submitted to ifl 2013

First Class Syntax, Semantics, and Their Composition

Marcos Viera
Instituto de Computación, Universidad de la República, Montevideo, Uruguay
mviera@fing.edu.uy

S. Doaitse Swierstra
Department of Computer Science, Utrecht University, Utrecht, The Netherlands
doaitse@cs.uu.nl

Abstract

Ideally complexity is managed by composing a system out of quite a few, more or less independent, and much smaller descriptions of various aspects of the overall artifact. When describing (extensible) programming languages, attribute grammars have turned out to be an excellent tool for modular definition and integration of their different aspects.

In this paper we show how to construct a programming language implementation by composing a collection of attribute grammar fragments describing separate aspects of the language. More specifically we describe a coherent set of libraries and tools which together make this possible in Haskell, *where the correctness of the composition is enforced through the Haskell type system*'s ability to represent attribute grammars as plain Haskell values and their interfaces as Haskell types makes this possible.

Semantic objects thus constructed can be combined with parsers which are constructed on the fly out of parser fragments and are also represented as typed Haskell values. Again the type checker prevents insane compositions.

Using a very small example language and some simple extensions, we show how all our techniques fit together towards the construction of extensible compilers out of a collection of pre-compiled, statically type-checked "language definition fragments".

Keywords Attribute Grammars,Typed Grammars,Typed Transformations,Haskell,Extensible Languages

1. Introduction

Since the introduction of the very first programming languages, and the invention of grammatical formalisms for describing them, people have been looking into how to enable an initial language definition to be extended by someone other than the original language designers. In the extreme case a programmer, starting from an empty initial language, could thus compose his favorite language out of a collection of pre-compiled language-definition fragments. Such language fragments may range from the definition of a simple syntactic abbreviation like list comprehensions to the addition of completely new language concepts, or even extensions to the type system.

In solving the problem of how to compose a compiler, various lines of attack have been pursued. The most direct and least invasive approach, which is so widely applied that one may not recognize it as an approach to the goal sketched above, is to make use of libraries defined in the language itself, thus simulating real extensibility. Over the years this method has been very effective, and especially modern, lazily evaluated, statically typed functional languages such as Haskell serve as an ideal environment for applying this technique; the definition of many so-called combinator libraries in Haskell has shown the effectiveness of this approach, which had been characterized as the construction of *embedded domain specific languages* (EDSL). The ability to define operators and precedences can be used to mimic syntactic extensions. Unfortunately not all programming languages really support this approach very well, given the flood of so-called modeling languages and frameworks from which lots of boilerplate code is generated.

At the other extreme of the spectrum we start from a base language and the *compiler text* for that base language. Just before the compiler is compiled itself, several extra ingredients can 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 [9] has shown the effectiveness of this approach using attribute grammars as the composing mechanism. This approach however is not very practical when defining relatively small language extensions; we do not want every individual user 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 we may also loose important guarantees provided by e.g. the type system of the language being defined; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

So the question arises of how we can do better than only providing powerful abstraction mechanisms without opening up the whole source of the compiler. The most commonly found approach is to introduce so-called *syntax-macros* [19], which enable the programmer to add syntactic sugar to a language by defining new notation in terms of already existing notation. Despite the fact that this approach may be very effective, it also has severe shortcomings; 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 is given in terms of invisible intermediate program representations. Hence the implementation details shine through, and error messages produced can be confusing or even incomprehensible.

Given the above considerations we impose some quite heavy restrictions on ourselves. In the first place extensions should go beyond merely syntactic extensions as is the case with the original syntax macros, which only map new syntax onto existing syntax; we want also to gain access to the part of the compiler which deals with the static semantics, e.g., in order to report errors in terms of the extended syntax instead of the original one. We seek extension at the semantic level, i.e. by using some sort of plug-in architecture; we will do so by constructing a core compiler as a collection of pre-compiled components, to which extra components can be added and for which existing components can be redefined at will. The questions we answer in this paper are how to compose a compiler out of *separately compiled and statically type checked* language-definition fragments and how to construct such fragments using a domain specific language embedded in Haskell.

