, in case of fields with the same label (i.e., for rules for the same production), the rule combination (ext) is applied to the values. To perform the union we iterate over the second record, inserting the next element into the first one if it is new and combining it with an existing entry if it exists:

$$
\begin{array}{l}
\textbf{instance}\ Com\ r\ (Record\ HNil)\ r \\
\quad \textbf{where}\ r \oplus \_ = r \\
\textbf{instance}\ (HasLabel\ lprd\ r\ b \\
\qquad\qquad ,ComSingle\ b\ (Prd\ lprd\ rprd)\ r\ r''' \\
\qquad\qquad ,Com\ r'''\ (Record\ r')\ r'') \\
\quad \Rightarrow Com\ r\ (Record\ (HCons\ (Prd\ lprd\ rprd)\ r'))\ r'' \\
\quad \textbf{where} \\
\qquad r \oplus (Record\ (HCons\ prd\ r')) = r'' \\
\qquad\quad \textbf{where}\ b\ \ \ = hasLabel\ (labelLVPair\ prd)\ r \\
\qquad\qquad\qquad r''' = comsingle\ b\ prd\ r \\
\qquad\qquad\qquad r''\ \ = r''' \oplus (Record\ r')
\end{array}
$$

We use the class $ComSingle$ to insert a single element into the first record. The type-level Boolean parameter b is used to distinguish those cases where the left hand operand already contains a field for the rule to be added and the case where it is new. [6]

[4] The instances for error cases could just be left undefined, yielding to "undefined instance" type errors. In our library we use a class $Fail$ (as defined in (Kiselyov et al. 2004), section 6) in order to get more instructive type error messages.

[5] We assume that the monomorphism restriction has been switched off.

[6] This parameter can be avoided by allowing overlapping instances, but we prefer to minimize the number of Haskell extensions we use.

```
class ComSingle b f r r' | b f r → r'
  where comsingle :: b → f → r → r'
```

If the first record has a field with the same label *lprd*, we update its value by composing the rules.

```
instance (HasField lprd      r (Rule sc ip ic' sp' ic'' sp'')
         , HUpdateAtLabel lprd (Rule sc  ip
                                     ic  sp
                                     ic'' sp'')
                           r r')
  ⇒ ComSingle HTrue (Prd lprd (Rule sc ip ic sp ic' sp'))
              r r'
  where
    comsingle _ f r = hUpdateAtLabel n ((r # n) `ext` v) r
      where n = labelLVPair f
            v = valueLVPair f
```

In case the first record does not have a field with the label, we just insert the element in the record.

```
instance ComSingle HFalse f (Record r)
                   (Record (HCons f r))
  where comsingle _ f (Record r) = Record (HCons f r)
```

5. Semantic Functions

Our overall goal is to construct a *Tree*-algebra and a *Root*-algebra. For the domain associated with each non-terminal we take the function mapping its inherited to its synthesized attributes. The hard work is done by the function *knit*, the purpose of which is to combine the data flow defined by the DDG –which was constructed by combining all the rules for this production– with the semantic functions of the children (describing the flow of data from their inherited to their synthesized attributes) into the semantic function for the parent.

With the attribute computations as first-class entities, we can now pass them as an argument to functions of the form *sem_<nt>*. The following code follows the definitions of the data types at hand: it contains recursive calls for all children of an alternative, each of which results in a mapping from inherited to synthesized attributes for that child followed by a call to knit, which stitches everything together:

```
sem_Root asp (Root t)
   = knit (asp # p_Root) {{ ch_tree .=. sem_Tree asp t }}
sem_Tree asp (Node l r)
   = knit (asp # p_Node) {{ ch_l .=. sem_Tree asp l
                          ,  ch_r .=. sem_Tree asp r }}
sem_Tree asp (Leaf i )
   = knit (asp # p_Leaf) {{ ch_i .=. sem_Lit i }}

sem_Lit e (Record HNil) = e
```

Since this code is completely generic we provide a Template Haskell function *deriveAG* which automatically generates the functions such as *sem_Root* and *sem_Tree*, together with the labels for the non-terminals and labels for referring to children. Thus, to completely implement the *repmin* function we need to undertake the following steps:

