UUAG Meets AspectAG

How to make Attribute Grammars First-Class

Marcos Viera

S. Doaitse Swierstra

Arie Middelkoop

Technical Report UU-CS-2011-029
Oct 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

UUAG Meets AspectAG: How to make Attribute Grammars First-Class

Marcos Viera*[1], S. Doaitse Swierstra†[2] and Arie Middelkoop‡[2]

[1]Instituto de Computación , Universidad de la República, Montevideo, Uruguay
[2]Department of Computer Science, Utrecht University, Utrecht, The Netherlands

October 12, 2011

Abstract

The Utrecht University Attribute Grammar Compiler (*UUAGC*) takes attribute grammar declarations from *multiple source files* and generates an attribute grammar evaluator consisting of *a single Haskell source text*. A problem with this generative approach is that, once the code is generated and compiled, neither new attributes can be introduced nor existing ones can be modified without providing access to all the source code and without having to regenerate and recompile the entire program.

In contrast to this textual approach we recently constructed the Haskell combinator library *AspectAG* with which one can construct attribute grammar fragments as a *Haskell value*. Such descriptions can be individually type-checked, compiled, distributed and composed to construct a compiler. This method however results in rather inefficient compilers, due to the cost of composition.

In this paper we show how we can combine the two approaches by generating the *AspectAG* code fragments from original *UUAGC* sources, making it possible to trade between efficiency and flexibility. This both enables a couple of optimizations for *AspectAG* resulting in a considerable speed improvement and makes that existing *UUAGC* code can be reused in a flexible environment.

1 Introduction

The key advantage of using attribute grammar systems is that they allow us to describe the semantics of a programming language *in a modular way*. A complete evaluator can be assembled from a large collection of attribute grammar fragments, each describing a specific aspect of the language at hand. As such, attribute grammars provide a solution to the so-called expression problem.

The solutions to the quest for modular language description can be found at the textual level, as in most attribute grammar systems, or at the semantic level, where we define language descriptions as first class values, which can be linked together to form a complete language description.

The first approach is supported by, amongst many others, the Utrecht Attribute Grammar System [DS04], which reads in a complete language definition from a large collection of files, each describing a separate aspect of the language. These fragments are assembled and analyzed together. From the result a large monolithic compiler is generated. This approach leads to efficient compilers, which can however not easily be changed once generated and compiled. Thus, adding something like a syntax-macro mechanism becomes cumbersome.

*mviera@fing.edu.uy
†doaitse@swierstra.net
‡ariem@cs.uu.nl

At the other extreme we find the attempts to assemble the semantics from individual fragments, which, in case of Haskell, use monad transformers to stack a large collection of relatively independent computations over the syntax tree, each taking care of one of the aspects that together make up a complete compiler [Jon99, SO11]. Unfortunately the monad-based approach comes with its own problems: one gets easily lost in the stack of monads, one is sometimes obliged to impose an order which does not really make sense, and the type system makes it hard to e.g. compose state out of a number of individual states which probably carry the same type. Furthermore the implicit order in which attributes have to be evaluated becomes very explicit in the way the monads are composed.

Recently we have designed a completely different way of constructing first-class language definition fragments, by designing a collection of combinators (the *AspectAG* Haskell package) which make it possible to formulate attribute grammars using an Embedded Domain Specific Language in Haskell [VSS09]. Albeit easy to use for the experienced Haskell programmer, it has a steep learning curve for the uninitiated. A second disadvantage is that the approach is relatively expensive: in order to be able to redefine attributes or to add new attributes later, we encode the lists of inherited and synthesized attributes of a non-terminal as an *HList*-encoded [KLS04] value, indexed by types using the Haskell class mechanism. In this way the checking for the well-formedness of the attribute grammar is realized through the Haskell class system. Once the language gets complicated (in our Haskell compiler *UHC* [DFS09] some non-terminals have over 20 attributes), the cost of accessing attributes overshadows the cost of the actual computations.

In this paper we seek to alleviate the aforementioned two problems by using the original *UUAGC* input code to generate the *AspectAG* code from. We furthermore take the opportunity here to group collections of attributes which are not likely to change, so the *HList* values are shortened considerably, thus relieving the costs of the extra available expressibility. Only the attributes which are likely to be adapted in other language fragments have to be made part of these *HList* values at the top level; hence we only pay for the extra flexibility where needed.

In section 2 we describe the way the *UUAGC* represents grammars and our running example, which consists of an initial language fragment and a small extension. In section 3 we describe how to generate *AspectAG* code out of the *UUAGC* sources, whereas in section 4 we describe how some optimizations to the generated code can be performed, and show some figures showing the performance gain achieved in this way.

2 Attribute Grammars

2.1 Initial Attribute Grammars

As running example we use a small expression language with declarations, of which the semantics boils down to the evaluation of the main expression. In Figure 1 we show an implementation of the semantics in terms of attribute grammars, using the syntax of the Haskell preprocessor Utrecht University Attribute Grammar Compiler (*UUAGC*).