The main contribution of this paper is to show how several related techniques we have previously developed can be combined in a unified approach to construct extensible compilers. The solution we present builds on:

- the introduction of a naming structure which makes it possible to represent mutually dependent structures and the possibility to manipulate such structures in a type-safe way [4]
- the description of typed grammar fragments as first class Haskell values [34], and the typed Left-Corner Transform to remove left-recursion [5]
- the possibility to construct self-analysing, error correcting parsers on the fly [27, 28]
- the possibility to deal with attribute grammars as first class Haskell values, which can be transformed, composed and finally evaluated [32, 33].

These techniques make use of many well-known Haskell extensions, such as multi-parameter type classes, functional dependencies, generalised algebraic data types and arrow notation. For simplicity, in the rest of the paper we will refer to this just as Haskell.

In Section 2 we introduce the syntax of a small language and its extension, as it is to be provided by the language definer and extender. In Section 3 we show the techniques we use to represent the syntax, and in Section 4 show the corresponding static semantics parts. We close by discussing related work, future work and present our conclusions.

2. Extensible Languages

In this section we show how to express extensible languages. The architecture of our approach is depicted in Figure 1; boxes represent (groups of Haskell) modules and arrows are **import** relations.

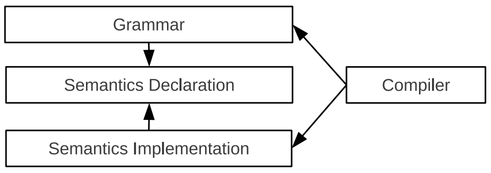

Figure 1. Initial Language

In the rest of the section we will take a detailed look at each module, and how everything fits together in the construction of a compiler. Our running example will be a small expression language with declarations. The purpose of this example is to introduce the main characteristics of out approach in an easy way. For a more involved example we refer to [31], the implementation of a compiler for the Pascal-like Oberon0 [37] language.

We will refer to our expression language as the *initial grammar*:

$$\begin{array}{lll} root & ::= & decls\ \texttt{"main"}\ \texttt{"="}\ exp \\ decls & ::= & var\ \texttt{"="}\ exp\ decls \mid empty \\ exp & ::= & exp\ \ \texttt{"+"}\ term\ \ \mid term \\ term & ::= & term\ \texttt{"*"}\ factor \mid factor \\ factor & ::= & int \mid var \end{array}$$

Note that this concrete grammar uses the syntactic categories exp, $term$ and $factor$ to represent operator precedences.

To implement this language fragment, a language implementer has to provide the Haskell code of Figure 2, expressing himself using our `murder`[1] combinator library (of course one might generate this from the grammar description) and the arrow-interface[2]. This corresponds to the module $Grammar$ in Figure 1. Without delving into details in this section, observe that the context-free grammar just given can be immediately recognized in the structure of the code. This grammar fragment description consists of a sequence of transformations, introducing new non-terminals to the grammar. The notation is to be read as $output \leftarrow transformation \prec input$. Each non-terminal (syntactic category) of the context free grammar is introduced (using $addNT$) by defining a list of productions (alternatives) separated by `<|>` (choice) operators, where each production contains a sequence of elements to be recognized.

The parameter sf is a record containing the "semantics of the language". The type of this record is declared in the module $Semantics\ Declaration$, for example:

$$\begin{array}{l} \mathbf{data}\ SemLang\ decls\ main\ rs\ name\ val\ rest\ ds\ nds \\ \qquad\qquad\quad al\ ar\ as\ ml\ mr\ ms\ value\ cs\ var\ vs \\ \quad = SemLang\ \{ \begin{array}[t]{ll} semRoot & :: decls \rightarrow main \rightarrow rs \\ ,\ semDecls & :: name \rightarrow val \rightarrow rest \rightarrow ds \\ ,\ semNoDecl & :: nds \\ ,\ semAdd & :: al \rightarrow ar \rightarrow as \\ ,\ semMul & :: ml \rightarrow mr \rightarrow ms \\ ,\ semCst & :: value \rightarrow cs \\ ,\ semVar & :: var \rightarrow vs\ \} \end{array} \end{array}$$

The functions contained in the record (accessed as e.g. $semMul\ sf$) describe how to map the semantic values associated with the abstract syntax trees corresponding to the non-terminals in the right-hand side of a production onto the semantic value of the left hand side of that production (and eventually the value associated with the root of a parse tree). We call these *semantic functions*, because they give meaning to the constructs of the language. As we will see in Section 4, the semantics of our simple expression language is composed by two aspects: pretty-printing and expression evaluation. The record is parametrized by the types that compose the types of its fields, i.e. the semantic functions of the productions. Such a record describes the abstract syntax of the language.