- Generate the semantic functions and the corresponding labels by using:

  ```
  $(deriveAG ''Root)
  ```

- Define and compose the aspects as shown in the previous sections, resulting in *asp_repmin*.
- Define the function *repmin* that takes a *tree*, executes the semantic function for the tree and the aspect *asp_repmin*, and selects the synthesized attribute *sres* from the result.

  ```
  repmin tree
     = sem_Root asp_repmin (Root tree) () # sres
  ```

5.1 The Knit Function

As said before the function *knit* takes the combined rules for a node and the semantic functions of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes. We start out by constructing an empty output family, containing an empty attribution for each child and one for the parent. To each of these attributions we apply the corresponding part of the rules, which will construct the inherited attributes of the children and the synthesized attributes of the parent (together forming the output family). Rules however contain references to the input family, which is composed of the inherited attributes of the parent *ip* and the synthesized attributes of the children *sc*.

```
knit :: (Empties fc ec, Kn fc ic sc)
     ⇒ Rule sc ip ec (Record HNil) ic sp
       → fc → ip → sp
knit rule fc ip =
  let ec           = empties fc
      (Fam ic sp) = rule (Fam sc ip)
                         (Fam ec emptyRecord)
      sc           = kn fc ic
  in  sp
```

The function *kn*, which takes the semantic functions of the children (*fc*) and their inputs (*ic*), computes the results for the children (*sc*). The functional dependency $fc \rightarrow ic\ sc$ indicates that *fc* determines *ic* and *sc*, so the shape of the record with the semantic functions determines the shape of the other records:

```
class Kn fc ic sc | fc → ic sc where
  kn :: fc → ic → sc
```

We declare a helper instance of *Kn* to remove the *Record* tags of the parameters, in order to be able to iterate over their lists without having to tag and untag at each step:

```
instance Kn fc ic sc
 ⇒ Kn (Record fc) (Record ic) (Record sc) where
  kn (Record fc) (Record ic) = Record $ kn fc ic
```

When the list of children is empty, we just return an empty list of results.

```
instance Kn HNil HNil HNil where
  kn _ _ = hNil
```

The function *kn* is a type level *zipWith* ($), which applies the functions contained in the first argument list to the corresponding element in the second argument list.

```
instance Kn fcr icr scr
      ⇒ Kn (HCons (Chi lch (ich → sch)) fcr)
           (HCons (Chi lch ich)          icr)
           (HCons (Chi lch sch)          scr)
   where
  kn ~(HCons pfch fcr) ~(HCons pich icr) =
    let scr = kn fcr icr
        lch = labelLVPair  pfch
        fch = valueLVPair pfch
        ich = valueLVPair pich
    in  HCons (newLVPair lch (fch ich)) scr
```

The class *Empties* is used to construct the record, with an empty attribution for each child, which we have used to initialize the computation of the input attributes with.

```
class Empties fc ec | fc → ec where
  empties :: fc → ec
```

In the same way that *fc* determines the shape of *ic* and *sc* in *Kn*, it also tells us how many empty attributions *ec* to produce and in which order:

```
instance Empties fc ec
     ⇒ Empties (Record fc) (Record ec) where
  empties (Record fc) = Record $ empties fc

instance Empties fcr ecr
     ⇒ Empties (HCons (Chi lch fch)              fcr)
                (HCons (Chi lch (Record HNil)) ecr)
       where
  empties ~(HCons pch fcr) =
    let ecr = empties fcr
        lch = labelLVPair pch
    in  HCons  (newLVPair lch emptyRecord) ecr

instance Empties HNil HNil where
  empties _ = hNil
```

6. Common Patterns

At this point all the basic functionality of attribute grammars has been implemented. In practice however we want more. If we look at the code in Figure 2 we see that the rules for *ival* at the production *Node* are "free of semantics", since the value is copied unmodified to its children. If we were dealing with a tree with three children instead of two the extra line would look quite similar. When programming attribute grammars such patterns are quite common and most attribute grammar systems contain implicit rules which automatically insert such "trivial" rules. As a result descriptions can decrease in size dramatically. The question now arises whether we can extend our embedded language to incorporate such more high level data flow patterns.

