

# Attribute Grammar Macros

Marcos Viera<sup>1</sup> and Doaitse Swierstra<sup>2</sup>

<sup>1</sup> Instituto de Computación, Universidad de la República  
Montevideo, Uruguay  
`mviera@fing.edu.uy`

<sup>2</sup> Department of Computer Science, Utrecht University  
Utrecht, The Netherlands  
`doaitse@cs.uu.nl`

**Abstract.** Having extensible languages is appealing, but raises the question of how to construct extensible compilers and how to compose compilers out of a collection of pre-compiled components.

Being able to deal with attribute grammar fragments as first-class values makes it possible to describe semantics in a compositional way; this leads naturally to a plug-in architecture, in which a core compiler can be constructed as a (collection of) pre-compiled component(s), and to which extra components can safely be added as need arises.

We extend *AspectAG*, a Haskell library for building strongly typed first-class attribute grammars, with a set of combinators that make it easy to describe semantics in terms of already existing semantics in a macro-like style, just as syntax macros extend the syntax of a language. We also show how existing semantics can be redefined, thus adapting some aspects from the behavior defined by the macros.

## 1 Introduction

Since the introduction of the very first programming languages, and the invention of grammatical formalisms for describing them, people have investigated how an initial language definition can be extended by someone else than the original language designer by providing separate language-definition fragments.

The simplest approach starts from the *text* which describes a compiler for the base language. Just before the compiler is compiled, several extra ingredients may be added textually. In this way we get great flexibility and there is virtually no limit to the things we may add. The Utrecht Haskell Compiler [5] has shown the effectiveness of this approach by composing a large number of attribute grammar fragments textually into a complete compiler description. This approach however is not very practical when defining relatively small language extensions; we do not want an individual user to have to generate a completely new compiler for each small extension. Another problematic aspect of this approach is that, by making the complete text of the compiler available for modification or extension, we also lose important safety guarantees provided by e.g. the type system; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

So the question arises how we can reach the effect of textual composition, but without opening up the whole compiler source. The most commonly found approach is to introduce so-called *syntax macros* [8], which enable the programmer to add *syntactic sugar* to a language by defining new notation *in terms of already existing syntax*.

In this paper we will focus on how to provide such mechanisms *at the semantic level* [9] too. As a running example we take a minimal expression language described by the grammar:

$$\begin{aligned} \text{expr} &\rightarrow \text{"let" var "=" expr "in" expr} \mid \text{term "+" expr} \mid \text{term} \\ \text{term} &\rightarrow \text{factor "*" term} \mid \text{factor} \\ \text{factor} &\rightarrow \text{int} \mid \text{var} \end{aligned}$$

with the following abstract syntax (as a Haskell data type):

```
data Root = Root { expr :: Expr }
data Expr = Cst { cv :: Int } | Var { vnm :: String }
          | Mul { me1 :: Expr, me2 :: Expr }
          | Add { ae1 :: Expr, ae2 :: Expr }
          | Let { lnm :: String, val :: Expr, body :: Expr }
```

Suppose we want to extend the language with one extra production for defining the square of a value. A syntax macro aware compiler might accept definitions of the form  $\text{square } (se :: \text{Expr}) \Rightarrow \text{Mul } se \ se$ , translating the new syntax into the existing abstract syntax.

Although this approach may be very effective and seems attractive, such transformational programming [3] has its shortcomings too; as a consequence of mapping the new constructs onto existing constructs and performing any further processing such as type checking on this simpler, but often more detailed program representation, feedback from later stages of the compiler is given in terms of the intermediate program representations in which the original program structure is often hard to recognise. For example, if we do not change the pretty printing phase of the compiler, the expression  $\text{square } 2$  will be printed as  $2 * 2$ . Hence the implementation details shine through, and the produced error messages can be confusing or even incomprehensible. Similar problems show up when defining embedded domain specific languages: the error messages from the type system are typically given in terms of the underlying representation [6].

In a previous paper [17] we introduced AspectAG<sup>3</sup>, a Haskell library of first-class attribute grammars, which can be used to implement a language semantics and its extensions in a safe way, i.e. by constructing a core compiler as a (collection of) pre-compiled component(s), to which extra components can safely be added at will. In this paper we show how we can define the semantics of the right hand side in terms of existing semantics, in the form of *attribute grammar macros*.

<sup>3</sup> <http://hackage.haskell.org/package/AspectAG>

We also show how, by using first class attribute grammars, the already defined semantics can easily be *redefined* at the places where it makes a difference, e.g. in pretty printing and generating error messages.

The functionality provided by the combination of attribute grammar macros and redefinition is similar to the *forwarding attributes* [15] technique for higher-order attribute grammars, implemented in the Silver AG system [16]. We however implement our proposal as a set of combinators embedded in Haskell, such that the correctness of the composite system is checked by the Haskell type checker.

