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 type data 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 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). 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 \( \text{sem}_\text{Tree} (\text{asp}_{\text{smín}}) \), passing down the globally minimal value from the root to the leaves as the parameter \( \text{ival} (\text{asp}_{\text{real}}) \), and the construction of the resulting tree as the second component of the result \( \text{asp}_{\text{res}} \).

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 leaves 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:

\[
\text{repmin} = \text{sem}_\text{Root} (\text{asp}_{\text{smín}} \oplus \text{asp}_{\text{ival}} \oplus \text{asp}_{\text{res}})
\]

so we can easily replace the aspect \( \text{asp}_{\text{smín}} \) by \( \text{asp}_{\text{avg}} \):

\[
\text{repavg} = \text{sem}_\text{Root} (\text{asp}_{\text{avg}} \oplus \text{asp}_{\text{ival}} \oplus \text{asp}_{\text{res}})
\]

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
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 Strings 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 accessed, 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

$ (\text{deriveAG } \text{"smin","ival","sres"})$

\[
\begin{align*}
\text{asp\_smin} &= \text{synthesize smin at } \{ \text{Tree} \} \\
\text{use at } \{ \text{Node} \} \\
\text{define at } \text{Leaf} = i \\
\text{asp\_ival} &= \text{inherit ival at } \{ \text{Tree} \} \\
\text{copy at } \{ \text{Node} \} \\
\text{define at } \text{Root.tree} = \text{tree\_smin} \\
\text{asp\_sres} &= \text{synthesize sres at } \{ \text{Root, Tree} \} \\
\text{use at } \{ \text{Node} \} \\
\text{define at } \text{Root.tree} = \text{tree\_sres} \\
\text{Leaf} &= \text{Leaf lhs.ival} \\
\text{asp\_repmin} &= \text{asp\_smin} \oplus \text{asp\_sres} \oplus \text{asp\_ival} \\
\text{repmin } t &= \text{select sres from compute } \text{asp\_repmin } t
\end{align*}
\]

Figure 3. \text{repmin} in our embedded DSL looks like. In Figure 3 we give our definition of the \text{repmin} problem in a lightly sugared notation.

To completely implement the \text{repmin} function the user of our library\footnote{Available as \text{AspectAG} in Hackage.} 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 \text{asp\_repmin};
- composing the aspects into a single large aspect \text{asp\_repmin};
- define the function \text{repmin} that takes a \text{tree}, executes the semantic function for the tree and the aspect \text{asp\_repmin}, and selects the synthesized attribute \text{sres} from the result.

To represent type-level values we define a new type \text{HBool} for type-level Boolean values. The class \text{HBool} represents the type-level type of Booleans. We may read the instance definitions as “the type-level values \text{HTrue} and \text{HFalse} have the type-level type \text{HBool}:

\begin{align*}
\text{class } \text{HBool } x \\
\text{data } \text{HTrue} & = 0 :: \text{HTrue} \\
\text{data } \text{HFalse} & = 1 :: \text{HFalse} \\
\text{instance } \text{HBool } \text{HTrue} \\
\text{instance } \text{HBool } \text{HFalse}
\end{align*}

Since we are only interested in type-level computation, we defined \text{HTrue} and \text{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 \text{HOr} for type-level disjunction:

\[
\begin{align*}
\text{class } (\text{HBool } t, \text{HBool } t', \text{HBool } t'') \\
\Rightarrow \text{HOr } t \ t' \ t'' & | t \ t' \ t'' \\
\text{where } \text{hOr} : & : t \rightarrow t' \rightarrow t''
\end{align*}
\]

The context \text{HList}

\[
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{The DDG for \text{Node}}
\end{figure}
\]
HBool. The functional dependency \( t' \rightarrow t'' \) 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:

```haskell
instance HOr 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 \( HFalse \), respectively. So, the second instance is chosen to select \( hOr \) from and thus \( t'' \) is inferred to be \( HTrue \).

Despite the fact that 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.

This implies that once \( t \) and \( t' \) are \( HTrue \) and \( HFalse \), respectively. So, the second instance is chosen to select \( hOr \) from and thus \( t'' \) is inferred to be \( HTrue \).

If we had defined \( HTrue \) and \( HFalse \) in the following way:

```haskell
data HTrue = HTrue ; hTrue = HTrue :: Htrue
data 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:

```haskell
data HNil = HNil
data HCons e l = HCons e l
```