6.1 Copy Rule

The most common pattern is the copying of an inherited attribute from the parent to all its children. We capture this pattern with the an operator *copy*, which takes the name of an attribute *att* and an heterogeneous list of non-terminals *nts* for which the attribute has to be defined, and generates a copy rule for this. This corresponds closely to the introduction of a *Reader* monad.

```
copy :: (Copy att nts vp ic ic', HasField att ip vp)
     ⇒ att → nts → Rule sc ip ic sp ic' sp
```

Thus, for example, the rule *node_ival* of Figure 6 can now be written as:

```
node_ival input = copy ival { nt_Tree } input
```

The function *copy* uses a function *defcp* to define the attribute *att* as an inherited attribute of its children. This function is similar in some sense to *inhdef*, but instead of taking a record containing the new definitions it gets the value *vp* of the attribute which is to be copied to the children:

```
copy att nts (Fam _ ip) = defcp att nts (ip # att)

defcp :: Copy att nts vp ic ic'
      ⇒ att → nts → vp → (Fam ic sp → Fam ic' sp)
defcp att nts vp (Fam ic sp) =
  Fam (cpychi att nts vp ic) sp
```

The class *Copy* iterates over the record *ic* containing the output attribution of the children, and inserts the attribute *att* with value *vp* if the type of the child is included in the list *nts* of non-terminals and the attribute is not already defined for this child.

```
class Copy att nts vp ic ic' | ic → ic' where
  cpychi :: att → nts → vp → ic → ic'

instance Copy att nts vp (Record HNil) (Record HNil)
  where cpychi _ _ _ _ = emptyRecord

instance (Copy att nts vp (Record ics) ics'
         , HMember (Proxy t) nts mnts
         , HasLabel att vch mvch
         , Copy' mnts mvch att vp
                 (Chi (Proxy (lch, t)) vch)
                 pch
         , HExtend pch ics' ic)
  ⇒ Copy att nts vp
        (Record (HCons (Chi (Proxy (lch, t)) vch) ics))
        ic
  where
  cpychi att nts vp (Record (HCons pch ics)) =
    cpychi' mnts mvch att vp pch .*. ics'
    where ics'  = cpychi att nts vp (Record ics)
          lch   = sndProxy (labelLVPair pch)
          vch   = valueLVPair pch
          mnts  = hMember lch nts
          mvch  = hasLabel att vch
```

The function *cpychi'* updates the field *pch* by adding the new attribute:

```
class Copy' mnts mvch att vp pch pch'
      | mnts mvch pch → pch'
  where
    cpychi' :: mnts → mvch → att → vp → pch → pch'
```

When the type of the child doesn't belong to the non-terminals for which the attribute is defined we define an instance which leaves the field *pch* unchanged.

```
instance Copy' HFalse mvch att vp pch pch where
  cpychi' _ _ _ _ pch = pch
```

We also leave *pch* unchanged if the attribute is already defined for this child.

```
instance Copy' HTrue HTrue att vp pch pch where
  cpychi' _ _ _ _ pch = pch
```

In other case the attribution *vch* is extended with the attribute (*Att att vp*).

```
instance HExtend (Att att vp) vch vch'
  ⇒ Copy' HTrue HFalse att vp (Chi lch vch)
     (Chi lch vch') where
  cpychi' _ _ att vp pch = lch .=. (att .=. vp .*. vch)
    where lch = labelLVPair pch
          vch = valueLVPair pch
```

6.2 Other Rules

In this section we introduce two more constructs of our DSL, without giving their implementation. Besides the *Reader* monad, there is also the *Writer* monad. Often we want to collect information provided by some of the children into an attribute of the parent. This can be used to e.g. collect all identifiers contained in an expression. Such a synthesized attribute can be declared using the *use* rule, which combines the attribute values of the children in similar way as Haskell's *foldr1*. The *use* rule takes the following arguments: the attribute to be defined, the list of non-terminals for which the attribute is defined, a monoidal operator which combines the attribute values, and a unit value to be used in those cases where none of the children has such an attribute.