In Section 2 we give a top-level overview of our approach. In Section 3 we describe our approach to first-class attribute grammars, and in Section 4 we show how to define semantic macros and how to redefine attributes. We close by presenting our conclusions and future work.

## 2 Attribute Grammar Combinators

Before delving into the technical details, we show in this section how the semantics of our running example language and some simple extensions can be implemented using our approach. We have chosen our example to be very simple, in order to help the understanding of the technique. For a more involved example, including an implementation of the Oberon-0 language [18] using macros to represent the **FOR** and **CASE** statements in terms of a core sub-language, we refer to the web page of the AspectAG project<sup>4</sup>.

The semantics are defined by two aspects: pretty printing, realized by a synthesized attribute *spp*, which holds a pretty printed document, and expression evaluation, realized by two attributes: a synthesized *sval* of type *Int*, which holds the result of an expression, and an inherited *ienv* which holds the environment ( $[(String, Int)]$ ) in which an expression is to be evaluated. We show how the attributes are directly definable in Haskell using the functions *syndefM* and *inhdefM* from the AspectAG library, which define a single synthesized or inherited attribute respectively. Figure 1 lists some of the rule definitions of the semantics of our example. In our naming convention a rule with name *attProd* defines the attribute *att* for the production *Prod*. The rule *sppAdd* for the attribute *spp* of the production *Add* looks for its children attributions and binds them ( $e_i \leftarrow at\ ch\_ae_i$ ) and then combines the pretty printed children  $e_i \# spp$  with the string "+" using the pretty printing combinator ( $>\#\<$ ) for horizontal (beside) composition, from the *uulib*<sup>5</sup> library. The rule *ienvLet* specifies that the *ienv* value coming from the parent (*lhs* stands for "left-hand side") is copied to the *ienv* position of the child *val*; the *ienv* attribute of the *body* is this environment extended with a pair composed of the name (*lnm*) associated with the first child and the value (the *sval* attribute) of the second child.

In Figure 2 we show for each production of the example how we combine the various aspects introduced by the attributes using the function *ext*.

<sup>4</sup> <http://www.cs.uu.nl/wiki/bin/view/Center/AspectAG>

<sup>5</sup> <http://hackage.haskell.org/package/uulib>

```

    -- Pretty-Printing
    sppRoot = syndefM spp $ liftM (#spp) (at ch_expr)
    ...
    sppAdd  = syndefM spp $ do e1 ← at ch_ae1
                              e2 ← at ch_ae2
                              return $ e1 # spp >#< "+" >#< e2 # spp

    ...

    -- Environment
    ienvRoot = inhdefM ienv { nt_Expr } $
              do return {{ ch_expr .=. ([] :: [(String, Int)]) }}
    ...
    ienvLet  = inhdefM ienv { nt_Expr } $
              do lnm ← at ch_lnm
                 val ← at ch_val
                 lhs ← at lhs
                 return {{ ch_val .=. lhs # ienv
                           , ch_body .=. (lnm, val # sval) : lhs # ienv }}

    -- Value
    svalRoot = syndefM sval $ liftM (#sval) (at ch_expr)
    ...
    svalVar  = syndefM sval $ do vnm ← at ch_vnm
                                 lhs  ← at lhs
                                 return $ fromJust (lookup vnm (lhs # ienv))
    ...

```

**Fig. 1.** Fragments of the specification of the example’s semantics using the AspectAG library

The semantics we associate with an abstract syntax tree is a function which maps the inherited attributes of the root node to its synthesized attributes. So for each production that may be applied at the root node of the tree we have to construct a function that takes the semantics of its children and uses these to construct the semantics of the complete tree. We will refer to such functions as *semantic functions*. The hard work is done by the function *knit*, that “ties the knot”, combining the attribute computations (i.e. the data flow at the node) with the semantics of the children trees (describing the flow of data from their inherited to their synthesized attributes) into the semantic function for the parent. The following code defines the semantic functions of the production *Add*:

$$semExpr\_Add\ sae1\ sae2 = knit\ aspAdd\ \{\{ ch\_ae1\ .=. sae1, ch\_ae2\ .=. sae2\}\}$$

where the function *knit* is applied to the combined attributes for the production.

