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, mviera@fing.edu.uy
[2]Department of Computer Science, Utrecht University, Utrecht, The Netherlands, {doaitse,ariem}@cs.uu.nl

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*. The problem with such generative approaches 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 increased flexibility.

We show how to combine the two approaches by generating *AspectAG* code fragments from *UUAGC* sources, thus making it possible to trade between efficiency and flexibility, enabling a couple of optimizations for *AspectAG* resulting in a considerable speed improvement and making existing *UUAGC* code reusable 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 an aspect oriented 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.

Solutions to the quest for composable language description can be found at the textual level, as done by most attribute grammar systems [7, 3, 1], or at the semantic level, where language descriptions become first class values, which can be composed to build a complete language description.

The first approach is supported by, amongst many others, the Utrecht University Attribute Grammar System (*UUAGC*) [2], 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, leading to a large, monolithic and eficiënt compiler, which however cannot easily be adapted once generated and compiled. In the object-oriented world we find similar weaving based approaches in e.g. Lisa [7] and Jastadd [3], which are both Java based

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 [4, 8]. 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, non monad-based, approach to describing first-class language definition fragments; using a collection of combinators (the *AspectAG* Haskell package) it becomes possible to express attribute grammars using an Embedded Domain-Specific Language in Haskell [10]; unfortunately it is both a bit more verbose than the specific syntax as provided by the *UUAGC* system and relatively expensive. In order to provide the possibility to redefine attributes or to add new attributes elsewhere, we encode the lists of inherited and synthesized attributes of a non-terminal as an *HList*-encoded [5] 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* [1] some non-terminals have over 20 attributes), the cost of accessing attributes may become noticeable. Note that, in contrast to the weaving based approaches, this approach supports *separate compilation of individual aspects*: each generated fragment is individually checked to be well-typed and once compiled its source is not modifiable by other extensions.

In this paper we seek to alleviate the aforementioned verbosity and inefficiency by generating *AspectAG* code from the original *UUAGC* code. We furthermore take the opportunity to group collections of attributes which are not likely to be adapted, so we can shorten the *HList* values, thus relieving the costs of the extra available expressibility. Only the attributes which are 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 introduce 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 and in section 4 we describe how we optimize the generated code.

2 Attribute Grammars

2.1 Initial Attribute Grammars

```
LangDef.ag

  DATA Root  | Root decls : Decls main : Expr
  DATA Decls | Decl name : String val : Expr rest : Decls | NoDecl
  DATA Expr  | Add l : Expr r : 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 = l.sval + r.sval
           | Val  lhs.sval = value
           | Var  lhs.sval = case lookup var lhs.ienv of Just v → v

  ATTR Decls Expr INH ienv : [(String, Int)]
  SEM Root  | Root    decls.ienv = [ ]
                      main.ienv = decls.senv
  SEM Expr  | Add     l.ienv      = lhs.ienv
                      r.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

An Attribute Grammar is a context-free grammar where the nodes in the parse tree are decorated with a (usually quite large) number of values, called *attributes*. 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 the semantics in terms of *UUAGC* input. 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 semantics in the form of compilers and interpreters. We distinguish 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].

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 attribute (**SYN** *senv*) for passing the final environment back to the *Root* to 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 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 *ienv* list with a pair composed of the *name* used in the declaration and the value *sval* of the child with name *val* (*val.sval*).

Declaration of trivial rules, like the definitions of *ienv* for the production *Add*, are not necessary; they are automatically generated by *UUAGC* using so called *copy rules*, thus avoiding the need to write a lot of "boiler-plate" code.

When the *UUAGC* compiler weaves its input files into a Haskell program the rules' expressions are copied almost verbatim into the generated program: only the attribute references are replaced by references to values defined in 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

In this subsection we show how we can extend the given language *without touching the code written* (neither the generated nor the compiled code). In our compiler we want to generate error messages, so we introduce an extra synthesized attribute (*serr*, Figure 2), in which we report occurences of dual declarations (*name* is already an element of the *ienv*) and absent declarations (*name* is not an element of *ienv*).

To compile this code using *UUAGC* and *ghc* we 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, compile the composite result with *ghc* and link it with yet another call to *ghc*. Keep in mind that by doing so we only generate the semantic part of the compiler, which has to be completed with a few lines of main program containing the parsers from which refer to the generated semantic part.

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

```
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 = l.serr ++ r.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