$$\begin{aligned}&use :: (Use\ att\ nts\ a\ sc, HExtend\ (Att\ att\ a)\ sp\ sp')\\&\quad \Rightarrow att \to nts \to (a \to a \to a) \to a\\&\qquad \to Rule\ sc\ ip\ ic\ sp\ ic\ sp'\end{aligned}$$

Using this new combinator the rule $node\_smin$ of Figure 6 becomes:

$$node\_smin = use\ smin\ \{\ nt\_Tree\ \}\ min\ 0$$

A third common pattern corresponds to the use of the *State* monad. A value is threaded in a depth-first way through the tree, being updated every now and then. For this we have chained attributes (both inherited and synthesized). If a definition for a synthesized attribute of the parent with this name is missing we look for the right-most child with a synthesized attribute of this name. If we are missing a definition for one of the children, we look for the right-most of its left siblings which can provide such a value, and if we cannot find it there, we look at the inherited attributes of the father.

$$\begin{aligned}&chain :: (Chain\ att\ nts\ val\ sc\ ic\ sp\ ic'\ sp'\\&\qquad , HasField\ att\ ip\ val)\\&\quad \Rightarrow att \to nts \to Rule\ sc\ ip\ ic\ sp\ ic'\ sp'\end{aligned}$$

7. Defining Aspects

Now we have both implicit rules to define attributes, and explicit rules which contain explicit definitions, we may want to combine these into a single *attribute aspect* which contains all the definitions for single attribute. We now refer to Figure 9 which is a desugared version of the notation presented in the introduction.

An inherited attribute aspect, like asp_ival in Figure 9, can be defined using the function $inhAspect$. It takes as arguments: the name of the attribute att, the list nts of non-terminals where the attribute is defined, the list $cpys$ of productions where the copy rule has to be applied, and a record $defs$ containing the explicit definitions for some productions:

$$\begin{aligned}&inhAspect\ att\ nts\ cpys\ defs\\&\quad = (defAspect\ (FnCpy\ att\ nts)\ cpys)\\&\quad \oplus (attAspect\ (FnInh\ att\ nts)\ \ defs)\end{aligned}$$

The function $attAspect$ generates an attribute aspect given the explicit definitions, whereas $defAspect$ constructs an attribute aspect based in a common pattern's rule. Thus, an inherited attribute aspect is defined as a composition of two attribute aspects: one with the explicit definitions and other with the application of the copy rule. In the following sections we will see how $attAspect$ and $defAspect$ are implemented.

A synthesized attribute aspect, like asp_smin and asp_sres in Figure 9, can be defined using $synAspect$. Here the rule applied is the use rule, which takes op as the monoidal operator and $unit$ as the unit value.

$$\begin{aligned}&synAspect\ att\ nts\ op\ unit\ uses\ defs\\&\quad = (defAspect\ (FnUse\ att\ nts\ op\ unit)\ uses)\\&\quad \oplus (attAspect\ (FnSyn\ att)\qquad\qquad defs)\end{aligned}$$

A chained attribute definition introduces both an inherited and a synthesized attribute. In this case the pattern to be applied is the chain rule.

$$\begin{aligned}&chnAspect\ att\ nts\ chns\ inhdefs\ syndefs\\&\quad = (defAspect\ (FnChn\ att\ nts)\ chns)\\&\quad \oplus (attAspect\ (FnInh\ att\ nts)\ \ inhdefs)\\&\quad \oplus (attAspect\ (FnSyn\ att)\qquad syndefs)\end{aligned}$$

7.1 Attribute Aspects

Consider the explicit definitions of the aspect asp_sres. The idea is that, when declaring the explicit definitions, instead of completely writing the rules, like:

$$\begin{aligned}&\{\{\ p\_Root .=. (\lambda input \to\\&\qquad\qquad syndef\ sres\ ((chi\ input\ \#\ ch\_tree)\ \#\ sres))\\&,\ \ p\_Leaf\ .=. (\lambda input \to\\&\qquad\qquad syndef\ sres\ (Leaf\ (par\ input\ \#\ ival)))\ \}\}\end{aligned}$$

we just define a record with the functions from the input to the attribute value:

$$\begin{aligned}&\{\{\ p\_Root .=. (\lambda input \to (chi\ input\ \#\ ch\_tree)\ \#\ sres)\\&,\ \ p\_Leaf\ .=. (\lambda input \to Leaf\ (par\ input\ \#\ ival))\ \}\}\end{aligned}$$