The resulting semantic functions can be associated with the concrete syntax by using parser combinators [14] in an applicative style:

```

aspRoot = sppRoot 'ext' svalRoot 'ext' ienvRoot
aspCst  = sppCst  'ext' svalCst
aspVar  = sppVar  'ext' svalVar
aspMul  = sppMul  'ext' svalMul 'ext' ienvMul
aspAdd  = sppAdd  'ext' svalAdd 'ext' ienvAdd
aspLet  = sppLet  'ext' svalLet  'ext' ienvLet

```

**Fig. 2.** Composition of the semantics

```

pExpr  = semExpr_Let  <$ pKeyw "let" <*> pString
          <*> pKeyw "=" <*> pExpr
          <*> pKeyw "in" <*> pExpr
          <|> semExpr_Add <$> pTerm <*> pKeyw "+" <*> pExpr <|> pTerm
pTerm  = semExpr_Mul <$> pFactor <*> pKeyw "*" <*> pTerm <|> pFactor
pFactor = semExpr_Cst <$> pInt <|> semExpr_Var <$> pString

```

Thus far we have described a methodology to define the static semantics of a language. The goal of this paper is to show how we can define new productions by combining existing productions, while probably updating some of the aspects. We want to express the semantics of new productions *in terms of already existing semantics* and *by adapting parts of the semantics* resulting from such a composition.

To show our approach we will extend the language of our example with some extra productions; one for defining the square of a value, one for defining the sum of the squares of two values, and one for doubling a value:

```

expr → ... | "square" expr | "pyth" expr expr | "double" expr

```

In the rest of this section we define the semantic functions *semExpr\_Sq*, *semExpr\_Pyth* and *semExpr\_Double*, of the new productions, in a macro style, although providing specific definitions for the pretty-printing attributes. Thus, if the expressions' parser is extended with these new productions:

```

pExpr = ... <|> semExpr_Sq <$ pKeyw "square" <*> pExpr
          <|> semExpr_Pyth <$ pKeyw "pyth" <*> pExpr <*> pExpr
          <|> semExpr_Double <$ pKeyw "double" <*> pExpr

```

the semantic action associated to parse, for example, "square 2" returns the value 2 for the attribute *sval* and "square 2" for *spp*.

Thus far, when extending the example language with a *square* production, we would have to define its semantics from scratch, i.e we had to define all its attributes in the same way we did for the original language. Thus, if the semantics of a language are defined by about twenty attributes<sup>6</sup> (to perform pretty-printing, name binding, type checking, optimizations, code generation,

<sup>6</sup> As is the case in the UHC Haskell compiler.

etc.), a definition of all these twenty attributes has to be provided. To avoid this, we introduce *attribute grammar macros* in Figure 3 to define the extensions of the example.

The square of a value is the multiplication of this value by itself. Thus, the semantics of multiplication can be used as a basis, by passing to it the semantics of the only child ( $ch\_se$ ) of the square production both as  $ch\_me_1$  and  $ch\_me_2$ . We do so in the definition of  $aspSq$  in Figure 3; we declare an attribute grammar macro based on the attribute computations for the production  $Mul$ , defined in  $aspMul$ , with its children ( $ch\_me_1$  and  $ch\_me_2$ ) mapped to the new child  $ch\_se$ .

$  \begin{aligned}  aspSq &= agMacro(aspMul, ch\_me_1 \hookrightarrow ch\_se \\  &\quad \langle.\rangle ch\_me_2 \hookrightarrow ch\_se) \\  aspPyth &= agMacro(aspAdd, ch\_ae_1 \Rightarrow (aspSq, ch\_se \hookrightarrow ch\_pe_1) \\  &\quad \langle.\rangle ch\_ae_2 \Rightarrow (aspSq, ch\_se \hookrightarrow ch\_pe_2)) \\  aspDouble &= agMacro(aspMul, ch\_me_1 \Rightarrow (aspCst, ch\_cv \rightsquigarrow 2) \\  &\quad \langle.\rangle ch\_me_2 \hookrightarrow ch\_de)  \end{aligned}  $
---

**Fig. 3.** Language Extension

Attribute macros can map children to other macros, and so on. For example, in the definition of  $aspPyth$  (sum of the squares of  $ch\_pe_1$  and  $ch\_pe_2$ ) the children are mapped to macros based on the semantics of square ( $aspSq$ ).

When defining a macro based on the semantics of a production which has literal children, these children can be mapped to literals. In the definition of  $aspDouble$  the child  $ch\_me_1$  of the multiplication is mapped to a constant, which is mapped to the literal 2.

In some cases we may want to introduce a specialized behavior for some specific attributes of an aspect defined by a macro. For example, the pretty printing attribute  $spp$  of the macros of Figure 3 currently is expressed in terms of the base rule. Thus when pretty printing *square*  $x$ , instead  $x * x$  will be shown. Fortunately it turns out to be very easy to overwrite the definition of some specific attribute instead of adding a new one. This is implemented by the functions  $synmodM$  and  $inhmodM$ .