In Section 4 we show how to construct and adapt the semantic functions (module $Semantics\ Implementation$ in Figure 1) using the `uuagc`-system combined with a first-class attribute grammar library. We map the abstract parse tree of the program onto a call tree of the semantic function calls. The resulting meaning of a parse tree is a function which can be seen as a mapping from the inherited to the synthesized attributes. Thus, a production is defined by a semantic function and a sequence of non-terminals and terminals (`"*"`), the latter corresponding to literals which are to be recognized.

As usual, some of the elementary parsers return values which are constructed by the scanner. For such terminals we have a couple of predefined special cases, such as int, which returns the integer value from the input and var which returns a recognized variable name.

An initial grammar is also an *extensible grammar*. It exports (with $exportNTs$) its starting point ($root$) and a list of *exportable non-terminals* each consisting of a label (by convention of the form

[1] MUtually Recursive Definitions Explicitly Represented: `http://hackage.haskell.org/package/murder`

[2] Using Arrow syntax [24], which is inspired by the **do**-notation for $Monads$

```
gramIni sf = proc () → do
  rec root   ← addNT ≺ ⟦ (semRoot  sf) decls "main" "=" exp ⟧
      decls  ← addNT ≺ ⟦ (semDecls sf) var   "=" exp decls ⟧ <|> ⟦ (semNoDecl sf) ⟧
      exp    ← addNT ≺ ⟦ (semAdd   sf) exp   "+" term   ⟧ <|> ⟦ term ⟧
      term   ← addNT ≺ ⟦ (semMul   sf) term "*" factor ⟧ <|> ⟦ factor ⟧
      factor ← addNT ≺ ⟦ (semCst   sf) int ⟧ <|> ⟦ (semVar sf) var ⟧
  exportNTs ≺ exportList root $ export ntDecls decls . export ntExp     exp
                              .  export ntTerm term . export ntFactor factor
```

Figure 2. Initial Language Grammar

```
import Grammar        (gramIni)
import SemanticsImpl (semIni)
compiler = genCompiler (gramIni semIni)
```

Figure 3. Initial Language Compiler

nt...) and the collection of right hand sides. These right hand sides can be used and modified in future extensions.

Figure 3 contains a fragment of a (very simple) compiler of the example language; it corresponds to the module *Compiler* of Figure 1. The function *genCompiler* closes a grammar and generates a parser integrated with the semantics for the language starting from the first non-terminal, which in our case is *root*. The left-corner transform is applied to remove possible left recursion from the grammar, in order to use straightforward top-down parsing techniques in the actual parsing process.

2.1 Language Extension

The language (and thus compiler of that language) can be extended without having either to re-compile or to inspect the grammar and semantic components of the compiler for the initial language. Figure 4 shows the structure of a compiler produced as an extension of an initial language including the introduction of new syntax. In this case both the grammar and the semantics are being extended.

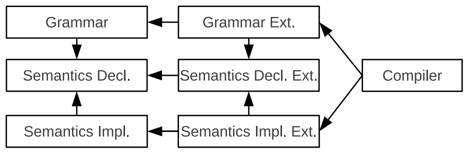

Figure 4. Language Extension

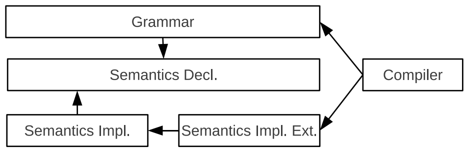

Figure 5. Lang. Semantics Modification

If the extension only involves modification of the semantics (e.g. to add new aspects or redefine existing ones), then it suffices to add an extension to the module containing the *Semantics Implementation* (Figure 5).