By mapping the function $((.)\ (syndef\ \ sres))$ over such records, we get back our previous record containing rules. The function $attAspect$ updates all the values of a record by applying a function to them:

$$\begin{aligned}&\textbf{class}\ AttAspect\ rdef\ defs\ rules \mid rdef\ defs \to rules\\&\qquad \textbf{where}\ attAspect :: rdef \to defs \to rules\\&\textbf{instance}\ (AttAspect\ rdef\ (Record\ defs)\ rules\\&\qquad\quad , Apply\ rdef\ def\ rule\\&\qquad\quad , HExtend\ (Prd\ lprd\ rule)\ rules\ rules')\\&\qquad \Rightarrow AttAspect\ rdef\\&\qquad\qquad\qquad (Record\ (HCons\ (Prd\ lprd\ def)\\&\qquad\qquad\qquad\qquad\qquad\qquad defs))\\&\qquad\qquad\qquad rules'\\&\quad \textbf{where}\\&\qquad attAspect\ rdef\ (Record\ (HCons\ def\ defs)) =\\&\qquad\quad \textbf{let}\ lprd = (labelLVPair\ def)\\&\qquad\quad \textbf{in}\ \ lprd .=. apply\ rdef\ (valueLVPair\ def)\\&\qquad\qquad .*. attAspect\ rdef\ (Record\ defs)\\&\textbf{instance}\ AttAspect\ rdef\ (Record\ HNil)\ (Record\ HNil)\\&\quad \textbf{where}\ attAspect\ \_\ \_ = emptyRecord\end{aligned}$$

The class $Apply$ (from the HList library) models the function application, and it is used to add specific constraints on the types:

$$\begin{aligned}&\textbf{class}\ Apply\ f\ a\ r \mid f\ a \to r\ \textbf{where}\\&\quad apply :: f \to a \to r\end{aligned}$$

In the case of synthesized attributes we apply $((.)\ (syndef\ att))$ to values of type $(Fam\ \ sc\ \ ip\ \ \to\ \ val)$ in order to construct a rule of type $(Rule\ \ sc\ \ ip\ \ ic\ \ sp\ \ ic\ \ sp')$. The constraint $HExtend\ (LVPair\ att\ val)\ sp\ sp'$ is introduced by the use of $syndef$. The data type $FnSyn$ is used to determine which instance of $Apply$ has to be chosen.

$$\begin{aligned}&\textbf{data}\ FnSyn\ att = FnSyn\ att\\&\textbf{instance}\ HExtend\ (LVPair\ att\ val)\ sp\ sp'\\&\quad \Rightarrow Apply\ (FnSyn\ att)\ (Fam\ sc\ ip \to val)\\&\qquad\qquad (Rule\ sc\ ip\ ic\ sp\ ic\ sp')\ \textbf{where}\\&\quad apply\ (FnSyn\ att)\ f = syndef\ att.f\end{aligned}$$

In the case of inherited attributes the function applied to define the rule is $((.)\ (inhdef\ att\ nts))$.

$$\begin{aligned}&\textbf{data}\ FnInh\ att\ nt = FnInh\ att\ nt\\&\textbf{instance}\ Defs\ att\ nts\ vals\ ic\ ic'\\&\quad \Rightarrow Apply\ (FnInh\ att\ nts)\ (Fam\ sc\ ip \to vals)\end{aligned}$$

```
asp_smin = synAspect smin { nt_Tree }                                                  -- synthesize at
                          min 0 { p_Node }                                             -- use at
                          {{ p_Leaf .=. (λ(Fam chi _) → chi # ch_i) }}                 -- define at
asp_ival  = inhAspect ival  { nt_Tree }                                                -- inherit
                          { p_Node }                                                   -- copy at
                          {{ p_Root .=. (λ(Fam chi _) → {{ ch_tree .=. (chi # ch_tree) # smin }}) }}  -- define at
asp_sres  = synAspect sres  { nt_Root, nt_Tree }                                       -- synthesize at
                          Node (Leaf 0) { p_Node }                                     -- use at
                          {{ p_Root .=. (λ(Fam chi _   ) → (chi # ch_tree) # sres)     -- define at
                          ,  p_Leaf .=. (λ(Fam _   par) → Leaf (par # ival)) }}
```