In Figure 4 we show how the pretty printing attributes of the language extensions we defined in Figure 3 can be redefined to reflect their original appearance in the input program:

### 3 AspectAG

In this section we describe AspectAG, a library for defining first-class attribute grammars. The key technique underlying our embedded approach lies in using the HList library [7] for typed heterogeneous collections (extensible polymorphic

```

sppSq      = synmodM spp $ do de ← at ch_de
              return $ "square" >#< de # spp

aspSq'     = sppSq `ext` aspSq

sppPyth    = synmodM spp $ do e1 ← at ch-pe1
              e2 ← at ch-pe2
              return $ "pyth" >#< e1 # spp >#< e2 # spp

aspPyth'   = sppPyth `ext` aspPyth

sppDouble  = synmodM spp $ do de ← at ch_de
              return $ "double" >#< de # spp

aspDouble' = sppDouble `ext` aspDouble

```

**Fig. 4.** Redefinition of the *spp* attribute

records) for representing collections of attributes, and expressing the AG well-formedness conditions by type-level predicates (i.e., type-class constraints), thus mimicking dependently typed programming techniques in Haskell [10].

Heterogeneous lists are constructed using the functions  $(.*)$  and  $hNil$ , modeling the structure of a normal list both at the value and the type level. An *extensible record* is an heterogeneous list of uniquely labeled fields marked with the type *Record*. A field  $(l \text{ .} v)$  relates a (first-class) label  $l$  with the value  $v$ . Extensible records can be constructed with the functions  $(.*)$  and  $emptyRecord$ ; where  $(.*)$  is overloaded to not only extend the list both at type and value level, but also to impose by (type class) constraints that elements in a record are uniquely labeled. In order to keep our programs readable we will use the some syntactic sugar to denote lists and records in the rest of the paper:

- $\{ v_1, \dots, v_n \}$  for  $(v_1 \text{ .} * \dots \text{ .} * v_n \text{ .} * hNil)$
- $\{ \{ f_1, \dots, f_n \} \}$  for  $(f_1 \text{ .} * \dots \text{ .} * f_n \text{ .} * emptyRecord)$

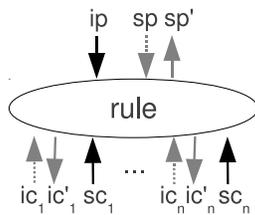
Thus, if  $label_1$  and  $label_2$  are labels, the following is the definition of a record ( $myR$ ) with the elements *True* and "bla":

$$myR = \{ \{ label_1 \text{ .} True, label_2 \text{ .} "bla" \} \}$$

The operator  $(\#)$  is used to retrieve the value part corresponding to a specific label from a record, statically enforcing that the record indeed has a field with this label. The expression  $(myR \# label_2)$  returns the string "bla", while, given a label  $label_3$ , the expression  $(myR \# label_3)$  does not compile.

### 3.1 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 by a *Record*, in which each field represents the name and value of an attribute.



**Fig. 5.** Rule: black arrows represent input and gray arrows represent output; dotted gray arrows represent the already constructed output which can be used to compute further output elements (hence the direction of the arrow)

When inspecting what happens at a production (a node of the abstract syntax tree) we see that information flows from the inherited attribute of the parent ( $ip$ ) and the synthesized attributes of the children ( $sc$ ) to the synthesized attributes ( $sp$ ) of the parent and the inherited attributes of the children ( $ic$ ). Henceforth the attributes  $ip$  and  $sc$  together are called *input family* while the attributes  $sp$  and  $ic$  are called *output family*, both represented by:

**data**  $Fam\ children\ parent = Fam\ children\ parent$

A  $Fam$  contains a single attribution for the parent and a collection of attributions for the children. Hence the type  $parent$  will always be a  $Record$  with fields labeled by attribute names; the type of  $children$  is a  $Record$  with fields labeled by children names and attributions ( $Records$ ) as values. The labels of the children can be defined out of the abstract syntax using the Template Haskell function  $deriveAG$ . For our example, the call  $\$(deriveAG\ "Root)$  generates the labels  $ch\_expr$ ,  $ch\_cv$ ,  $ch\_vnm$ ,  $ch\_me_1$ ,  $ch\_me_2$ ,  $ch\_ae_1$ ,  $ch\_ae_2$ ,  $ch\_lnm$ ,  $ch\_val$  and  $ch\_body$ .

Attributes are defined by *rules* [4], where a rule is a mapping from an input family (the inherited attributes of the parent and the synthesized attributes of the children) to a function which extends the output family (the inherited attributes of the children and the synthesized attributes of the parent) with the new elements defined by this rule:

**type**  $Rule\ sc\ ip\ ic\ sp\ ic'\ sp' = Fam\ sc\ ip \rightarrow (Fam\ ic\ sp \rightarrow Fam\ ic'\ sp')$

Figure 5 shows a graphic representation of a rule; each rule describes a node of a data flow graph which has an underlying tree-shaped structure induced by the abstract syntax tree at hand.

**Rule Definition** The functions  $syndefM$  and  $inhdefM$  are versions of  $syndef$  and  $inhdef$ , that use a  $Reader$  monad to make definitions look somewhat “prettier”.

