

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

CS & E Department
Chalmers University of Technology
Göteborg, Sweden
wouter@chalmers.se

Abstract

Attribute grammar programs are factored along two axes: the function axis and the data axis. As a consequence they can be easily adapted and extended. The question arises whether the attribute grammar approach to programming can be made available as an embedded domain specific language (EDSL) in Haskell. In this paper we show how this can be achieved by making extensive use of the type class mechanism. The paper identifies the characteristic properties of attribute grammars, shows how to emulate them, and shows how to overcome some problems associated with earlier attempts to do so. Finally we show how common patterns found in attribute grammar programming can be made explicit, leading to extensions of our EDSL.

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 programming 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).

Thus far one of the solutions to this problem has been the use of attribute grammars; whereas early attribute grammar systems

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

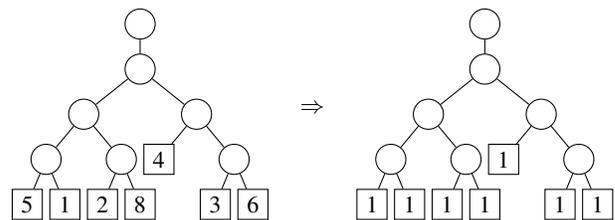


Figure 1. *repmim* replaces leaf values by their minimal value

favoured a more functional view, the newer systems really allow one to choose along which axis to structure one’s program and even switching structuring axis within a single program poses no problems. The question to be answered in this paper is whether, and if so how and with which limitations, we can bring this way of structuring your program text to the Haskell world.

As running example we use the classic *repmim* function (Bird 1984). This function takes as argument a tree, 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). Its code is given in Figure 2. It 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. As we can see from the code this is the case at the *Root* level: the computed minimal value *smin* is passed to the function *sem_Tree tree* and passed down to its recursive calls, so it can be used in the construction of the *Leaf*-s of the resulting tree, which is returned in *sres*.

If we now want to change the function *repmim* into a function *repavg* which replaces the leaves by the average value of the leaves, we have to make quite a few changes: instead of computing the minimal value we have to compute the sum of the leaf values and the total number of leaves and at the root level we have to divide the first one by the second one to get the value to be used in the construction of the new tree. On the other hand the traversal of the tree, passing the value to be used in constructing the new leaves, and the construction of the new tree remain unchanged. Apparently the function *repmim* consists of three *aspects*: collecting information in an upwards fashion, passing down the resulting value, and constructing the resulting tree. What we are now looking for is a way to define the function *repmim* as:

```

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 2. The *repmIn* code

```
repmIn = sem_Root (asp_smin ⊕ asp_ival ⊕ asp_sres)
```

so we can subsequently define the function *repavg* by just defining the new aspect *asp_savg*, and then write:

```
repavg = sem_Root (asp_savg ⊕ asp_ival ⊕ asp_sres)
```

In Figure 3 we have expressed to solution of the *repmIn* problem in terms of a domain specific language, i.e., as an attribute grammar (Swierstra et al. 1999); in fact the *repmIn* code of Figure 2 has been generated from this description. Note that attributes are introduced separately, and that for each attribute/alternative pair we have a separate piece of code describing what to compute; the defining expressions (at the right hand side of the =-signs) are all written in Haskell with some syntactic extensions to refer to attribute values (the identifiers starting with a @). These expressions are copied almost verbatim-wise into the generated program, with only the attribute references replaced by references to values in defined in the generated program. With respect to correctness the attribute grammar system only checks whether for all attributes a definition has been given, whereas type checking of the defining expressions is left to the Haskell compiler which compiles the generated program.

```

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 3. AG specification of *repmIn*

As a consequence type errors are reported in terms of the generated program. Although this works very well in practice, the ques-

tion arises whether we really need such a special domain specific language, and whether we can define a set of combinators which enables us to encode attribute grammars directly in Haskell, thus making the separate generation step uncalled for and immediately profiting from Haskell type checker and its error messages at the original source level.