The sequence \( HCons \ True \ (HCons \ "bla" \ HNil) \) is a correct heterogeneous list with type \( HCons \ Bool \ (HCons \ String \ HNil) \). Since we want to prevent that an expression \( HCons \ True \ False \) 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 this constraint by adding a context condition to the \( HCons \)... instance:

```haskell
class HList l
data HList l ⇒ HList (HCons e l)
```

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

```haskell
class HExtend e l l' | e l → l', l' → e l where hExtend : e l → l'
```

The functional dependency \( e l → l' \) makes that \( HExtend \) 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 \( hExtend \) performs the same computation at the level of values. The instance of \( HExtend \) for heterogeneous lists includes the well-formedness condition:

```haskell
instance HList l ⇒ H Extend e l (HCons e l) where hExtend = HCons
```

The main reason for introducing the class \( HExtend \) 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:

```haskell
newtype Record r = Record r
```

An empty record is a \( Record \) containing an empty heterogeneous list:

```haskell
emptyRecord :: :: Record HNil
emptyRecord = Record HNil
```

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

```haskell
newtype LVPair l v = LVPair (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 \( labelLVPair \), which exposes the phantom type parameter:

```haskell
labelLVPair :: LVPair l v → l
labelLVPair = ⊥
```

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:

```haskell
data Proxy e; proxy = ⊥ :: Proxy e
```

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

```haskell
data Label1; label1 = proxy :: Proxy Label1
data Label2; label2 = proxy :: Proxy Label2
```

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

```haskell
field1 = LVPair True :: LVPair (Proxy Label1) Bool
field2 = LVPair "bla" :: 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 an instance of the class \( HRLabelSet \) to be available when defining extendability for records:

```haskell
instance HRLabelSet (HCons (LVPair l v) r) ⇒ 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:

```haskell
class HasField l r v | l r → v where hLookupByLabel :: l → r → v
```

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 \( hLookupByLabel \) returns the value of type \( v \). So, the following expression returns the string "bla":

```haskell
hLookupByLabel label2 myR
```
The possibility to update an element in a record at a given label position is provided by:

```haskell
class HUpdateAtLabel l v r r' | l v r → r' where
hUpdateAtLabel :: l → v → r → r'
```

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

\[
\text{\(\cdot\)} = hExtend \\
_\cdot =, 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, ..., vn \} for (v₁ \& ... \& vn \&HNil)
- \{[ v₁, ..., vn ]\} for (v₁ \& ... \& vn \& emptyRecord)

So, for example the definition of \text{myR} can now be written as:

\[
\text{myR} = \{[\text{label1} := \text{True}, \text{label2} := \text{"bla"}]\}
\]

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.

\[
\text{type Att att val} = LVPair att val
\]

The labels\(^2\) (attribute names) for the attributes of the example are:

\[
\text{data Att.smin; smin = proxy :: Proxy Att.smin} \\
\text{data Att.iwal; iwal = proxy :: Proxy Att.iwal} \\
\text{data Att.sres; sres = proxy :: Proxy Att.sres}
\]

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 family is represented by an instance of:

\[
\text{data Fam c p} = \text{Fam c p}
\]

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

\[
\text{type Chi ch atts} = LVPair ch atts
\]

where \text{ch} is a label that represents the name of that child and \text{atts} is again a \text{Record} with the fields of type \text{Att} associated with this particular child. In our example the \text{Root} production has a single child \text{Ch.tree} of type \text{Tree}, the \text{Node} production has two children labelled by \text{Ch.l} and \text{Ch.r} of type \text{Tree}, and the \text{Leaf} production has a single child called \text{Ch.i} of type \text{Int}. Thus we generate, using template Haskell:

\[
\text{data Ch.tree; Ch.tree} = \text{proxy :: Proxy (Ch.tree, Tree)} \\
\text{data Ch.r; Ch.r} = \text{proxy :: Proxy (Ch.r, Tree)} \\
\text{data Ch.l; Ch.l} = \text{proxy :: Proxy (Ch.l, Tree)} \\
\text{data Ch.i; Ch.i} = \text{proxy :: Proxy (Ch.i, Int)}
\]

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.

Families are used to model the input and output attributes of attribution 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 \text{Node} production, which are the synthesized attributes of the parent (\text{sp}) and the inherited attributions for the left and right child (\text{il} and \text{ir}): 

\[
\begin{align*}
\text{type SP} & = \text{Record (HCons (Att (Proxy Att.smin) Int) HCons (Att (Proxy Att.sres) Tree) HNil)} \\
\text{type IL} & = \text{Record (HCons (Att (Proxy Att.iwal) Int) HNil)} \\
\text{type IR} & = \text{Record (HCons (Att (Proxy Att.iwal) Int) HNil)}
\end{align*}
\]

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

\[
\begin{align*}
\text{type IC} & = \text{Record (HCons (Chi (Proxy (Ch.l, Tree)) IL) HCons (Chi (Proxy (Ch.r, Tree)) IR) HNil)}
\end{align*}
\]

We now have all the ingredients to define the output family for \text{Node.s}:

\[
\begin{align*}
\text{type Output.Node} & = \text{Fam IC SP}
\end{align*}
\]

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:

\[
\begin{align*}
\text{type Rule sc ip ic sp ic' sp'} & = \text{Fam sc ip → Fam ic sp → Fam ic' sp'}
\end{align*}
\]

Thus, the type \text{Rule} states that a rule takes as input the synthesized attributes of the children \text{sc} and the inherited attributes of the parent \text{ip} and returns a function from the output constructed thus far (inherited attributes of the children \text{ic} and synthesized attributes of the parent \text{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:

\[
\begin{align*}
\text{ext :: Rule sc ip ic' sp' ic'' sp'' → Rule sc ip ic sp ic' sp' → Rule sc ip ic sp ic'' sp''}
\end{align*}
\]

(If \text{ext} \(g\) input = \(f\) input \(g\) input)

3.1 Rule Definition

We now introduce the functions \text{syndef} and \text{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 \text{prod_att} defines the attribute \text{att} for the production \text{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_l)
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 }
node_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.