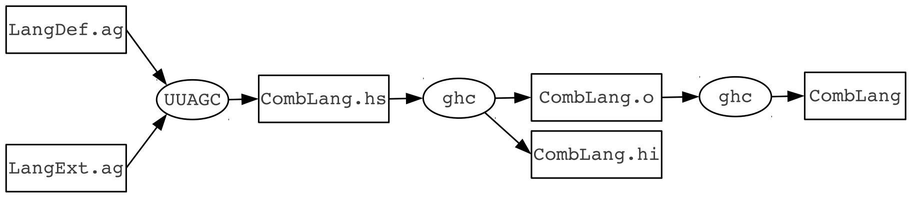

Figure 3: Compilation Process with UUAGC

To use *AspectAG* almost the same code has to be written, but by passing some extra flags to *UUAGC* we generate *human-readable AspectAG* code. This enables a completely different construction process (Figure 4), which makes it possible to 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. Such extensions could also be written in *AspectAG* directly.

To switch on this extension in *UUAGC* we pass the flag `--aspectag`:

```
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, which specify the computation of these attributes, end up in the `.ag` file (the implementation). Figure 5 shows the attribute grammar specification of Figure 1 adapted to our approach. Notice

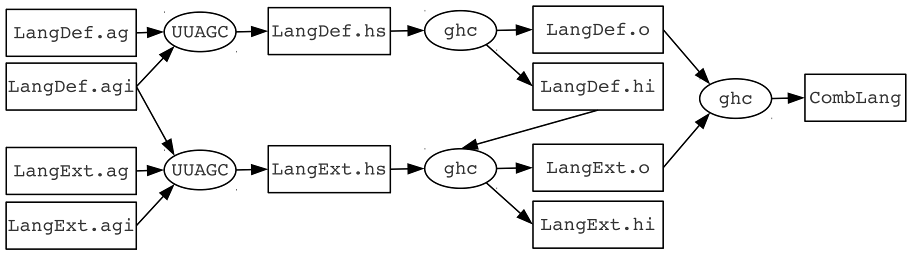

Figure 4: Compilation Process with our extension of UUAGC

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 specific rules for the attributes *sval*, *ienv* and *serr* have to be defined (in case they differ from the otherwise generated copy rules).

An important feature of the *AspectAG* library is that, besides adding new attributes or productions, existing definitions for attributes can be overwritten. 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 environment we pass to such right-hand side expressions. In Figure 8 we show how this can be done using := instead of =, the *UUAGC* syntactic form of *attribute redefinitions*.

3 From UUAG to AspectAG

The target of our translation is *AspectAG* [10], 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.

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* [5] typeful heterogeneous collections. *HList* is implemented using type-level programming techniques, where types are used to represent type-level values and classes are used to represent type-level types and functions [6]. The record *parent* represents the set of attributes for the parent and the record *children* is a type indexed 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.

```
LangDef.agi

  DATA Root  | Root decls : Decls main : Expr
  DATA Decls | Decl name : String val : Expr rest : Decls | NoDecl
  DATA Expr  | Add l : Expr r : 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  = l.sval + r.sval
            | Val     lhs.sval  = value
            | Var     lhs.sval  = case lookup var lhs.ienv of Just v → v

  SEM Root  | Root    decls.ienv = [ ]
                      main.ienv = decls.senv
  SEM Expr  | Add     l.ienv     = lhs.ienv
                      r.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

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 $Rule\ sc\ ip\ ic\ sp\ ic'\ sp' = Fam\ sc\ ip \to (Fam\ ic\ sp \to Fam\ ic'\ sp')$

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

$ext\ ::\ Rule\ sc\ ip\ ic'\ sp'\ ic''\ sp'' \to Rule\ sc\ ip\ ic\ sp\ ic'\ sp'$
$\quad \to Rule\ sc\ ip\ ic\ sp\ ic''\ sp''$
$(f\ `ext`\ g)\ input = f\ input\ .\ g\ input$

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. This is the job of *AspectAG*'s function *knit*, which takes a (composite) rule and a record containing the semantic functions

```
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 = l.serr ++ r.serr
            | Val     lhs.serr = [ ]
            | Var     lhs.serr = case lookup var lhs.ienv of
                                    Just _    → [ ]
                                    Nothing → [var ++ " undefined"]
```

Figure 6: Language Extension: Errors

```
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

of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes.

3.1 The Translation

The translation to *AspectAG* is quite straightforward. In the rest of this section we will show, with some examples, its most important aspects.

3.1.1 Grammar

Since we use extensible records, labels have to be generated to refer to the children of the productions of the grammar. For example, the child label generated out of the `.agi` file of Figure 7 is *ch_expr_Neg_Expr*. A label in an *HList* is represented by a plain Haskell value of a singleton type.