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 for the parent as constructed thus far ( $sp$ ):

$$\text{syndef att val (Fam ic sp) = Fam ic (att .=. val .* sp)}$$

$$\text{syndefM att mval inpFam = syndef att (runReader mval inpFam)}$$

Let us take a look at how the rule definition *sppAdd* of the attribute *spp* for the production *Add* is defined using *syndef* instead of *syndefM*:

$$\begin{aligned} \text{sppAdd (Fam sc ip)} \\ = \text{syndef spp } \$ ((\text{sc} \# \text{ch\_ae}_1) \# \text{spp}) >\#\langle "+" \rangle\#\langle ((\text{sc} \# \text{ch\_ae}_2) \# \text{spp}) \end{aligned}$$

The children *ch\_ae<sub>1</sub>* and *ch\_ae<sub>2</sub>* are retrieved from the input family so we can subsequently retrieve the attribute *spp* from these attributions, and construct the computation of the synthesized attribute *spp*. The function *inhdef* introduces new inherited attributes for a collection of non-terminals at the same time, all with the same name.

$$\begin{aligned} \text{inhdef} &:: \text{Defs att nts vals ic ic}' \\ &\Rightarrow \text{att} \rightarrow \text{nts} \rightarrow \text{vals} \rightarrow (\text{Fam ic sp} \rightarrow \text{Fam ic}' sp) \end{aligned}$$

It results in a function which updates the output constructed thus far and takes the following parameters: the attribute *att* which is being defined, the list *nts* of non-terminals with which this attribute is being associated, and a record *vals* labeled with child names and containing values, describing how to compute the attribute being defined at each of the applicable child positions. The class *Defs* introduces a type-level function used to iterate over the record *vals* and to compute the new record of inherited attributes *ic'*, extending the record *ic* with the inherited attributes defined thus far.

Thus, in the rule *ienvLet*, described before, we give a definition for the attribute *ienv* for each child of which the semantic category is in the list  $\{\{ \text{nt\_Expr} \}\}$ , and these are stored in an extensible record labeled by the names of the children. It is the possibility of defining such functions in Haskell which shows the advantages of expressing one's attribute grammars using an embedded domain specific language.

**Rules Composition** The composition of two rules is the composition of the two functions resulting from applying each of them to the input family:

$$\begin{aligned} \text{ext} &:: \text{Rule sc ip ic}' \text{ sp}' \text{ ic}'' \text{ sp}'' \rightarrow \text{Rule sc ip ic sp ic}' \text{ sp}' \\ &\rightarrow \text{Rule sc ip ic sp ic}'' \text{ sp}'' \\ (\text{rule1 'ext' rule2}) \text{ input} &= \text{rule1 input} \circ \text{rule2 input} \end{aligned}$$

Figure 6 represents a composition *rule1 'ext' rule2*, of rules with two children. By inspecting the labyrinths of this figure, it can be seen how the inputs (black arrows) are shared and the outputs are combined by using the outputs of *rule2* (solid gray) as output constructed thus far of *rule1* (dotted gray). Thus, the outputs constructed thus far (dotted gray) of the composed rule are passed to *rule2* and the resulting outputs (solid gray) of the composed rule are equivalent to the resulting outputs of *rule1*.

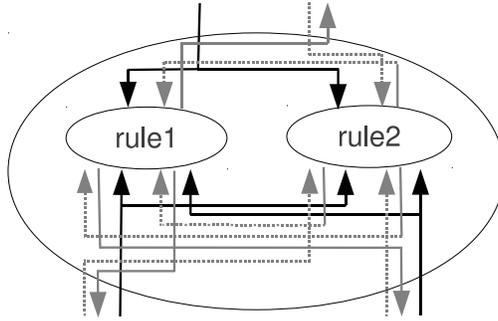


Fig. 6. Rules Composition: produces a new rule, represented by the external oval

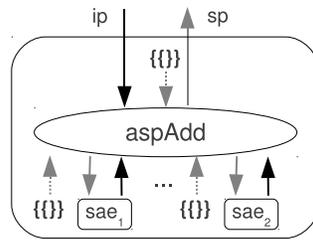


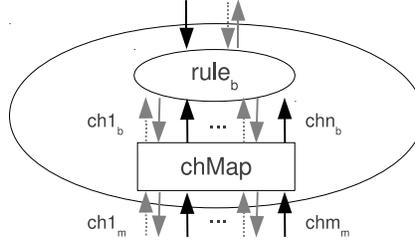
Fig. 7. Rule Knitting: produces a semantic function (external rounded rectangle)

**Semantic Functions** Figure 7 represents the resulting semantic function for the production *Add*. Notice that the function *knit* initializes the already constructed outputs with empty records (`{{}}`).