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 makes heavy use of so-called extensible records (Gaster and Jones 1996), necessitates the introduction of a large set of combinators, which essentially encoding 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 which do not seem to become part of future Haskell.

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., one in which we have 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 however, the static check, which verifies that for all attributes proper definitions have been provided, 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 the patterns corresponding to the aforementioned monadic constructs can be described, again using the Haskell class mechanism.

To give an impression of how our embedded DSL looks like we give some syntactically sugared code in Figure 4. In the figure we see that each of the three aspects is being defined by a separate piece of code, in which we specify 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 the definitions for the attribute at the various rules where it needs to be defined.

Together these rules define for each of the productions a so-called Data Dependency Graph (DDG), which basically is a data-flow graph (Figure 5), with as incoming values the inherited at-

```

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_repmim = asp_smin  $\oplus$  asp_sres  $\oplus$  asp_ival

repmim t = select sres from compute asp_repmim t

```

Figure 4. Our embedded DSL

tributes of the father node and the synthesized attributes of the children nodes (indicated by open arrows), and as outputs the inherited attributes of the children nodes and the synthesized attributes of the father node (closed arrows). The semantics of our DSL is defined as the data-flow graph which is composed of all the DGG's 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 its inherited attributes onto its synthesized attributes.

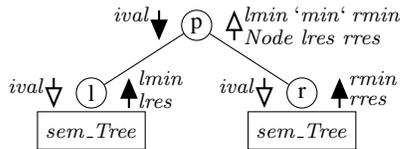


Figure 5. The DDG for Node

As a result we have constructed a combinator based implementation of attribute grammars in Haskell; it has statically type checked semantic functions, it is statically checked for correctness at the attribute grammar level, and high-level attribute evaluation patterns can be described.

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.

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 \oplus operator which combines the aspects. In Section 5 we introduce a function *knit* which takes the DGG 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) 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 de-

pendently typed programming in Haskell (Hallgren 2001; McBride 2002) which in their turn are making 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* consists in the use of types to represent type-level values, and classes to represent type-level types and functions. Suppose we want to represent Boolean values at the type level. In that case we define two types, *HTrue* and *HFalse*

```

data HTrue
hTrue =  $\perp$  :: HTrue

data HFalse
hFalse =  $\perp$  :: HFalse

```

Since we are only interested in type-level computation, we defined *HTrue* and *HFalse* as empty types, with as the only possible inhabitant the untouchable value \perp . The class *HBool* represents the type-level type of Booleans. We may read the following code as “the type-level values *HTrue* and *HFalse* have the type-level type *HBool*”:

```

class HBool x
instance HBool HTrue
instance HBool HFalse

```

Finally, we use multi-parameter classes to describe type-level *relations* and functional dependencies to restrict such relations to functions. For example, we define the class *HOr* for the 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' -> 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 mutually exclusive instance declarations:

```

instance HOr HFalse HFalse HFalse
where hOr _ _ = hFalse

instance HOr HTrue HFalse HTrue
where hOr _ _ = hTrue

instance HOr HFalse HTrue HTrue
where hOr _ _ = hTrue

instance HOr HTrue HTrue HTrue
where hOr _ _ = hTrue

```

If we write (*hOr hTrue hFalse*), we know that *t* and *t'* are *HTrue* and *False*, respectively. So, the second instance is chosen to select *hOr* from and thus *t''* is inferred to be *HTrue*.