Figure 9. Aspects definition for repmin

```
      (Rule sc ip ic sp ic' sp) where
  apply (FnInh att nts) f = inhdef att nts.f
```

7.2 Default Aspects

The function $defAspect$ is used to construct an aspect given a rule and a list of production labels.

```
class DefAspect deff prds rules | deff prds → rules
  where defAspect :: deff → prds → rules
```

It iterates over the list of labels $prds$, constructing a record with these labels and a rule determined by the parameter $deff$ as value. For inherited attributes we apply the copy rule $copy\ att\ nts$, for synthesized attributes $use\ att\ nt\ op\ unit$ and for chained attributes $chain\ att\ nts$. The following types are used, in a similar way than in $attAspect$, to determine the rule to be applied:

```
data FnCpy att nts = FnCpy att nts
data FnUse att nt op unit = FnUse att nt op unit
data FnChn att nt = FnChn att nt
```

Thus, for example in the case of the aspect asp_ival, the application:

```
defAspect (FnCpy ival { nt_Tree }) { p_Node }
```

generates the default aspect:

```
{{ p_Node .=. copy ival { nt_Tree } }}
```

8. Related Work

There have been several previous attempts at incorporating first-class attribute grammars in lazy functional languages. To the best of our knowledge all these attempts exploit some form of extensible records to collect attribute definitions. They however do not exploit the Haskell class system as we do. de Moor et al. (2000b) introduce a whole collection of functions, and a result it is no longer possible to define copy, use and chain rules. Other approaches fail to provide some of the static guarantees that we have enforced (de Moor et al. 2000a).

The exploration of the limitations of type-level programming in Haskell is still a topic of active research. For example, there has been recent work on modelling relational data bases using techniques similar to those applied in this paper (Silva and Visser 2006).

As to be expected the type-level programming performed here in Haskell can also be done in dependently typed languages such as Agda (Norell 2008; Oury and Swierstra 2008). By doing so, we use Boolean values in type level-functions, thereby avoiding the need for a separate definition of the type-level Booleans. This would certainly simplify certain parts of our development. On the other hand, because Agda only permits the definition of total functions, we would need to maintain even more information in our types to make it evident that all our functions are indeed total.

An open question is how easy it will be to extend the approach taken to more global strategies of accessing attributes definitions; some attribute grammars systems allow references to more remote attributes (Reps et al. 1986; Boyland 2005). Although we are convinced that we can in principle encode such systems too, the question remains how much work this turns out to be.

Another thing we could have done is to make use of associated types (Chakravarty et al. 2005) in those cases where our relations are actually functions; since this feature is still experimental and has only recently become available we have refrained from doing so for the moment.

9. Conclusions

In the first place we remark that we have achieved all four goals stated in the introduction:

1. removing the need for a whole collection of indexed combinators as used in (de Moor et al. 2000b)
2. replacing extensible records completely by heterogeneous collections
3. the description of common attribute grammar patterns in order to reduce code size, and making them almost first class objects
4. give a nice demonstration of type level programming

We have extensive experience with attribute grammars in the construction of the Utrecht Haskell compiler (Dijkstra et al. 2009). The code of this compiler is completely factored out along the two axes mentioned in the introduction (Dijkstra and Swierstra 2004; Fokker and Swierstra 2008; Dijkstra et al. 2007), using the notation used in Figure 2. In doing so we have found the possibility to factor the code into separate pieces of text indispensable.

We also have come to the conclusion that the so-called monadic approach, although it may seem attractive at first sight, in the end brings considerable complications when programs start to grow (Jones 1999). Since monad transformers are usually type based we already run into problems if we extend a state twice with a value of the same type without taking explicit measures to avoid confusion. Another complication is that the interfaces of non-terminals are in general not uniform, thus necessitating all kind of tricks to change the monad at the right places, keeping information to be reused later, etc. In our generated Haskell compiler (Dijkstra et al. 2009) we have non-terminals with more than 10 different attributes, and glueing all these together or selectively leaving some out turns out to be impossible to do by hand.