An Attribute Grammar is a context-free grammar where the nodes in the parse tree are decorated with (a usually quite large) a number of values, called *attributes*. The grammar describing the abstract syntax trees of the language is introduced by the **DATA** definitions *Root*, *Expr* and *Decls*. Attributes define semantics for the language in terms of the grammar and in their defining expression may refer to other attributes. A tree-walk evaluator generated from the AG computes values for these attributes, and thus provides implementations for the language's semantics in the form of compilers and interpreters. There are two kinds of attributes: *synthesized*, and *inherited* attributes. For each production we distinguish two sets of attributes: the *input-family*, which contains the inherited attribute of the parent node and the synthesized attributes of the children nodes, and the *output-family*, consisting of the inherited attributes of the children nodes and the synthesized attributes of the parent node. For each rule and for each member of the output family of that rule, we define how it is to be computed in terms of the members of the input family. [1]

[1] We use the following naming convention for attributes: all synthesized attributes start with 's' and all inherited

LangDef.ag

```
DATA Root
| Root decls : Decls main : Expr
DATA Decls
| Decl name : String val : Expr rest : Decls
| NoDecl
DATA Expr
| Add left : Expr right : Expr
| Val value : Int
| Var var : String

ATTR Root Expr SYN sval : Int
SEM Root
| Root   lhs.sval  = main.sval
SEM Expr
| Add    lhs.sval  = left.sval + right.sval
| Val    lhs.sval  = value
| Var    lhs.sval  = case lookup var lhs.ienv of
                       Just v   -> v
                       Nothing -> 0

ATTR Decls Expr INH ienv : [(String, Int)]
SEM Root
| Root   decls.ienv = []
         main.ienv  = decls.senv
SEM Expr
| Add    left.ienv  = lhs.ienv
         right.ienv = lhs.ienv
SEM Decls
| Decl   val.ienv   = []
         rest.ienv  = (name, val.sval) : lhs.ienv

ATTR Decls SYN senv : [(String, Int)]
SEM Decls
| Decl   lhs.senv = rest.senv
| NoDecl lhs.senv = lhs.ienv
```

Figure 1: AG specification of the language semantics

In our example (Figure 1) we use three attributes: one attribute (**SYN** *sval*) holding the result value, one attribute (**INH** *ienv*) in which we assemble the environment from the declarations (*ienv*) and one atribute (**SYN** *senv*) for passing the final environment back to the Root so it can be used in the main expression.

In a **SEM** block we specify how attributes from the output family are to be computed out of attributes from the input family. The defining expressions at the right hand side of the =-signs are almost plain Haskell code, using minimal syntactic extensions to refer to attributes from the input family. We refer to a synthesized attribute of a child using the notation *child.attribute* and

attributes start with 'i'.

to an inherited attribute of the production itself (the left-hand side) as **lhs**.*attribute*. Terminals are referred to by the name introduced in the **DATA** declaration. For example, the rule for the attribute *ienv* for the child *rest* of the production *Decl* extends the inherited list *ienv* by a pair composed of the *name* used in the declaration and the value *sval* of the child with name *val* (*val.sval*).

When the *UUAGC* compiler generates a Haskell program out of a collection of **SEM** blocks the rules' expressions are copied almost verbatim into the generated program: only the attribute references are replaced by references to values defined by the generated program. The *UUAGC* compiler checks whether a definition has been given for each attribute; whereas type checking of the defining expressions is left to the Haskell compiler when compiling the generated program.

2.2 Attribute Grammar Extensions

LangExt.ag

```
ATTR Root Decls Expr SYN serr : [String]
SEM Root
| Root     lhs.serr = decls.serr ++ main.serr
SEM Decls
| Decl     lhs.serr
    = (case lookup name lhs.ienv of
         Just _   -> [name ++ " duplicated"]
         Nothing -> []) ++ val.serr ++ rest.serr
| NoDecl lhs.serr = []
SEM Expr
| Add      lhs.serr = left.serr ++ right.serr
| Val      lhs.serr = []
| Var      lhs.serr
    = case lookup var lhs.ienv of
         Just _   -> []
         Nothing -> [var ++ " undefined"]
```

Figure 2: Semantics extended with an attribute that collects errors

In this subsection we now show how we can extend the given language, *without touching the code written*. The use of variables and declarations in the example language can be erroneous. Computing error messages describing such situations is straightforward (Figure 2); we introduce an extra synthesized attribute (*serr*) in which we collect the error messages corresponding to dual declarations (*name* is already an element of the *ienv*) and to absent declarations (*name* is not an element of *ienv*).

To compile this code using *UUAGC* and *ghc* we have to follow the process described in Figure 3; i. e. use *UUAGC* to generate a completely fresh Haskell file out of the two related attribute grammar sources, and compile and link it with *ghc*. Keep in mind that by doing that we are only generating the semantics part of the compiler, which has to be completed with a few lines of main program containing the parsers and referring to these semantics.