Despite the fact that it 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:

```

data HTrue = HTrue
hTrue :: HTrue
hTrue = HTrue

data HFalse = HFalse
hFalse :: HFalse
hFalse = HFalse

```

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:

```
data  $HNil = HNil$ 
data  $HCons\ e\ l = HCons\ e\ l$ 
```

The sequence $HCons\ True\ (HCons\ \text{"b1a"}\ HNil)$ is a correct heterogeneous list with type $HCons\ Bool\ (HCons\ [Char]\ HNil)$. We do not want however that the expression $HCons\ True\ False$ represent a correct heterogeneous list too, since the second $HCons$ argument is not a type-level list. This constraint is encoded by the class $HList$ and its instances, where the constraint is represented by a context condition for the $HCons...$ instance:

```
class  $HList\ l$ 
instance  $HList\ HNil$ 
instance  $HList\ l \Rightarrow HList\ (HCons\ e\ l)$ 
```

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

```
class  $HExtend\ e\ l\ l' \mid e\ l \rightarrow l', l' \rightarrow e\ l$ 
where  $hExtend :: e \rightarrow l \rightarrow l'$ 
```

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

```
instance  $HList\ l \Rightarrow HExtend\ e\ l\ (HCons\ e\ l)$ 
where  $hExtend = HCons$ 
```

The main reason for introducing the class $Extend$ 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

Extensible records (or just “records” from now on) are labeled collections (maps from labels to values, not containing duplicate labels) which can be extended and shrunk. We provide its API. They are modeled in $HList$ as heterogeneous lists of fields, constructed with the data type $Record$:

```
newtype  $Record\ r = Record\ r$ 
```

Thus, an empty record is a $Record$ with an empty heterogeneous list:

```
 $emptyRecord :: Record\ HNil$ 
 $emptyRecord = Record\ HNil$ 
```

A field with label l and value of type v is represented with the following type:

```
newtype  $LVPair\ l\ v = LVPair\ \{value\ LVPair :: v\}$ 
```

Labels are now almost first-class objects, and can be used as type-level values. Note that the label in $LVPair$ is a phantom type (Hinze 2003). We can retrieve the label value using the function $labelLVPair$, which exposes the phantom type parameter:

```
 $labelLVPair :: LVPair\ l\ v \rightarrow l$ 
 $labelLVPair = \perp$ 
```

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:

```
data  $Proxy\ e$ 
 $proxy :: Proxy\ e$ 
 $proxy = \perp$ 
```

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

```
data  $Label1; label1 = proxy :: Proxy\ Label1$ 
data  $Label2; label2 = proxy :: Proxy\ Label2$ 
 $field1 = LVPair\ True :: LVPair\ (Proxy\ Label1)\ Bool$ 
 $field2 = LVPair\ \text{"b1a"} :: LVPair\ (Proxy\ Label2)\ [Char]$ 
 $myR = Record\ (HCons\ field1\ (HCons\ field2\ HNil))$ 
```

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 in instance of the class $HRLabelSet$ to be available when defining extendability for records:

```
instance  $HRLabelSet\ (HCons\ (LVPair\ l\ v)\ r)$ 
 $\Rightarrow HExtend\ (LVPair\ l\ v)\ (Record\ r)$ 
 $(Record\ (HCons\ (LVPair\ l\ v)\ r))$ 
where  $hExtend\ f\ (Record\ r) = Record\ (HCons\ f\ r)$ 
```

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

```
class  $HasField\ l\ r\ v \mid l\ r \rightarrow v$  where
 $hLookupByLabel :: l \rightarrow r \rightarrow v$ 
```

At the type-level it is statically checked that the record r indeed has a field with label l and returns the type-level value v . At value-level the member $hLookupByLabel$ returns the value of type v . So, the following expression returns $True$:

```
 $hLookupByLabel\ label1\ myR$ 
```

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

```
class  $HUpdateAtLabel\ l\ v\ r\ r' \mid l\ v\ r \rightarrow r'$  where
 $hUpdateAtLabel :: l \rightarrow v \rightarrow r \rightarrow r'$ 
```

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

```
 $(*) = hExtend$ 
 $_. = v = LVPair\ v$ 
 $r \# l = hLookupByLabel\ l\ r$ 
```

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

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

So, for example the value myR can now be defined as:

```
 $myR = \{\{label1\}_. True, label2\}_. \text{"b1a"}\}$ 
```

3. Rules

In this subsection we will first show explicitly how attributes and rules are represented. Later we will show how they can be defined by the user in a more implicit way.