## 4 Attribute Grammar Macros

An attribute grammar macro is determined by a pair with the *base rule* ( $rule_b$ ) of the macro and the mapping ( $chMap$ ) between the children of this rule and their newly defined semantics, and returns a *macro rule*. As shown in Figure 8,  $chMap$  (rectangle) is an interface between the children of the base rule (inner oval) and the children of the macro rule (outer oval). The number of children of the macro rule (below  $chMap$  in the figure) does not need to be the same as the number of children of the base rule.

The function  $agMacro$  constructs the macro rule; it performs the “knitting” of  $rule_b$ , by applying this rule to its input and the output produced thus far. These elements have to be obtained from the corresponding elements of the macro rule and the mapping  $chMap$ . To keep the code clear, we will use the subindex  $b$  for the elements of the base rule and  $m$  for the elements of the macro rule. Thus, the macro rule takes as input the family ( $Fam\ sc_m\ ip_m$ ) and updates the output family constructed thus far ( $Fam\ ic_m\ sp_m$ ) to a new output family ( $Fam\ ic''_m\ sp'_m$ ):



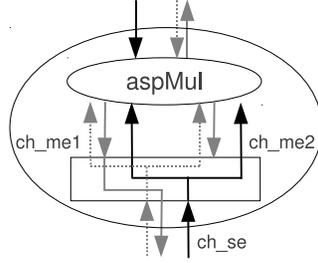
**Fig. 8.** AG Macro

$$\begin{aligned}
 & agMacro (rule_b, chMap) (Fam sc_m ip_m) (Fam ic_m sp_m) = \\
 & \quad \mathbf{let} \ ip_b = ip_m \\
 & \quad \quad sp_b = sp_m \\
 & \quad \quad (Fam ic'_b sp'_b) = rule_b \ (Fam sc_b ip_b) (Fam ic_b sp_b) \\
 & \quad \quad (ic'_m, ic_b, sc_b) = chMap (sc_m, ic_m) (ic'_b, emptyRecord, emptyRecord) \\
 & \quad \quad ic''_m = hRearrange (recordLabels ic_m) ic'_m \\
 & \quad \quad sp'_m = sp'_b \\
 & \quad \mathbf{in} \ (Fam ic''_m sp'_m)
 \end{aligned}$$

The inherited and synthesized attributes of the parent of the base rule ( $ip_b$  and  $sp_b$ ) respectively correspond to  $ip_m$  and  $sp_m$ , the inherited and synthesized attributes of the parent of the macro rule. The inherited and synthesized attributes of the children of the base rule ( $ic_b$  and  $sc_b$ ), as well as the updated inherited attributes of the children of the macro rule ( $ic'_m$ ), are generated by the children mapping function  $chMap$ . The function  $chMap$  takes as input a pair  $(sc_m, ic_m)$  with the synthesized attributes and the inherited attributes constructed thus far of the children of the macro rule, and returns a function that updates a triple with the updated inherited attributes ( $ic'_m$ ) of the children of the macro rule and the inherited ( $ic_b$ ) and synthesized ( $sc_b$ ) attributes of the children of the base rule. We start with an “initial” triple composed of the updated inherited attributes of the children of the base rule ( $ic'_b$ ), which has been converted into  $ic'_m$ , and two empty records (to be extended to  $ic_b$  and  $sc_b$ ). Notice that the attributes we pass to  $chMap$  are effectively the ones indicated by the incoming arrows in Figure 8.

The rearranging of  $ic'_m$  is just a technical detail stemming from the use of HList; by doing this we make sure that the children in  $ic_m$  and  $ic'_m$  are in the same order, thus informing the type system that both represent the same production. The synthesized attributes of the parent of the macro rule ( $sp'_m$ ) are just  $sp'_b$ , the synthesized attributes of the parent of the base rule.

Mapping functions resemble rules in the sense that they take an input and return a function that updates its “output”, that in this case is the triple  $(ic'_m, ic_b, sc_b)$  instead of an output family. Thus, they can be combined in the



**Fig. 9.** *aspSq*

same way as rules are combined; the combinator ( $\langle \cdot \rangle$ ), used in Figure 3, is exactly the same as the *ext* function but with a different type:<sup>7</sup>

$$\begin{aligned}
\langle \cdot \rangle &:: ((sc_m, ic_m) \rightarrow ((ic'1_m, ic1_b, sc1_b) \rightarrow (ic'2_m, ic2_b, sc2_b))) \\
&\rightarrow ((sc_m, ic_m) \rightarrow ((ic'0_m, ic0_b, sc0_b) \rightarrow (ic'1_m, ic1_b, sc1_b))) \\
&\rightarrow ((sc_m, ic_m) \rightarrow ((ic'0_m, ic0_b, sc0_b) \rightarrow (ic'2_m, ic2_b, sc2_b))) \\
(chMap1 \langle \cdot \rangle chMap2) \text{ inp} &= chMap1 \text{ inp} \circ chMap2 \text{ inp}
\end{aligned}$$