With our solution almost the same code has to be written but, by passing some extra flags to *UUAGC*, the compilation process is completely different (Figure 4). This approach enables us have a compiled definition of the semantics of a core language and to introduce relative small extensions to it later, without neither the need to reconstruct the whole compiler, nor even requiring the sources of the core language to be available! Thus, for example, a core language compiler and a set of optional extensions can be distributed (without sources), such that the user can link his own extended compiler together.

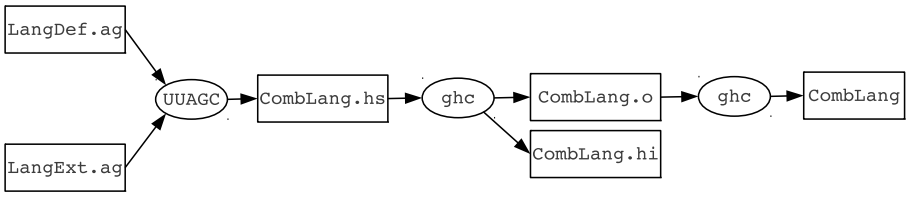

Figure 3: Compilation Process with UUAGC

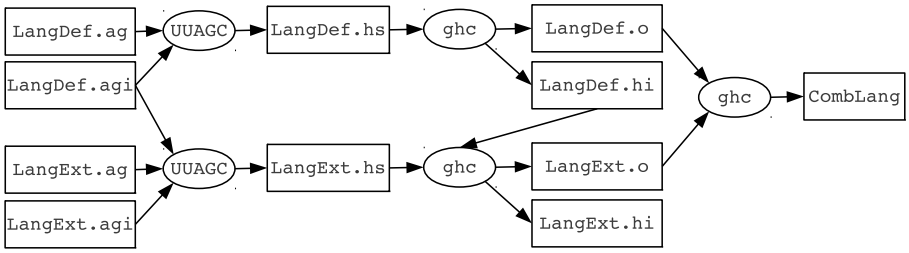

Figure 4: Compilation Process with our extension of UUAGC

To allow this extension in *UUAGC* the flag `--aspectag` has to be passed in the following way:

```
uuagc -a --aspectag LangDef
uuagc -a --aspectag LangExt
```

With `--aspectag` we make *UUAGC* generate *AspectAG* code out of a set of `.ag` files and their corresponding `.agi` files, as we show in the following sections.

An `.agi` file includes the declaration of a grammar and its attributes (the interface), while the **SEM** blocks specifying the computation of these attributes are included in the `.ag` file (the implementation). Figure 5 shows the attribute grammar specification of Figure 1 adapted to our approach. Notice that the code is exactly the same, although distributed over a file `Langdef.agi` containing **DATA** and **ATTR** declarations, and a file `Langdef.ag` with the rules.

In Figure 6 we adapt the extension of Figure 2. In this case a new keyword **EXTENDS** is used to indicate which attribute grammar is being extended. Extensions are incremental. Thus, if we define yet another extension (Figure 7) which adds a new production representing negating expressions to the attribute grammar resulting from the previous extension *LangExt*, the rules for the attributes *sval*, *ienv* and *serr* have to be defined.

3 From UUAG to AspectAG

Before describing the internal workings of *AspectAG* we show how *UUAGC* compiles attribute grammars into Haskell based evaluators using the code in Figure 1 as an example. It consists of the data types declarations for *Root*, *Decls* and *Expr*, corresponding to the non-terminals of the grammar, and functions *sem_Root*, *sem_Decls* and *sem_Expr* to compute the values of the attributes.

The generation of the data types out of the **DATA** declarations is straightforward, representing each production by a constructor of the type corresponding to the non-terminal:

data *Root* = *Root Decls Expr*
data *Decls* = *Decl String Expr Decls* | *NoDecl*
data *Expr* = *Add Expr Expr* | *Val Int* | *Var String*

LangDef.agi

```
DATA Root
| Root decls : Decls main : Expr
DATA Decls
| Decl name : String val : Expr rest : Decls
| NoDecl
DATA Expr
| Add left : Expr right : Expr
| Val value : Int
| Var var : String

ATTR Root Expr SYN sval : Int
ATTR Decls Expr INH ienv : [(String, Int)]
ATTR Decls SYN senv : [(String, Int)]
```

LangDef.ag

```
SEM Root
| Root    lhs.sval  = main.sval
SEM Expr
| Add     lhs.sval  = left.sval + right.sval
| Val     lhs.sval  = value
| Var     lhs.sval  = case lookup var lhs.ienv of
                        Just v    -> v
                        Nothing -> 0

SEM Root
| Root    decls.ienv = []
          main.ienv = decls.senv
SEM Expr
| Add     left.ienv  = lhs.ienv
          right.ienv = lhs.ienv
SEM Decls
| Decl    val.ienv   = []
          rest.ienv  = (name, val.sval) : lhs.ienv

SEM Decls
| Decl    lhs.senv = rest.senv
| NoDecl  lhs.senv = lhs.ienv
```