An *attribution* is a finite mapping from attribute names to attribute values. We implement attributions as records, where each field represents the name and value of an attribute.

type *Att att val* = *LVPair att val*

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

data *Att_smin*; *smin* = *proxy* :: *Proxy Att_smin*
data *Att_ival*; *ival* = *proxy* :: *Proxy Att_ival*
data *Att_sres*; *sres* = *proxy* :: *Proxy 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 set is described by an instance of the polymorphic data type *Fam*:

data *Fam c p* = *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 for each child a field of type:

type *Chi ch atts* = *LVPair ch 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 define:

data *Ch_tree*; *ch_tree* = *proxy* :: *Proxy (Ch_tree, Tree)*
data *Ch_r*; *ch_r* = *proxy* :: *Proxy (Ch_r, Tree)*
data *Ch_l*; *ch_l* = *proxy* :: *Proxy (Ch_l, Tree)*
data *Ch_i*; *ch_i* = *proxy* :: *Proxy (Ch_i, Int)*

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

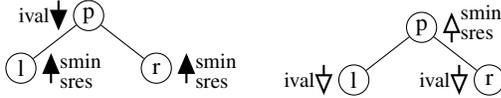


Figure 6. Reprin's input and output families for Node

Families are used to model the input and output attributes of attribute computations. For example, Figure 6 shows the input (black arrows) and output (white arrows) attribute families of the *reprin* 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*):

type *SP* = *Record (HCons (Att (Proxy Att_smin) Int) HCons (Att (Proxy Att_sres) Tree) HNil)*
type *IL* = *Record (HCons (Att (Proxy Att_ival) Int) HNil)*
type *IR* = *Record (HCons (Att (Proxy Att_ival) Int) HNil)*

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

type *IC* = *Record (HCons (Chi (Proxy (Ch_l, Tree) IL) HCons (Chi (Proxy (Ch_r, Tree) IR) HNil)*

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

type *Output_Node* = *Fam SP IC*

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:

type *Rule sc ip ic sp ic' sp'*
= *Fam sc ip* → *Fam ic sp* → *Fam ic' 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:

ext :: *Rule sc ip ic' sp' ic'' sp''* → *Rule sc ip ic sp ic' sp'*
→ *Rule sc ip ic sp ic'' sp''*
(*f 'ext' g*) *input* = *f input.g 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 7 lists all the rule definitions for our running example. The naming convention is such that a rule with name *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 3.

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 7. Rule definitions for *reprin*

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 8. 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 for duplicated attribute definitions.

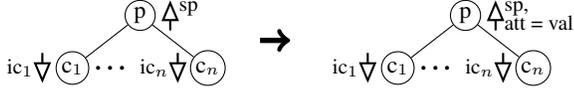


Figure 8. Synthesized attribute definition

Let us take a look at the rule definition $node_smin$ of the attribute $smin$ for the production $Node$ in Figure 7. 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 9. The attribute $smin$ is required (underlined) in the children l and r of the input, and the parent of the output is extended with $smin$.

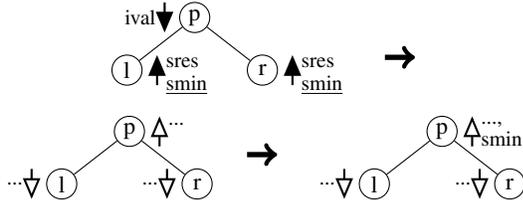


Figure 9. Rule $node_sres$

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 by each child and an attribute $smin$ can be safely added to the current synthesized attribution of the parent:¹

```

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

¹In order to keep the explanation simple we will suppose that min is not overloaded, and takes Int 's as parameter.

The parameter nts takes over the role of the INH declaration in Figure 3. 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 (def att nts vals ic) sp