```plaintext
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.

```
\[
\begin{array}{c}
\text{Figure 7. Synthesized attribute definition} \\
\end{array}
\]

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

```plaintext
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 sp 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.

```plaintext
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 Defs 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.

```plaintext
class Defs att nts 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.

```plaintext
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 HBools (either HTTrue or HFalse) which are passed as parameters to SingleDef. We are now ready to give the definition for the non-empty case:

\[
\text{instance \{Defs att nts (Record vs) ic ic'}
\]

\[
, \text{HasLabel (ProxY (ich, t)) ic'} \text{ mch}
\]

\[
, \text{HMember (ProxY t) nts mnts}
\]

\[
, \text{SingleDef mch mcts att}
\]

\[
(\text{Chi (ProxY (lch, t)) vch})
\]

\[
ic' \text{ ic''})
\]

\[
\Rightarrow \text{Defs att nts}
\]

\[
(\text{Record ((HCons (Chi (ProxY (ich, t)) vch) vs)) ic ic''})
\]

\[
\text{where}
\]

\[
defs \text{ att nts } \sim (\text{Record (HCons pch vs)) ic} =
\]

\[
singledef \text{ mch mcts att pch ic'}
\]

\[
\text{where ic'} = \text{defs att nts (Record vs) ic}
\]

\[
lch = \text{labelLVPair pch}
\]

\[
mch = \text{HasLabel lch ic'}
\]

\[
\text{mnts} = h\text{Member (sndProxY lch) nts}
\]

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

\[
\text{class HBool b } \Rightarrow \text{HasLabel l r b} \mid l \rightarrow b
\]

\[
\text{instance HasLabel l r b } \Rightarrow \text{HasLabel l (Record r) b}
\]

\[
\Rightarrow \text{HasLabel l (HCons (LVPair lp vp) r) b''}
\]

\[
\text{instance HasLabel l HNil HFalse}
\]

\[
\text{hasLabel} :: \text{HasLabel l r b } \Rightarrow l \rightarrow r \rightarrow b
\]

\[
\text{hasLabel} = \bot
\]

\[
\text{sndProxY} :: \text{ProxY (a, b)} \rightarrow \text{ProxY b}
\]

\[
\text{sndProxY} = \bot
\]

We only show the instance with both mch and mnts equal to HTTrue, 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.\(^3\)

\[
\text{class SingleDef mch mnts att pv ic ic'}
\]

\[
| \text{mch mnts pv } ic \rightarrow ic'
\]

\[
\text{where singledef} :: \text{mch } \rightarrow \text{mnts } \rightarrow \text{att } \rightarrow \text{pv } \rightarrow \text{ic } \rightarrow \text{ic'}
\]

\[
\text{instance (HasField lch ic och)}
\]

\[
, \text{HExtend (Att att vch) och och'}
\]

\[
, \text{HUpdateAtLabel lch och'} \mid \text{ic' ic''})
\]

\[
\Rightarrow \text{SingleDef HTTrue HTTrue att (Chi lch vch) ic'}
\]

\[
\text{where singledef } \sim att \text{ pch ic}
\]

\[
= h\text{UpdateAtLabel lch (att } \Rightarrow \text{vch } \ast \text{ och }) \mid \text{ic'}
\]

\[
\text{where lch = labelLVPair pch}
\]

\[
vch = \text{valueLVPair pch}
\]

\[
och = h\text{LookupByLabel lch ic}
\]

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.

\[4\]

### 4. Aspects

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

\[
\text{type Prd prd rule } = \text{LVPair prd rule}
\]

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

\[
data P\_\text{Root}; p\_\text{Root} = \text{proxy :: ProxY P\_Root}
\]

\[