Figure 5: Language semantics

The *semantic function* for a non-terminal is a function that takes an abstract syntax tree and builds the function that maps the inherited attributes of the root of this tree to its synthesized attributes. The body of semantic functions is defined using the rules for the output attributes of the production and by calling the semantic functions on the children of the root node with their inherited attribute values as parameters, and returning the values of their synthesized attributes. The following is the code of the "tree-walk evaluator" for the productions of the non-terminal

LangExt.agi

```
EXTENDS "LangDef"
ATTR Root Decls Expr SYN serr : [String]
```

LangExt.ag

```
SEM Root
| Root    lhs.serr = decls.serr ++ main.serr
SEM Decls
| Decl    lhs.serr
    = (case lookup name lhs.ienv of
         Just _   -> [name ++ " duplicated"]
         Nothing -> []) ++ val.serr ++ rest.serr
| NoDecl lhs.serr = []
SEM Expr
| Add     lhs.serr = left.serr ++ right.serr
| Val     lhs.serr = []
| Var     lhs.serr
    = case lookup var lhs.ienv of
         Just _   -> []
         Nothing -> [var ++ " undefined"]
```

Figure 6: Language Extension: Errors

Decls:[2]

```
sem_Decls (Decl name val rest)
  = λ(ienv) -> let (vres ) = (sem_Expr  val )
                             ([]                  )
                   (renv) = (sem_Decls rest)
                             ((name, vres) : ienv)
               in  (renv)
sem_Decls (NoDecl           )
  = λ(ienv) ->     (ienv)
```

In the case of the production *Decl*, the inherited attribute *ienv* is the input of the function. The semantic functions of the children *val* and *res* are invoked, applied to the code defined in Figure 1 for their inherited attributes, to obtain their synthesized attributes *vres* and *renv*. The resulting synthesized attribute is *renv* (in Figure 1 *rest.senv*).

If we add a new attribute, like the one in Figure 2, we have to inspect the whole grammar and modify all the semantic functions which will refer to this attribute. We then generate entirely new code and the new *sem_Decls* becomes:

```
sem_Decls (Decl name val rest)
  = λ(ienv) ->
      let (vres, verr) = (sem_Expr  val )
                         ([]                  )
          (renv, rerr) = (sem_Decls rest)
                         ((name, vres) : ienv)
      in  (restsenv, (case lookup name ienv of
                        Just _   -> [name ++
```

[2]A superfluous pair of parentheses shows where in principle Cartesian products are generated.

`LangExt2.agi`

EXTENDS `"LangExt"`
DATA *Expr*
| *Neg expr* : *Expr*

`LangExt2.ag`

SEM *Expr*
| *Neg* **lhs**.*sval* = −*expr*.*sval*

SEM *Expr*
| *Neg expr*.*ienv* = **lhs**.*ienv*

SEM *Expr*
| *Neg* **lhs**.*serr* = *expr*.*serr*

Figure 7: Language Extension: Negation

```
                                        " duplicated"]
                          Nothing → []) ++ verr ++ rerr)
sem_Decls (NoDecl                )
  = λ(ienv) →
        (ienv,      [])
```

In the following subsections we introduce *AspectAG* [VSS09], a strongly-typed Haskell library for attribute grammars, where our individual language fragments become first-class values which can be compiled, stored, redefined and combined. While introducing *AspectAG* concepts we will also show how we automatically may generate this code out of a *UUAGC* based description.

3.1 Record-based Approach

AspectAG is based on the idea of de Moor et al. [dMBS00, dMPJW00] of splitting the semantic functions into the rules determining the computation of the attributes and their application to the semantics of the children, in such a way that attribute computations can be manipulated and combined before being applied.

When computing the attribute values for a production instance in the abstract syntax tree, information flows from the inherited attributes of the parent (or left-hand side) and the synthesized attributes of each of the children (together called the *input family* from now on) to the synthesized attributes of the parent and the inherited attributes of the children (henceforth called in the *output family*).

In *AspectAG* families are represented with a type:

data *Fam parent children* = *Fam parent children*

where both *parent* and *children* are extensible records, which are implemented using *HList* [KLS04] typeful heterogeneous collections. The record *parent* represents the set of attributes for the parent and the record *children* is a collection of records, each one containing the attributes for a child. Notice that the labels of the fields in *children* determine the production for which a family is defined.

In order to make attribute computation composable we define a rule as a mapping from the attributes in the input family to a function which extends a family of output attributes with the new elements defined by this rule:

```
type Rule sc ip ic sp ic' sp'
  = Fam sc ip → (Fam ic sp → Fam ic' sp')
```

Thus, the type *Rule* states that a rule takes as input the synthesized attributes of the children *sc* and the inherited attributes of the parent *ip* and returns a function from the output family constructed thus far (inherited attributes of the children *ic* and synthesized attributes of the parent *sp*) to the extended output, which in principle may contain more attributes as indicated by the new types *ic'* and *sp'*. Note that a rule does not change the input family.