```

The class Def is defined by induction over the record $vals$ containing the new definitions. The function def 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
    def _ _ _ 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 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 $HBools$ (either $HTrue$ or $HFalse$) which are passed as parameters to $SingleDef$. We are now ready to give the definition for the non-empty case:

```

instance (Defs att nts (Record vs) ic ic'
        , HasLabel (Proxy (lch, t)) ic' mch
        , HMember (Proxy t) nts mnts
        , SingleDef mch mnts att
        (Chi (Proxy (lch, t)) vch)
        ic' ic'')
       => Defs att nts
       (Record (HCons (Chi (Proxy (lch, t)) vch) vs))
       ic ic''

```

where

```

defs att nts ~(Record (HCons pch vs)) ic =
    singledef mch mnts att pch ic'

```

```

where ic' = defs att nts (Record vs) ic
      lch = labelLVPair pch
      mch = hasLabel lch ic'
      mnts = hMember (sndProxy lch) nts

```

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

```

class HBool b => HasLabel l r b | l r -> b
instance HasLabel l r b => HasLabel l (Record r) b
instance (HEq l lp b, HasLabel l r b', HOr b b' b'')
       => HasLabel l (HCons (LVPair lp vp) r) b''
instance HasLabel l HNil HFalse

```

```

hasLabel :: HasLabel l r b => l -> r -> b
hasLabel = ⊥
sndProxy :: Proxy (a, b) -> Proxy b
sndProxy _ = ⊥

```

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.²

```

class SingleDef mch mnts att pv ic ic'
  | mch mnts pv ic -> ic'
  where singledef :: mch -> mnts -> att -> pv -> ic -> ic'
instance (HasField lch ic och
        , HExtend (Att att vch) och och'
        , HUpdateAtLabel lch och' ic ic')
 => SingleDef HTrue HTrue att (Chi lch vch) ic ic'
where singledef _ _ att pch ic =
  hUpdateAtLabel lch (att .=. vch *. och) ic
  where lch = labelLVPair pch
        vch = valueLVPair pch
        och = hLookupByLabel lch ic

```

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

As we have seen an *aspect* is a group of related attributes (one or more), which are defined across multiple productions. We represent aspects as records which contain for each production a rule field.

```

type Prd prd rule = LVPair prd rule

```

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

```

data P_Root; p_Root = proxy :: Proxy P_Root
data P_Node; p_Node = proxy :: Proxy P_Node
data P_Leaf; p_Leaf = proxy :: Proxy P_Leaf

```

We now can define the aspects of *repmin* as records with the rules of Figure 7³.

```

asp_smin = { { p_Leaf .=. leaff_smin
             , p_Node .=. nodef_smin } }
asp_ival = { { p_Root .=. rootf_ival
             , p_Node .=. nodef_ival } }
asp_sres = { { p_Root .=. rootf_sres
             , p_Node .=. nodef_sres
             , p_Leaf .=. leaff_sres } }

```

4.1 Aspects Combination

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

```

class Com r r' r'' | r r' -> r''
  where (⊕) :: r -> r' -> r''

```

²The 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.

³We assume that the monomorphism restriction has been switched off.

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

```

asp_repmin = asp_smin ⊕ asp_ival ⊕ 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:

```

instance Com r (Record HNil) r
  where r ⊕ _ = r
instance (HasLabel lprd r b
        , Com' b (Prd lprd rprd) r r'''
        , Com r''' (Record r') r'')
 => Com r (Record (HCons (Prd lprd rprd) r')) r''
where
  r ⊕ (Record (HCons prd r')) = r''
  where b = hasLabel (labelLVPair prd) r
        r''' = com' b prd r
        r'' = r''' ⊕ (Record r')

```

We use the class *Com'* 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.⁴

```

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''))
 => 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 DGG—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.