In our attribute grammar system ($uuagc$ on Hackage), we perform a global flow analysis, which makes it possible to schedule the computations explicitly (Kastens 1980). Once we know the evaluation order we do not have to rely on lazy evaluation, and all parameter positions can be made strict. When combined with a uniqueness analysis we can, by reusing space occupied by unreachable attributes, get an even further increase in speed. This leads to a considerable, despite constant, speed improvement. Unfortunately we do not see how we can perform such analyses with the approach described in this paper: the semantic functions defining the values of the attributes in principle access the whole input family, and we cannot find out which functions only access part of such a family, and if so which part.

Of course a straightforward implementation of extensible records will be quite expensive, since basically we use nested pairs to represent attributions. We think however that a not too complicated program analysis will reveal enough information to be able to transform the program into a much more efficient form by flattening such nested pairs. Note that thanks to our type-level functions, which are completely evaluated by the compiler, we do not have to perform any run-time checks as in (de Moor et al. 2000a): once the program type-checks there is nothing which will prevent it to run to completion, apart form logical errors in the definitions of the attributes.

Concluding we think that the library described here is quite useful and relatively easy to experiment with. We notice furthermore that a conventional attribute grammar restriction, stating that no attribute should depend on itself, does not apply since we build on top of a lazily evaluated language. An example of this can be found in online pretty printing (Swierstra 2004; Swierstra and Chitil 2009). Once we go for speed it may become preferable to use more conventional off-line generators. Ideally we should like to have a mixed approach in which we can use the same definitions as input for both systems.

10. Acknowledgments

We want to thank Oege de Moor for always lending an ear in discussing the merits of attribute grammars, and how to further profit from them. Marcos Viera wants to thank the EU project Lernet for funding his stay in Utrecht. Finally, we would like to thank the anonymous referees for their helpful reviews.

References

Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf.*, 21:239–250, 1984.

John Boyland. Remote attribute grammars. *Journal of the ACM (JACM*, 52(4), Jul 2005. URL `http://portal.acm.org/citation.cfm?id=1082036.1082042`.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM.

Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000a.

Oege de Moor, L. Peyton Jones, Simon, and Van Wyk, Eric. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000b. Springer-Verlag. ISBN 3-540-41172-0.

Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In *Implementation of Functional Languages*, 2007.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell Symposium*, New York, NY, USA, September 2009. ACM.

Jeroen Fokker and S. Doaitse Swierstra. Abstract interpretation of functional programs using an attribute grammar system. In Adrian Johnstone and Jurgen Vinju, editors, *Language Descriptions, Tools and Applications*, 2008.

Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. NOTTCS-TR 96-3, Nottingham, 1996.

Thomas Hallgren. Fun with functional dependencies or (draft) types as values in static computations in haskell. In *Proc. of the Joint CS/CE Winter Meeting*, 2001.

Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999. URL `http://www.cse.ogi.edu/ mpj/thih/thih-sep1-1999/`.

P. Jones, Mark. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13: 229–256, 1980.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

Ulf Norell. Dependently typed programming in Agda. In *6th International School on Advanced Functional Programming*, 2008.

Nicolas Oury and Wouter Swierstra. The power of Pi. In *ICFP '08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, 2008.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

W. Reps, Thomas, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 1986. ACM.

J.C. Reynolds. User defined types and procedural data as complementary approaches to data abstraction. In S.A. Schuman, editor, *New Directions in Algorithmic Languages*. INRIA, 1975.

Alexandra Silva and Joost Visser. Strong types for relational databases. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-489-8.

S. Doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *Journal of Functional Programming*, 19(01):1–16, 2009.

S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João A. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

S.D. Swierstra. Linear, online, functional pretty printing (extended and corrected version). Technical Report UU-CS-2004-025a, Inst. of Information and Comp. Science, Utrecht Univ., 2004.

Phil Wadler. The Expression Problem. E-mail available online., 1998.