```
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

3.1.2 Attribute Definition

A collection of synthesized or inherited attributes is an extensible record, too. Thus, for each **ATTR** declaration in the `.agi` file, a label has to be generated to refer to the defined attribute. The declaration **ATTR** *Decls* **SYN** *senv* : $\{[(String, Int)]\}$, in Figure 6, generates the label *att_senv*.

The *AspectAG* function *syndef* adds the definition of a synthesized attribute. It constructs a rule *Rule sc ip ic sp ic sp′*, where *sp′* is the record *sp* extended with a field representing the new attribute. 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 in the record *sc* of the environment with type *Fam sc ip* using the label *ch_rest_Decls_Decl*. 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*. 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
```

$$
\begin{array}{ll}
\quad val & \leftarrow at\ ch\_val\_Decls\_Decl \\
\quad return & \{\{\ ch\_val\_Decls\_Decl\ \ .{=}.\ [\,] \\
 & \ ,\ \ ch\_rest\_Decls\_Decl\ .{=}.\ (name, val\ \#\ att\_sval) : lhs\ \#\ att\_ienv\ \}\}
\end{array}
$$

The parameter *nts_ienv* is a list of labels representing the non-terminals for which the attribute *ienv* is defined (generated out of the **ATTR** declarations). The function *lhs* returns the record *ip* (inherited attributes of the parent) from the input family *Fam sc ip*. The defined computations for each child are returned in an extensible record[2], which is iterated by a "type-level function" (implemented by a type class called *Defs*) to extend the corresponding records in *ic*.

3.1.3 Generating the Semantic Functions

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*:

$$atts\_Decls\_Decl = ienv\_Decls\_Decl\ `ext`\ senv\_Decls\_Decl$$

The semantic functions of the non-terminal *Decl* is:

$$
\begin{array}{ll}
sem\_Decls\_Decl & = knit\ atts\_Decls\_Decl \\
sem\_Decls\_NoDecl & = knit\ atts\_Decls\_NoDecl
\end{array}
$$

This code is generated out of the **DATA** declarations.

3.1.4 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.

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:

$$
\begin{array}{l}
atts\_Decls\_Decl = serr\_Decls\_Decl\ `ext`\ LangDef.atts\_Decls\_Decl \\
sem\_Decls\_Decl = knit\ atts\_Decls\_Decl
\end{array}
$$

[2]We use syntactic sugar {{...}} for extensible records equivalent to the list notation [...].

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 some attributes are fixed and will not be redefined, the use of extensible records is not necessary: in those cases we can group such a collection of synthesized attributes into a single attribute *att_syn* and such a collection of 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 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 be included in the grouping. For example, a following call to *uuagc* with flag `--nogroup=env` generates the *AspectAG* code for the example with all the attributes grouped except *ienv*, which will be redefined in the extensions.

4.2 Static Productions

If we do not need the possibility to change the definition of already existing productions (note that our flexible approach did not forbid this thus far), a less flexible approach to represent productions can also be taken. The flag `--static` activates an optimization where the collection of child 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 specific *knit* functions are generated for such productions:

```
knit_Decls_Decl rule fc ip = sp
   where 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))
```

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, 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 grammar represented by a binary tree. Note that these

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

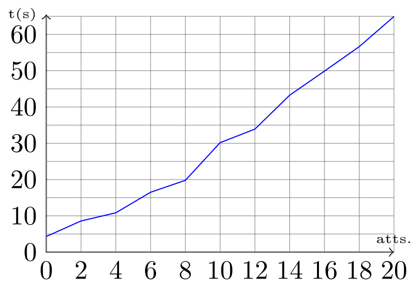

Figure 9: Grouping Syn. Attrs.

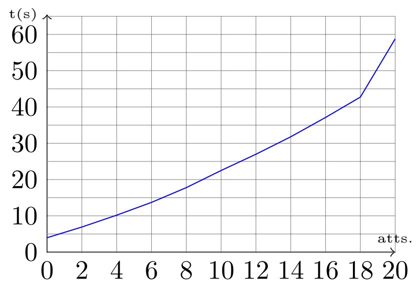

Figure 10: Grouping Inh. Attrs.

represent worst-case scenarios, since we hardly perform any work in the rules themselves. The y-axis represents the execution time (in seconds) and the x-axis the number of ungrouped attributes (the rest are grouped) 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 speedup, since whenever an attribute is needed it is located in constant time instead of linear time in the number of attributes.

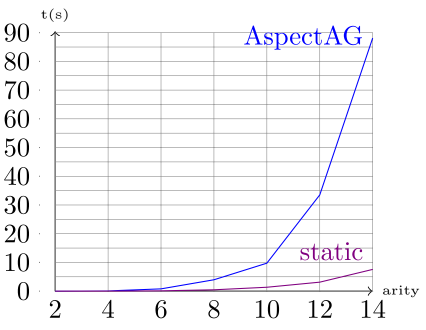

Figure 11: Static: Syn.

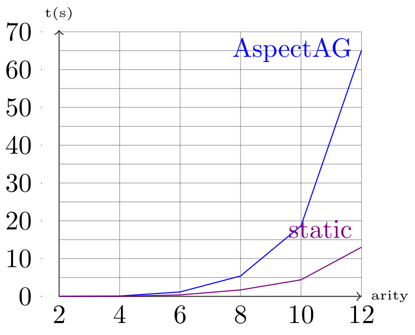

Figure 12: Static: Syn. and Inh.

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, because we are avoiding iterations over lists of children when evaluating the semantics for each node. In figures 13 and 14 we compared the performance of both optimizations and *AspectAG* in a simple grammar represented by a binary

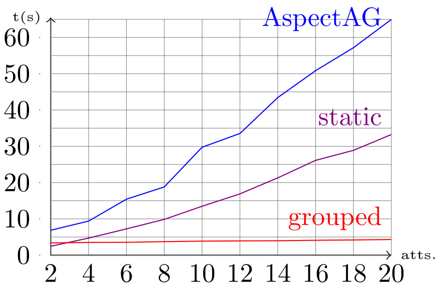

Figure 13: Static vs Grouped: Syn.

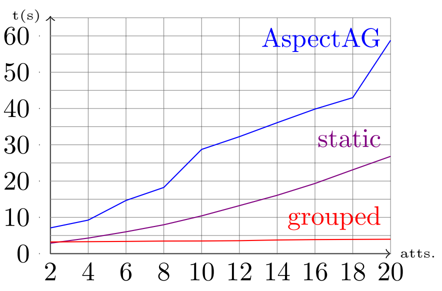

Figure 14: Static vs Grouped: Inh.

tree. In this case the x-axis represents the number 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*.

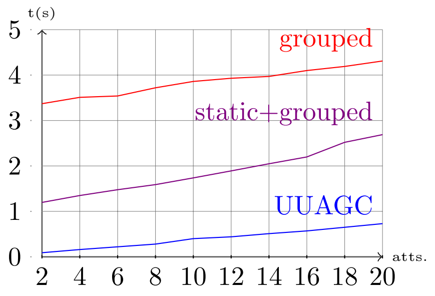

Figure 15: Static + Grouped: Syn.

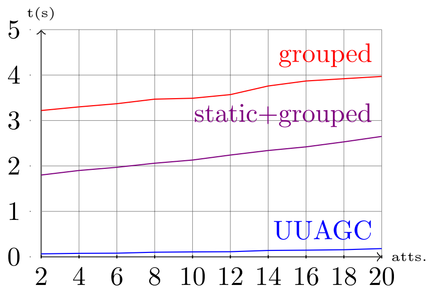

Figure 16: Static + Grouped: Inh.

5 Conclusions and Future Work

We have shown an approach to write *AspectAG* code, i.e. a strongly-typed flexible attribute system, in a less verbose Domain-Specific Language style.

We provide a framework for the generation of flexible compilers by taking a hybrid approach to their architecture. The core part is composed by a single monolithic part, which is evaluated efficiently, and a set of *redefinable* aspects. Extensions (and redefinitions) can be plugged into this core, albeit at a certain cost. In a companion paper [9] we proposed a syntax macros-like mechanism which follows this idea. The semantics of newly introduced syntax is defined in terms of already existing semantics. Some existing aspects, e.g. pretty-printing, may be redefined to provide accurate feedback.

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 usually be far less of a burden.

Possible future work is to add a new optimization, consisting in the generation of a less type-safe code than AspectAG. This involves the addition of a Haskell type-checking phase to UUAGC. Furthermore, a drawback of that approach is that it ties us to the use of *UUAGC*, not allowing the introduction of extensions written directly in the target language (e.g. AspectAG).

References

[1] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell 2009*, pages 93–104.

[2] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *AFP Summerschool*, number 3622 in LNCS, 2004.

[3] Görel Hedin and Eva Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47:37–58, April 2003.

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

[5] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004*. ACM Press.

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

[7] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Multiple attribute grammar inheritance. *Informatica (Slovenia)*, 24(3), 2000.

[8] Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP 2011*, pages 32–44. ACM.

[9] Marcos Viera and S. Doaitse Swierstra. Semantic Macros: Attribute Grammar Combinators. UU-CS 2011-028, Utrecht University, 2011.

[10] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In *ICFP 2009*. ACM.