⁴This parameter can be avoided by allowing overlapping instances, but we prefer to minimize the number of Haskell extensions we use

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*) together with the initial empty outputs for the children (*ec*). 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 3 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 7 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 a record *ip* containing the inherited attributes of the parent; from this record the value of the attribute *att* which is to be copied to the children can be obtained:

```
copy att nts (Fam _ ip) = defcp att nts ip
defcp :: (Copy att nts vp ic ic', HasField att ip vp)
       => att -> nts -> ip -> (Fam ic sp -> Fam ic' sp)
defcp att nts ip (Fam ic sp) =
  Fam (cpychi att nts (ip # att) 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.

```
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
         , Copy' mnts att nts 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 att nts vp pch *. ics'
  where ics' = cpychi att nts vp (Record ics)
        lch = sndProxy (labelLVPair pch)
        mnts = hMember lch nts
```

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

```
class Copy' mnts att nts vp pch pch' | mnts pch -> pch'
  where
    cpychi' :: mnts -> att -> nts -> 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 att nts vp pch pch where
  cpychi' _ _ _ _ pch = pch
```

If the type of the child belongs to *nts*, the attribution *vch* is extended with the attribute (*Att att vp*).

```
instance HExtend (Att att vp) vch vch'
  => Copy' HTrue att nts vp (Chi lch vch)
  (Chi lch vch') where
  cpychi' _ att nts 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 we find 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.

```
use :: (Use att nts a sc, HExtend (Att att a) sp sp')
     => att -> nts -> (a -> a -> a) -> a
     -> Rule sc ip ic sp ic sp'
```

Using this new combinator the rule *node_smin* of Figure 7 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.

```
chain :: (Chain att nts g sc ip ic ic' lic'
         , HasField att lic' vp
         , HExtend (Att att vp) sp sp')
       => att -> nts -> Rule sc ip ic sp ic' sp'
```

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 10 which is a desugared version of the notation presented in the introduction.

An inherited attribute aspect, like *asp_ival* in Figure 10, can be defined using the function *inhAspect*. It takes as arguments: the

```

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

```

Figure 10. Aspects definition for repmin

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:

```

inhAspect att nts cpys defs
= (attAspect (FnInh att nts) defs)
⊕ (defAspect (FnCpy att nts) cpys)

```

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 10, 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.

```

synAspect att nts op unit uses defs
= (attAspect (FnSyn att) defs)
⊕ (defAspect (FnUse att nts op unit) uses)

```

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

```

chnAspect att nts chns inhdefs syndefs
= (attAspect (FnInh att nt) inhdefs)
⊕ (attAspect (FnSyn att) syndefs)
⊕ (defAspect (FnChn att nts) chns)

```

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:

```

{{ p_Root :=. (λinput →
               syndef sres ((chi input # ch_tree) # sres))
  , p_Leaf :=. (λinput →
               syndef sres (Leaf (par input # ival))) }}

```

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

```

{{ p_Root :=. (λinput → (chi input # ch_tree) # sres)
  , p_Leaf :=. (λinput → Leaf (par input # ival)) }}

```

By mapping the function $((.) \text{ (syndef sres)})$ 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:

```

class AttAspect rdef defs rules | rdef defs → rules
  where attAspect :: rdef → defs → rules
instance (AttAspect rdef (Record defs) rules
         , Apply rdef def rule
         , HExtend (Prd lprd rule) rules rules')
  ⇒ AttAspect rdef
      (Record (HCons (Prd lprd def)
                    defs))
      rules'

```

```

where
  attAspect rdef (Record (HCons def defs)) =
    let lprd = (labelLVPair def)
    in lprd :=. apply rdef (valueLVPair def)
    *. attAspect rdef (Record defs)
instance AttAspect rdef (Record HNil) (Record HNil)
  where attAspect _ _ = emptyRecord