3.2 Grammar

Since we use extendible records, labels have to be generated to refer to the children of the productions of the grammar. A elements in an *HList* is referred to by a plain Haskell value of a singleton type, where such types are used to represent type-level values, and classes are used to represent type-level types and functions [Hal01, McB02]. The children labels generated out of the `.agi` files of the example are:

- From Figure 5: *ch_decls_Root_Root*, *ch_main_Root_Root*, *ch_name_Decls_Decl*, *ch_val_Decls_Decl*, *ch_rest_Decls_Decl*, *ch_left_Add_Expr*, *ch_right_Add_Expr*, *ch_value_Val_Expr* and *ch_var_Var_Expr*.
- From Figure 7: *ch_expr_Neg_Expr*.

3.3 Attribute Definition

A collection of synthesized or inherited attributes is represented by an *Hlist* value too, thus for each **ATTR** declaration a label has to be generated to refer to the defined attribute. By convention we use the name of the attribute prefixed by *att_* for attribute labels. For example, the declaration **ATTR** *Decls* **SYN** *senv* : {[(*String*, *Int*)]} generates the label *att_senv*.

The *AspectAG* function *syndef* adds the definition of a synthesized attribute. It takes a label *att* representing the name of the new attribute and a monadic computation (that "reads" from the input family *Fam sc ip*) resulting in the value (of type *a*) to be assigned to the attribute, and it builds a function which updates the record *sp* containing the synthesized attributes of the family output constructed thus far.

```
syndef ::  HExtend (Att att a) sp sp'
        ⇒ att → Reader (Fam sc ip) a
        → Rule sc ip ic sp ic sp'
```

The constraint *HExtend* (*Att att a*) *sp sp'* is a "type-level function" that extends the record *sp* with a field with label *att* and a value with type *a*.

We use *syndef* to generate the code for the rules for the synthesized attributes, like:

```
SEM Decls
| Decl lhs.senv = rest.senv
```

Resulting in the code:

```
senv_Decls_Decl = syndef att_senv $
  do
    rest ← at ch_rest_Decls_Decl
    return $ rest # att_senv
```

where *at ch_rest_Decls_Decl* locates the *rest* child using the label *ch_rest_Decls_Decl* in the record *sc* of the environment with type *Fam sc ip*. Having this record bound to *rest* the *HList* lookup operator # is used to locate the value of the attribute *att_senv*. The uses of such calls to *at* will

inform the type system that the input family *Fam sc ip* has to have a child *ch_rest_Decls_Decl* with a defined attribute *att_senv*. Such constraints turn up as class constraints, to be checked by the Haskell type checker.

The same procedure is followed to generate code for the inherited attributes, but using the function *inhdef*:

```
inhdef :: Defs att nts a ic ic'
       ⇒ att → nts → Reader (Fam sc ip) a
       → Rule sc ip ic sp ic' sp
```

Here the type *a* is a record with the computations for the children of the production, *nts* is a list of labels representing the non-terminals for which the attribute is defined (generated out of the **ATTR** declarations), and *Defs* is a "type-level function" that iterates over *a* extending the corresponding records in *ic*. Thus, for the declarations:

```
SEM Decls
| Decl val.ienv  = []
      rest.ienv = (name, val.sval) : lhs.ienv
```

The following code is generated:

```
ienv_Decls_Decl = inhdef att_ienv nts_ienv $
  do
    lhs    ← at lhs
    name ← at ch_name_Decls_Decl
    val    ← at ch_val_Decls_Decl
    return
      {{ ch_val_Decls_Decl  .=. []
      ,  ch_rest_Decls_Decl .=.
           (name, val # att_sval) : lhs # att_ienv }}
```

Where *lhs* returns the record *ip* (inherited attributes of the parent) from the input family *Fam sc ip* and the {{...}} notation is just syntactic sugar for *HList* extensible records, equivalent to the list notation [...].

3.4 Generating the Semantic Functions

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

```
ext :: Rule sc ip ic' sp' ic'' sp''
    → Rule sc ip ic sp ic' sp'
    → Rule sc ip ic sp ic'' sp''
(f `ext` g) input = f input.g input
```

Thus, when generating *AspectAG* code, all the rules for the attributes of each production are composed. In the example of Figure 5 the following composition is generated for the production *Decl*:

```
atts_Decls_Decl = ienv_Decls_Decl `ext`
                  senv_Decls_Decl
```

Once the computations are composed, the semantic functions corresponding to each production can be generated by applying them to the semantic functions of the children of the production; i. e. connecting the components of the DGG. This is the job of *AspectAG*'s function *knit*, which takes a (composite) rule and a record containing the semantic functions of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes.

```
knit :: (Kn fc ic sc, Empties fc ec)
     ⇒ Rule sc ip ec (Record HNil) ic sp
     → fc → (ip → sp)
```