data P\_\text{Node}; p\_\text{Node} = \text{proxy :: ProxY P\_Node}
\]

\[
data P\_\text{Leaf}; p\_\text{Leaf} = \text{proxy :: ProxY P\_Leaf}
\]

We now can define the aspects of repmin as records with the rules of Figure 6.\(^5\)

\[
\text{asp\_smin } = \{ p\_\text{Leaf } := \text{leaf\_smin}
\]

\[
, p\_\text{Node } := \text{node\_smin} \}
\]

\[
\text{asp\_ival } = \{ p\_\text{Root } := \text{root\_ival}
\]

\[
, p\_\text{Node } := \text{node\_ival} \}
\]

\[
\text{asp\_sres } = \{ p\_\text{Root } := \text{root\_sres}
\]

\[
, p\_\text{Node } := \text{node\_sres}
\]

\[
, p\_\text{Leaf } := \text{leaf\_sres} \}
\]

### 4.1 Aspects Combination

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

\[
\text{class Com r r' r'' } | r \rightarrow r' \rightarrow r''
\]

\[
\text{where ( } \oplus \text{ ) } : r \rightarrow r' \rightarrow r''
\]

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

\[
\text{asp\_repmn } = \text{asp\_smin } \oplus \text{asp\_ival } \oplus \text{asp\_sres}
\]

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 (\(\oplus\)) 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:

\[
\text{instance Com r (Record HNil) r}
\]

\[
\text{where r } \oplus r' = r
\]

\[
\text{instance (HasLabel lprd r b)
\]

\[
, \text{ComSingle b (Prd lprd rprd) r r''}
\]

\[
, \text{Com r'' (Record r' r'')} \Rightarrow \text{Com r (Record (HCons (Prd lprd rprd) r'') r'')}
\]

\[
\text{where}
\]

\[
r \oplus (\text{Record (HCons prd r')} = r''
\]

\[
\text{where b = hasLabel (labelLVPair prd) r}
\]

\[
r'' = \text{comsingle b prd r}
\]

\[
r' = r'' \oplus (\text{Record r'})
\]

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.\(^6\)

---

3 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.

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

5 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 \(lp\), we update its value by composing the rules.

\[
\text{instance } (\text{HasField } lp r (\text{Rule } sc ic ip ic sp ic'' sp'')) \Rightarrow \text{ComSingle } HT\text{rue } (\text{Prd } lp r (\text{Rule } sc ic ip ic sp ic'' sp''))
\]

\[
\text{where } \text{comsingle } _f r = \text{hUpdateAtLabel } n ((r \neq n) \cdot \text{ext} v) r
\]

\[
\text{where } n = \text{labelLVPair } f
\]

\[
v = \text{valueLVPair } f
\]

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

\[
\text{instance } \text{ComSingle } H\text{False } f (\text{Record } r) (\text{Record } (\text{HCons } f r))
\]

\[
\text{where } \text{comsingle } _f (\text{Record } r) = \text{Record } (\text{HCons } f r)
\]

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 \(kn\), 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 \(\text{sem}_{<nt>}\). 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 \(kn\), which stitches everything together:

\[
\text{sem}_\text{Root } \text{asp } (\text{Root } t) = \text{knit } (\text{asp } \# p_\text{Root}) \{ \text{ch}\_\text{tree }=. \text{sem}_\text{Tree } \text{asp } t \} \]

\[
\text{sem}_\text{Tree } \text{asp } (\text{Node } l r) = \text{knit } (\text{asp } \# p_\text{Node}) \{ \text{ch}\_l\. \text{sem}_\text{Tree } \text{asp } l , \text{ch}\_r\. \text{sem}_\text{Tree } \text{asp } r \}
\]

\[
\text{sem}_\text{Tree } \text{asp } (\text{Leaf } i) = \text{knit } (\text{asp } \# p_\text{Leaf}) \{ \text{ch}\_i\. \text{sem}_\text{Lit } i \}