```

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

```

class Apply f a r | f a → r where
  apply :: f → a → r

```

In the case of synthesized attributes we apply $((.) \text{ (syndef att)})$ to values of type $(Fam \text{ sc ip } \rightarrow \text{val})$ in order to construct a rule of type $(Rule \text{ sc ip ic sp ic sp'})$. 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.

```

data FnSyn att = FnSyn att
instance HExtend (LVPair att val) sp sp'
  ⇒ Apply (FnSyn att) (Fam sc ip → val)
      (Rule sc ip ic sp ic sp') where
  apply (FnSyn att) f = syndef att.f

```

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

```

data FnInh att nt = FnInh att nt
instance Def att nts vals ic ic'
  ⇒ Apply (FnInh att nts) (Fam sc ip → vals)
      (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. It iterates over the list of labels, constructing a record with these labels and the rule as value.

```
class DefAspect deff prds rules | deff prds → rules
  where defAspect :: deff → prds → rules
instance DefAspect deff HNil (Record HNil) where
  defAspect _ _ = emptyRecord
instance (Apply deff None deff'
  , DefAspect deff prds rules
  , HExtend (Prd prd deff') rules rules')
⇒ DefAspect deff (HCons prd prds) rules' where
  defAspect deff (HCons prd prds) =
    prd .=. apply deff none *. defAspect deff prds
```

As in `attAspect` we use the class `Apply` to introduce new constraints, but since there is no application in this case, we pass a dummy type `None` as argument.

```
data None; none = ⊥ :: None
```

For inherited attributes we apply the copy rule `copy att nts`:

```
data FnCpy att nts = FnCpy att nts
instance (Copy att nts vp ic ic'
  , HasField att ip vp)
⇒ Apply (FnCpy att nts) None
  (Rule sc ip ic sp ic' sp) where
  apply (FnCpy att nts) _ = copy att nts
```

For example in the case of the aspect `asp_ival`, the application

```
defAspect ival { nt_Tree } { p_Node }
```

generates the default aspect:

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

For synthesized attributes we apply the use rule `use att nt op unit`:

```
data FnUse att nt op unit = FnUse att nt op unit
instance (Use att nts a sc
  , HExtend (LVPair att a) sp sp')
⇒ Apply (FnUse att nts (a → a → a) a) None
  (Rule sc ip ic sp ic' sp') where
  apply (FnUse att nts op unit) _ = use att nt op unit
```

For chained attributes we apply the chain rule `chain att nts`:

```
data FnChn att nt = FnChn att nt
instance (Chain att nts g sc ip ic ic' lic'
  , HasField att lic' vp
  , HExtend (LVPair att vp) sp sp')
⇒ Apply (FnChn att nts) None
  (Rule sc ip ic sp ic' sp') where
  apply (FnChn att nts) _ = chain att nts
```

8. Related Work

There have been several previous attempts at incorporating first-class attribute grammars in lazy functional languages. To 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; in (de Moor et al. 2000b) a whole collection of functions is introduced and the definition of copy, use and chain rules is thus no longer possible. 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 (?) 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 already.

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 our Haskell compiler. 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 3. 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 compiler we have non-terminals with far over 10 different attributes, and gluing all these together or selectively leaving some out turns out to be impossible to do by hand.

In our attribute grammar system 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 out. 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 from 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, especially by incorporating more common attribute grammar patterns. 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.

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>.
- 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.
- 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. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
- 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. ISBN 3-540-67262-1.
- Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13: 229–256, 1980.
- Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4. doi: <http://doi.acm.org/10.1145/1017472.1017488>.
- Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796802004355>.
- 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. doi: <http://doi.acm.org/10.1145/512644.512645>.
- 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. doi: <http://doi.acm.org/10.1145/1159842.1159846>.
- 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.
- Phil Wadler. The Expression Problem. E-mail available online., 1998.