Since a rule returns a function which extends an output family, the function *knit* also has to produce an appropriate empty output family to apply to the rule. The semantic function of the non-terminal *Decl* is:

```
sem_Decls_Decl     = knit atts_Decls_Decl
sem_Decls_NoDecl = knit atts_Decls_NoDecl

sem_Decls (Decls_Decl _name _val _rest)
   = sem_Decls_Decl
        {{ ch_name_Decls_Decl .=. (sem_Term _name)
        ,  ch_val_Decls_Decl  .=. (sem_Expr  _val)
        ,  ch_rest_Decls_Decl .=. (sem_Decls _rest) }}
sem_Decls (Decls_NoDecl)
   = sem_Decls_NoDecl {{ }}
```

Note that the definition of the *sem_* functions only depends on the shape (names and types of the children) of each production. Thus, this code is generated out of the **DATA** declarations.

3.5 Extensions

The keyword **EXTENDS** indicates that an attribute grammar declaration *extends* an existing attribute grammar. In an extension we can both add new attributes or productions or redefine the computation of existing attributes.

When the code of an extension is generated, the names of context-free grammar and the previously defined attributes have to be imported from the code generated for the system to extend. We take this information from the (chain of) `.agi` file(s) of the extended module. Thus, for the example of Figure 6 we generate the import:

```
import LangDef
  (nt_Root, nt_Decls, nt_Expr
  , ch_decls_Root_Root, ch_main_Root_Root
  , ch_name_Decls_Decl, ch_val_Decls_Decl
  , ch_rest_Decls_Decl
  , ch_left_Add_Expr, ch_right_Add_Expr
  , ch_value_Val_Expr, ch_var_Var_Expr
  , sval, ienv, senv)
```

We also generate a qualified import of the whole module, so we can refer to already defined rules without name clashes:

```
import qualified LangDef
```

So, when introducing new attributes, we can perform the composition for each production where the attribute is defined, and knit it again. For example:

```
atts_Decls_Decl = serr_Decls_Decl `ext`
                  LangDef.atts_Decls_Decl
sem_Decls_Decl = knit atts_Decls_Decl
```

By building on top of the *AspectAG* library, we can both add new attributes, and overwrite existing ones. If we want to extend the example language in such a way that an expression in a declaration may refer to sibling declarations, we can do so by redefining the definition for the

LangExt3.agi

EXTENDS "LangExt2"

LangExt3.ag

SEM *Decls*
| *Decl val.ienv* := *rest.senv*

Figure 8: Language Extension: Attribute *ienv* redefined to allow the use of variables in declarations

environment we pass to such right-hand side expressions. In Figure 8 we show how this can be done using := instead of =, an extension to *UUAGC* syntax for declaring *attribute redefinitions*. When an attribute is overwritten using := a similar approach as when defining new attributes is taken. Instead of using *syndef* and *inhdef* to define attributes, the functions *synmod* and *inhmod* are used, which are almost identical to their respective *def* functions, with the difference that instead of extending a record with a new attribute, the value of an existing attribute is updated in the record.

4 Optimizations

The flexibility provided by the use of list-like structures to represent collections of attributes (and children) of productions has its consequences in terms of performance. In this section we propose a couple of optimizations, based on changing some of the extensible records we use by normal records (Cartesian products). Both optimizations can be performed automatically by the transformation.

4.1 Grouping Attributes

If the number of attributes is fixed and the attributes won't be redefined, the use of extensible records is not necessary. Thus, in those cases we can group all the synthesized attributes of a non-terminal into a single attribute *att_syn* and the inherited attributes into an attribute *att_inh*. The type of a grouping attribute is a (non extensible) record containing the grouped attributes.

Attributes defined in extensions cannot be grouped with the original attributes. Thus in our running example applying grouping does not make much sense, since every group will have only one attribute. But if the specifications in Figures 5 and 6 were joined in the generation process we will have the attributes *att_inh* and *att_syn* for *Decls* with types:

```
data Inh_Decls = Inh_Decls
                 { ienv_Inh_Decls :: [(String, Int)] }
data Syn_Decls = Syn_Decls
                 { senv_Syn_Decls :: [(String, Int)]
                 , serr_Syn_Decls :: [String] }
```

To define and access to the grouped attributes, one more level of indirection is added. Thus, the definition of *att_syn* for the production *Decl* is:

```
syn_Decls_Decl = syndef att_syn $
  do
    rest ← at ch_rest_Decls_Decl
    return
      Syn_Decls { senv_Syn_Decls =
        (senv_Syn_Decls (rest # att_syn)) }
```