We use the combinator ( $\longleftrightarrow$ ) to map a child  $lch_b$  of the base rule to a child  $lch_m$  of the macro rule.

$$\begin{aligned}
lch_b \longleftrightarrow lch_m &= \lambda(sc_m, ic_m) (ic'0_m, ic0_b, sc0_b) \rightarrow \\
&\mathbf{let} \ ic'1_m = hRenameLabel \ lch_b \ lch_m \ (hDeleteAtLabel \ lch_m \ ic'0_m) \\
&\quad ic1_b = lch_b \ .\# \ (ic_m \ \# \ lch_m) \ .\# \ ic0_b \\
&\quad sc1_b = lch_b \ .\# \ (sc_m \ \# \ lch_m) \ .\# \ sc0_b \\
&\mathbf{in} \ (ic'1_m, ic1_b, sc1_b)
\end{aligned}$$

The updated inherited attributes for the child  $lch_m$  correspond to the updated inherited attributes of the child  $lch_b$ . Thus, the new  $ic'_m$  ( $ic'1_m$ ) is the original one with the field  $lch_b$  renamed to  $lch_m$ . Since more than a single child of the base rule can be mapped to a child of the macro rule, like in *aspSq* of Figure 3, we have to avoid duplicates in the record by deleting a possible previous occurrence of  $lch_m$ . This decision fixes the semantics of multiple occurrences of a child in a macro: the child will receive the inherited attributes of its left-most mapping. We represent this behavior in Figure 9 with the gray arrow, which corresponds to the inherited attributes of  $ch\_me2$ , pointing nowhere outside the mapping. In the cases of the initial inherited attributes and the synthesized attributes, they have to be extended with a field corresponding to the child  $lch_b$  with the attributions for the child  $lch_m$  from the inherited and synthesized attributes, respectively, of the macro rule.

Inside a macro a child can be mapped to some other macro ( $rule_c, chMap$ ), where the subindex  $c$  stands for child. This is the case of the definitions of

<sup>7</sup> To avoid confusion with rule combination, instead of using apostrophes to denote updates we use numeric suffixes

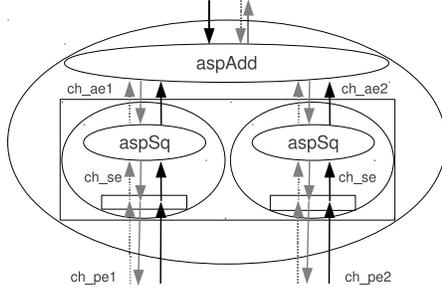


Fig. 10. aspPyth

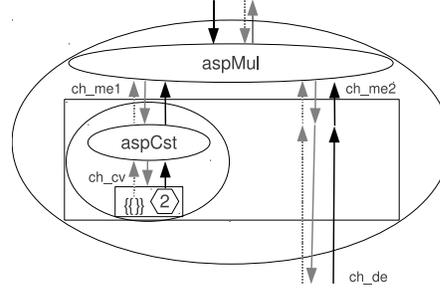


Fig. 11. aspDouble

*aspPyth* and *aspDouble*, graphically represented in Figure 10 and Figure 11, where the rectangles representing the children mappings have rules (ovals) inside.

$$\begin{aligned}
lch_b \implies (rule_c, chMap) &= \lambda(sc_m, ic_m) (ic'_0_m, ic0_b, sc0_b) \rightarrow \\
\mathbf{let} (Fam\ ic'_c\ sp'_c) &= agMacro (rule_c, chMap) (Fam\ sc_m (ic'_0_m \# lch_b)) \\
&\quad (Fam\ ic_m\ emptyRecord) \\
ic'_1_m &= hLeftUnion\ ic'_c (hDeleteAtLabel\ lch_b\ ic'_0_m) \\
ic1_b &= lch_b \text{ .}=. emptyRecord \text{ .}*. ic0_b \\
sc1_b &= lch_b \text{ .}=. sp'_c \quad \text{ .}*. sc0_b \\
\mathbf{in} (ic'_0_m, ic1_b, sc1_b)
\end{aligned}$$

In this case, the inner macro has to be evaluated using *agMacro*. The children of the inner macro will be included in the children of the outer macro; thus the synthesized attributes of the inner macro are included in  $sc_m$ , and the new inherited attributes of the children have to extend  $ic_m$ . The inherited attributes of the parent of the inner macro are the inherited attributes of the child  $lch_b$  of the base rule of the outer macro. The synthesized attributes of the parent of the inner macro are initialized with an empty attribution. The child  $lch_b$  is removed from  $ic'_0_m$ , because the macro rule will not include it. On the other hand, the inherited attributes of the children of the inner macro ( $ic'_c$ ) have to be added to the inherited attributes of the children of the macro. With the function *hLeftUnion* from HList we perform an union of records, choosing the elements of the left record in case of duplication. We initialize the inherited attributes for  $lch_b$  with an empty attribution, since it cannot be seen “from the outside”. The synthesized attributes are initialized with the resulting synthesized attributes of the inner rule.