\]

\[
\text{sem}_\text{Lit } e (\text{Record } H\text{Nil}) = e
\]

Since this code is completely generic we provide a Template Haskell function \(\text{deriveAG}\) which automatically generates the functions such as \(\text{sem}_\text{Root}\) and \(\text{sem}_\text{Tree}\), together with the labels for the non-terminals and labels for referring to children. Thus, to completely implement the \(\text{repmin}\) function we need to undertake the following steps:

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

\[
\$ (\text{deriveAG } "\text{Root})
\]

- Define and compose the aspects as shown in the previous sections, resulting in \(\text{asp}_\text{repmin}\).

- Define the function \(\text{repmin}\) that takes a tree, executes the semantic function for the tree and the aspect \(\text{asp}_\text{repmin}\), and selects the synthesized attribute \(sres\) from the result.

\[
\text{repmin } \text{tree} = \text{sem}_\text{Root } \text{asp}_\text{repmin } (\text{Root } \text{tree}) () \# \text{sres}
\]

5.1 The Knit Function

As said before the function \(\text{knit}\) 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 \(sc\).

\[
\text{knit } :: (\text{Empties } fc ec, \text{Kn } fc ec sc) \Rightarrow \text{Rule } sc ip ic sp ic' sp''
\]

\[
\text{knit } \text{rule } fc ip =
\]

\[
\text{let } ec = \text{empties } fc,
\]

\[
\text{(Fam } ic sp) = \text{rule } (\text{Fam } sc ip) \quad \text{(Fam } ec \text{emptyRecord)}
\]

\[
\text{sc } = \text{kn } fc ic \quad \text{in } sp
\]

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

\[
\text{class } \text{Kn } fc ic sc | fc → ic → sc \text{ where }
\]

\[
\text{kn } :: fc → ic → sc
\]

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

\[
\text{instance } \text{Kn } fc ic sc \Rightarrow \text{Kn } (\text{Record } fc) (\text{Record } ic) (\text{Record } sc) \text{ where }
\]

\[
\text{kn } (\text{Record } fc) (\text{Record } ic) = \text{Record } \$ \text{kn } fc ic
\]

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

\[
\text{instance } \text{Kn } H\text{Nil } H\text{Nil } H\text{Nil } \text{where }
\]

\[
\text{kn } _\text{} = H\text{Nil}
\]

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

\[
\text{instance } \text{Kn } fc er sc
\]

\[
\Rightarrow \text{Kn } (\text{HCons } (\text{Chi } lch (\text{ich } → \text{sch})) fc) er
\]

\[
(\text{HCons } (\text{Chi } lch ich) \quad icr)
\]

\[
(\text{HCons } (\text{Chi } lch sch) \quad scr)
\]

\[
\text{where }
\]

\[
\text{kn } \sim (\text{HCons } pfch fc er) \sim (\text{HCons } pich ich er) =
\]

\[
\text{let } er = \text{kn } fc er
\]

\[
lch = \text{labelLVPair } pfch
\]

\[
fch = \text{valueLVPair } pfch
\]

\[
ich = \text{valueLVPair } pich
\]

\[
\text{in } H\text{Cons } (\text{newLVPair } lch (fch ich)) er
\]
In the same way that in Figure 2 we see that the rules for the class Copy iterate 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 mnts of non-terminals and the attribute is not already defined for this child.

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 node呸 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 mnts 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 mnts vp ic ic', HasField att vp) => att -> mnts -> Rule ic ic sp sp'
```