By default, all the attributes of every production are grouped, but grouped attributes cannot be redefined without having to make changes to the entire group. The flag `--nogroup` lets us specify the list of attributes we do not want to include in the grouping. For example, the following call to *uuagc* generates the *AspectAG* code for the example with all the attributes grouped but *ienv*, which will be redefined in the extensions:

```
uuagc -a --aspectag --nogroup=ienv LangDef.ag
```

4.2 Static Productions

If we do not need the possibility to change the definition of already existing productions (i. e . we will never add or remove children to some productions), a less flexible approach to represent productions can be taken. The flag `--static` activates an optimization where the collection of children attributions are represented as records instead of extensible records. Thus, instead of defining the labels for the children of the productions, we define for each production a record with the children as fields. For example:

```
data Ch_Decls_Decl _name _val _rest
  = Ch_Decls_Decl { ch_name :: _name
                  , ch_val :: _val, ch_rest :: _rest }
```

In this case, the generic *knit* function cannot be used anymore and thus a specific *knit* function is generated for each production:

```
knit_Decls_Decl rule fc ip =
  let ec = Ch_Decls_Decl {{ }} {{ }} {{ }}
      (Fam ic sp) = rule (Fam sc ip) (Fam ec {{ }})
      sc = Ch_Decls_Decl
             ((ch_name fc) (ch_name ic))
             ((ch_val  fc) (ch_val  ic))
             ((ch_rest fc) (ch_rest ic))
  in  sp
```

Then, the semantic functions are also a bit different:

```
sem_Decls_Decl sn sv sr
  = knit_Decls_Decl atts_Decls_Decl
                    (Ch_Decls_Decl sn sv sr)
```

We cannot use the generic type-level function *Defs* to define inherited attributes. We must define a specific instance of *Defs* for each production:

```
instance (HExtend (LVPair att v2) ic2 ic'2
          , HExtend (LVPair att v3) ic3 ic'3)
  => Defs att nts
           (Ch_Decls_Decl v1                v2   v3  )
           (Ch_Decls_Decl (Record HNil) ic2  ic3 )
           (Ch_Decls_Decl (Record HNil) ic'2 ic'3)
where defs att nts vals ic
  = Ch_Decls_Decl (ch_name ic)
                  (att .=. ch_val  vals .*. ch_val  ic)
                  (att .=. ch_rest vals .*. ch_rest ic)
```

and adapt the rule definitions to the use of records. For example:

```
ienv_Decls_Decl = inhdef att_ienv nts_ienv $
  do
```

```
lhs    ← at lhs
name ← at ch_name_Decls_Decl
val    ← at ch_val_Decls_Decl
return Ch_Decls_Decl
  { ch_val_Decls_Decl  = [ ]
  , ch_rest_Decls_Decl = (name, val # att_sval)
                       : lhs # att_ienv }
```

4.3 Benchmarks

We benchmarked our optimizations against *AspectAG* and *UUAGC*, in order to analyze their performance impact. [3] Figures 9 and 10 show the effect of grouping attributes in a simple grammar represented by a binary tree. The y-axis represents the execution time (in seconds) and the x-axis the number of grouped attributes in a full tree with 15 levels. In Figure 9 we show the results for a system with twenty synthesized attributes. Figure 10 shows the results for twenty inherited attributes and one synthesized attribute to collect them. In both cases the effect of grouping attributes becomes clear; for a relative large number of attributes the grouping optimization achieves good speedups.

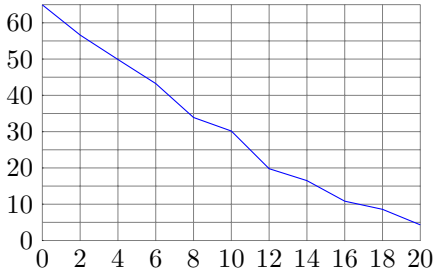

Figure 9: Grouping Synthesized Attributes

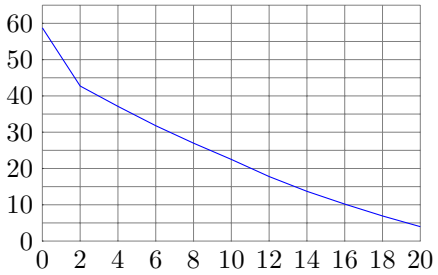

Figure 10: Grouping Inherited Attributes

In figures 11 and 12 we show the performance impact of the "static productions" optimization, as the number of children of the nodes increases. We tested with complete trees with depth 5; the x-axis represents the arity of the tree. Figure 11 shows the results for one synthesized attribute, while the results of Figure 12 include one synthesized and one inherited attribute. Thus, the optimization helps, and has a big impact on productions with many children.