With the combinator ( $\rightsquigarrow$ ) we define a mapping from a child with label  $lch$  to a literal value  $cst$ . For the base rule, the initial synthesized attributes of the child  $lch_b$  are fixed to the literal  $cst$ .

$$\begin{aligned}
lch_b \rightsquigarrow cst &= \lambda(-, -) (ic'_0_m, ic0_b, sc0_b) \rightarrow \\
\mathbf{let} ic'_1_m &= hDeleteAtLabel\ lch\ ic'_0_m \\
ic1_b &= lch_b \text{ .}=. emptyRecord \text{ .}*. ic0_b
\end{aligned}$$

$$\begin{array}{l} sc1_b = lch_b .=. cst \quad *. sc0_b \\ \mathbf{in} (ic'1_m, ic1_b, sc1_b) \end{array}$$

The (internal) macro associated to the mapping of the child  $ch\_me_1$  in Figure 11 shows the semantics of the combinator ( $-\rightsquigarrow$ ). The synthesized attributes of  $ch\_cv$  are fixed to the constant (hexagon) 2. Since the child is mapped to a constant, the inherited attributes are ignored (the arrow points nowhere). Although, we have to provide a (empty) set of inherited attributes constructed thus far to the rule  $aspCst$ .

#### 4.1 Attribute Redefinitions

We have shown how to introduce new syntax and how to express its meaning in terms of existing constructs. In this section we show how we can *redefine* parts of the just defined semantics by showing how to redefine attribute computations.

The function  $synmod$  (and its monadic version  $synmodM$ ) modifies the definition of an existing synthesized attribute:

$$synmod\ att\ val\ (Fam\ ic\ sp) = Fam\ ic\ (hUpdateAtLabel\ att\ val\ sp)$$

Note that the only difference between  $syndef$ , from subsection 3.1, and  $synmod$ , is that the latter updates an existing field of the attribution  $sp$ , instead of adding a new field. With the use of the HList's function  $hUpdateAtLabel$  we enforce (by type class constraints) the record  $sp$ , which contains the synthesized attributes of the parent constructed thus far, indeed contains a field labeled  $att$ . Thus, a rule created using  $synmod$  has to extend, using  $ext$ , some other rule that has already defined the synthesized attribute this rule is *redefining*.

The AspectAG library also provides functions  $inhmodM$  and  $inhmod$ , analogous to  $inhdefM$  and  $inhdef$ , that modify the definition of an inherited attribute for all children coming from a specified collection of semantic categories.

## 5 Conclusions and Future Work

Building on top of a set of combinators that allow us to formulate extensions to semantics as first class attribute grammars (i.e. as plain typed Haskell values), we introduced in this paper a mechanism which allows us to express semantics in terms of already existing semantics, without the need to use higher order attributes.

The programmer of the extensions does not need to know the details of the implementation of every attribute. In order to implement a macro or a redefinition for a production he only needs the names of the attributes used and the names of the children of the production, the latter being provided by the definition of the abstract syntax tree.

This work is part of a bigger plan, involving the development of a series of techniques [1, 2, 12, 13] to deal with the problems involved in both syntactic and semantic extensions of a compiler by composing compiled and type-checked

Haskell values. In this way we leverage the type checking capabilities of the Haskell world into such specifications, and we profit from all the abstraction mechanisms Haskell provides.

We already think that the current approach is to be preferred over stacking more and more monads when defining a compositional semantics as is conventionally done in the Haskell world [11].

## References

1. Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI 2009*, pages 15–26. ACM, 2009.
2. Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *LDTA 2009*, ENTCS, 2009.
3. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.
4. Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.
5. Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell 2009*, pages 93–104. ACM, 2009.
6. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *ICFP 2003*, pages 3 – 13. ACM Press, 2003.
7. Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004*, pages 96–107. ACM Press, 2004.
8. B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.
9. William Maddox. Semantically-sensitive macroprocessing. Technical report, Berkeley, CA, USA, 1989.
10. Conor McBride. Faking it simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, 2002.
11. Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP 2011*, pages 32–44. ACM, 2011.
12. S. Doaitse Swierstra. Parser combinators: from toys to tools. In *Haskell Workshop*, 2000.
13. S. Doaitse Swierstra. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, pages 252–300, 2008.
14. S. Doaitse Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*. Springer, 2009.
15. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *LNCS*. Springer-Verlag, 2002.
16. Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.
17. Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In *ICFP 2009*, pages 245–256. ACM, 2009.
18. Niklaus Wirth. *Compiler construction*. International computer science series. Addison-Wesley, 1996.