Thus, for example, the rule node呸 in Figure 6 can now be written as:

```
node呸 input = copy흕 { 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 indef, 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 mnts (Fam _ vp) = defcp att mnts (vp # att)
defcp :: Copy att mnts vp ic ic' => att -> mnts -> vp -> (Fam ic sp -> Fam ic' sp)
defcp att mnts vp (Fam ic sp) = Fam (cpychi att mnts 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 mnts of non-terminals and the attribute is not already defined for this child.

```
class Copy att mnts vp ic ic' | ic -> ic' where
cpychi :: att -> mnts -> vp -> ic -> ic'
```

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

instance (Copy att mnts vp (Record ics) ics') hasLabel att mch mch' hasLabel att mch mch'
where ics' = cpychi att mnts vp (Record ics)
  lch = sndProxy (labelLVPair pch)
  vch = valueLVPair pch
  mnts = hMember lch mnts
  mch = hasLabel att vch

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

```
class Copy' mnts mchch att vp pch pch' | mnts mchch pch -> pch'
```

instance Copy' HFalse mchch att vp pch pch' hasLabel att mchch pch

```
where cpychi' :: mnts -> mchch -> 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' HTrue HTrue att vp pch pch where cpychi' _ _ _ _ pch = pch
```

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

```
instance Copy' HExtend pch 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)
  1 (Chi lch vch') where
cpychi' _ _ att vp pch = lch ._. (att ._. vp ._. vch)
```

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.

\[
\text{use} :: \text{Use att nts a sc, HEextend (Att att a) sp sp'} \\
\Rightarrow \text{att} \rightarrow \text{nts} \rightarrow (a \rightarrow a \rightarrow a) \rightarrow a \\
\rightarrow \text{Rule sc ip ic sp ic sp}'
\]

Using this new combinator the rule \text{node.smin} of Figure 6 becomes:

\[
\text{node.smin} = \text{use smin \{ nt, Tree \} min 0}
\]

A third common pattern corresponds to the use of the \text{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.

\[
\text{chain} :: (\text{Chain att nts val sc ic sp ic'} sp') \\
, \text{HasField att ip val} \\
\Rightarrow \text{att} \rightarrow \text{nts} \rightarrow \text{Rule sc ip ic sp ic'} sp'
\]

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 \text{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 \text{asp.val} in Figure 9, can be defined using the function \text{inhAspect}. It takes as arguments: the name of the attribute \text{att}, the list \text{nts} of non-terminals where the attribute is defined, the list \text{cpsys} of productions where the copy rule has to be applied, and a record \text{defs} containing the explicit definitions for some productions:

\[
\text{inhAspect att nts cpsys defs} \\
= (\text{defAspect (FnCpy att nts)} \text{cpsys}) \\
\oplus (\text{attAspect (FnInh att nts)} \text{defs})
\]

The function \text{attAspect} generates an attribute aspect given the explicit definitions, whereas \text{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 \text{attAspect} and \text{defAspect} are implemented.

A synthesized attribute aspect, like \text{asp.smin} and \text{asp.sres} in Figure 9, can be defined using \text{synAspect}. Here the rule applied is the use rule, which takes \text{op} as the monoidal operator and \text{unit} as the unit value.

\[
\text{synAspect att nts op unit uses defs} \\
= (\text{defAspect (FnUse att nts op unit)} \text{uses}) \\
\oplus (\text{attAspect (FnSyn att)} \text{defs})
\]

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

\[
\text{chnAspect att nts chns inhdefs syndefs} \\
= (\text{defAspect (FnChn att nts)} \text{chns}) \\
\oplus (\text{attAspect (FnInh att nts)} \text{inhdefs}) \\
\oplus (\text{attAspect (FnSyn att)} \text{syndefs})
\]

7.1 Attribute Aspects

Consider the explicit definitions of the aspect \text{asp.sres}. The idea is that, when declaring the explicit definitions, instead of completely writing the rules, like:

\[
\text{syndef sres} = (\text{defAspect (FnChn att nts)} \text{chns}) \\
\oplus (\text{attAspect (FnInh att nts)} \text{inhdefs}) \\
\oplus (\text{attAspect (FnSyn att)} \text{syndefs})