In figures 13 and 14 we compared the performance of both optimizations and *AspectAG* in a simple grammar represented by a binary tree. In this case the x-axis represents the number

[3] Information avaliable at: `http://www.cs.uu.nl/wiki/bin/view/Center/Benchmarks`

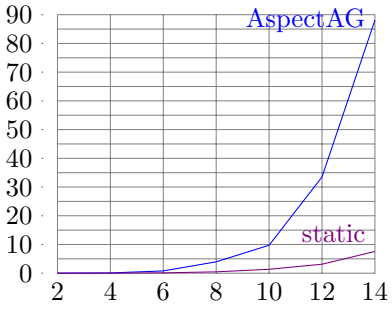

Figure 11: Static: Synthesized Attribute

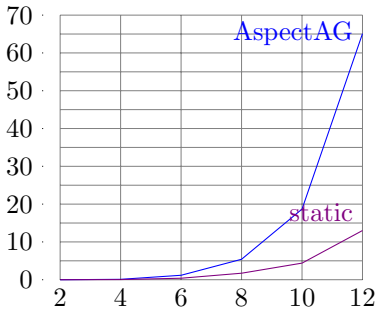

Figure 12: Static: Synthesized and Inherited Attribute

of (synthesized or inherited) attributes. As the number of attributes increases, the grouping optimization has a bigger performance impact. If we apply both optimizations together (figures 15 and 16) we obtain better times, although we are still quite far from the performance of *UUAGC*.

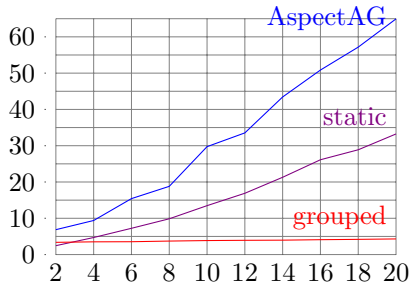

Figure 13: Static vs Grouped: Synthesized Attributes

If we apply both optimizations together (figures 15 and 16) we obtain better times, although we are still quite far from the performance of *UUAGC*.

5 Conclusions

We have shown how to generate flexible compilers, which can be easily extended by taking a hybrid approach to the architecture of a compiler: the core part is generated as a single monolithic part, which is evaluated efficiently, whereas extensions can be plugged in to this fixed part, albeit at a

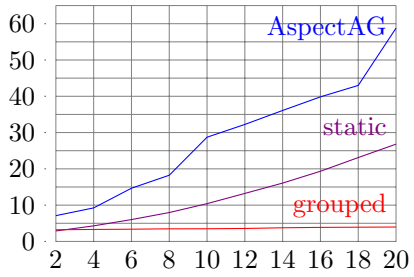

Figure 14: Static vs Grouped: Inherited Attributes

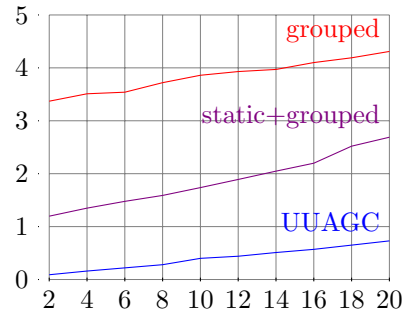

Figure 15: Static + Grouped: Synthesized Attributes

certain cost; we have constructed e.g. a syntax macro mechanism which makes it quite easy to extend the core compiler (parser and semantics) with new language constructs.

Summarizing, it can be seen that flexibility still has its cost, but the application of the optimizations is a good option as the number of attributes and/or children of the productions increases. One should keep in mind that the actual computations done in our examples in the rule functions is trivial. Hence in a real compiler, where most of the work is actually done in the rules, the overhead coming with the extra flexibility will be less burdening.

All code is available through the Hackage Haskell libraries.

References

[DFS09] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proc. of the 2nd ACM SIGPLAN symposium on Haskell*, pages 93–104. ACM, 2009.

[dMBS00] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

[dMPJW00] Oege de Moor, L. Peyton Jones, Simon, and Van Wyk, Eric. Aspect-oriented compilers. In *Proc. of the 1st International Symposium on Generative and Component-Based Software Engineering*, London, UK, 2000. Springer-Verlag.

[DS04] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

[Hal01] Thomas Hallgren. Fun with functional dependencies or (draft) types as values in static computations in haskell. In *Proc. of the Joint CS/CE Winter Meeting*, 2001.

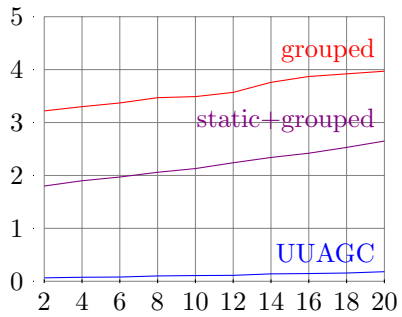

Figure 16: Static + Grouped: Inherited Attributes

[Jon99] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proc. of the ACM SIGPLAN workshop on Haskell*. ACM Press, 2004.

[McB02] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

[SO11] Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proc. of the 16th ACM SIGPLAN international conference on Functional programming*, pages 32–44. ACM, 2011.

[VSS09] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In *Proc. of the 14th ACM SIGPLAN international conference on Functional programming*. ACM, 2009.