\]

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

\[
\text{syndef sres} = (\text{defAspect (FnChn att nts)} \text{chns}) \\
\oplus (\text{attAspect (FnInh att nts)} \text{inhdefs}) \\
\oplus (\text{attAspect (FnSyn att)} \text{syndefs})
\]

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

\[
\text{attAspect rdef rules rdef rules rules} \\
\text{where attAspect :: rdef \rightarrow rdef \rightarrow rules}
\]

\[
\text{instance (AttAspect rdef (Record defs)) rules} \\
, \text{Apply rdef rule} \\
, \text{HEextend (Pred lprd rule) rules rules'} \\
\Rightarrow \text{AttAspect rdef} \\
(\text{Record (HCons (Pred lprd def) \text{defs})}) \\
\text{rules'}
\]

where

\[
\text{attAspect rdef (Record (HCons def defs)) =} \\
\text{let lprd = (labelLVPair def) \\
in lprd \Rightarrow . apply rdef (valueLVPair def)} \\
\times \text{attAspect rdef (Record defs)}
\]

\[
\text{instance AttAspect rdef (Record HNil) (Record HNil)} \\
\text{where attAspect \_ \_ = emptyRecord}
\]

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

\[
\text{class Apply f a r | f a \rightarrow r where} \\
\text{apply :: f a \rightarrow r}
\]

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

\[
\text{data FnSyn att = FnSyn att} \\
\text{instance HEextend (LVPair att val) sp sp'} \\
\Rightarrow \text{Apply (FnSyn att)) (Fam sc ip \rightarrow val)} \\
(\text{Rule sc ip ic sp ic sp'}) \text{where} \\
\text{apply (FnSyn att) f = syndef att.f}
\]

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

\[
\text{data FnInh att nt = FnInh att nt} \\
\text{instance Dsfig att nts vals ic ic'} \\
\Rightarrow \text{Apply (FnInh att nts) (Fam sc ip \rightarrow vals)}
\]
Aspects definition for repmin

\[ \text{asp}_{\text{smin}} = \text{synAspect} \text{ smin} \{ \text{nt}_\text{Tree} \} \]
\[ \text{def Aspect} \text{ smin} \{ \text{nt}_\text{Tree} \} \]
\[ \text{min 0} \{ \text{p}_\text{Node} \} \]
\[ \{\{ \text{p}_\text{Leaf} =. (\lambda (\text{Fam} \text{ chi}) \rightarrow \text{chi} \# \text{ch}_i) \} \} \]

\[ \text{asp}_{\text{ival}} = \text{inhAspect} \text{ ival} \{ \text{nt}_\text{Tree} \} \]
\[ \text{def Aspect} \text{ ival} \{ \text{nt}_\text{Tree} \} \]
\[ \{ \text{p}_\text{Node} \} \]
\[ \{\{ \text{p}_\text{Root} =. (\lambda (\text{Fam} \text{ chi}) \rightarrow \{ \{ \text{ch}_\text{tree} =. (\text{chi} \# \text{ch}_\text{tree}) \# \text{smin} \} \} \} \} \]

\[ \text{asp}_{\text{sres}} = \text{synAspect} \text{ sres} \{ \text{nt}_\text{Root}, \text{nt}_\text{Tree} \} \]
\[ \text{Node} \{ \text{Leaf} 0 \} \{ \text{p}_\text{Node} \} \]
\[ \{\{ \text{p}_\text{Root} =. (\lambda (\text{Fam} \text{ chi}) \rightarrow (\text{chi} \# \text{ch}_\text{tree}) \# \text{sres}) \}
\]
\[ , \text{p}_\text{Leaf} =. (\lambda (\text{Fam} \text{ par}) \rightarrow \text{Leaf} (\text{par} \# \text{ival})) \} \]

\[ \text{Rule sc ip ic sp ic'} \text{ sp} \) \]
\[ \text{apply} (\text{FnInh} \text{ att nts}) f = \text{inhdef att nts.f} \]

7.2 Default Aspects

The function \text{defAspect} is used to construct an aspect given a rule and a list of production labels.

\[ \text{class Def Aspect deff prds rules} \mid \text{def Aspect} \rightarrow \text{rules} \]
\[ \text{where} \text{def Aspect} :: \text{deff} \rightarrow \text{prds} \rightarrow \text{rules} \]

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

\[ \text{data FnCpy att nts} = \text{FnCpy att nts} \]
\[ \text{data FnUse att nt op unit} = \text{FnUse att nt op unit} \]
\[ \text{data FnChn att nt} = \text{FnChn att nt} \]

Thus, for example in the case of the aspect \text{asp}_{\text{ival}}, the application:

\[ \text{def Aspect} \{ \text{FnCpy ival} \{ \text{nt}_\text{Tree} \} \} \{ \text{p}_\text{Node} \} \]

generates the default aspect:

\[ \{\{ \text{p}_\text{Node} =. \text{copy ival} \{ \text{nt}_\text{Tree} \} \} \} \]

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 (usage 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


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


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.


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