We compose compositions in an incremental way; i.e. any new language extension will be applied to the previously extended language.

In the rest of the section we show how to extend the language just defined by adding new kinds of expressions such as conditional expressions and new syntactic categories such as conditions:

```
factor ::= ... | "if" cond "then" exp "else" exp
cond   ::= exp "==" exp | exp ">" exp
```

The grammar extension *gramExt* is again defined as a Haskell value, which *imports* an existing set of productions and builds an extended set, as shown in Figure 7. In this case the type of the *sf*

```
gramExt sf = proc imported → do
  let exp    = getNT ntExp    imported
  let factor = getNT ntFactor imported
  rec addProds ≺ (factor, ⟦ (semIf  sf) "if"   cond
                                         "then" exp
                                         "else" exp ⟧)
      cond  ← addNT  ≺ ⟦ (semEq sf) exp "==" exp ⟧
                   <|> ⟦ (semGr sf) exp ">"  exp ⟧
  exportNTs ≺ extendExport imported
                           (export ntCond cond)
```

Figure 6. Language Extension : If

record, defined in the module *Semantics Declaration Extension*, is:

```
data SemLangExt cnd thn els is el er es gl gr gs
  = SemLangExt {semIf  :: cnd → thn → els → is
               , semEq :: el → er → es
               , semGr :: gl → gr → gs}
```

We first show how to combine previously defined productions with the newly defined productions into an extended grammar: for each non-terminal to be extended, or used in an extension, we retrieve its list of productions (using *getNT*) from the *imported* non-terminals, and add new productions to this list using *addProds*. For example, for *factor* the new **if** ... **then** ... **else**... production is added by:

```
let exp    = getNT ntExp    imported
let factor = getNT ntFactor imported
addProds ≺ (factor, ⟦ (semIf sf) "if" cond
                                 "then" exp
                                 "else" exp ⟧)
```

Extra non-terminals can be added as well using *addNT*; in the example we add the non-terminal *cond* with its two productions to represent some simple conditions:

```
cond ← addNT ≺ ⟦ (semEq sf) exp "==" exp ⟧
           <|> ⟦ (semGr sf) exp ">"  exp ⟧
```

Finally, we extend the list of exportable non-terminals with (some of) the newly added non-terminals, so they can be extended by further fragments elsewhere:

```
exportNTs ≺ extendExport imported (export ntCond cond)
```

```
gramExt sf = proc imported → do
  let exp    = getNT ntExp    imported
  let factor = getNT ntFactor imported
  cond ← addNT  ≺   ⟦ (semEq sf) exp "==" exp ⟧
               <|>  ⟦ (semGr sf) exp ">"  exp ⟧
  addProds  ≺ (factor, ⟦ (semIf  sf) "if"   cond
                                     "then" exp
                                     "else" exp ⟧)
  exportNTs ≺ extendExport imported
                               (export ntCond cond)
```

Figure 7. Language Extension : If

Some extensions may require to modify some already defined productions of the grammar to extend. For example, suppose we want to extend our language with some unary operators **sq** (square), **pyth** (sum of squares) and **db** (double), whose precedence is in between *term* and *factor*. This can be done by adding a new syntactic category *unary*, for the new operators, and updating the references to *factor* in the productions of *term* to point to *unary*. Figure 8 shows how to implement this extension. The transformation *updProds* updates the productions of an existing non-terminal (e.g. *term*) by a given function. The function *mapNTProds factor unary* maps every reference to *factor* in the productions to point to *unary*.

```
gramExt2 sf = proc imported → do
  let term   = getNT ntTerm   imported
  let factor = getNT ntFactor imported
  unary ← addNT ≺ ⟦ (semSq   sf) "sq"   factor ⟧
              <|> ⟦ (semPyth sf) "pyth" factor ⟧
              <|> ⟦ (semDb   sf) "db"   factor ⟧
  updProds  ≺ (term, mapNTProds factor unary)
  exportNTs ≺ extendExport imported
                               (export ntUnary unary)
```

Figure 8. Language Extension : Sq, Pyth and Db

Because *gramIni*, *gramExt* and *gramExt2* are all proper Haskell values, which are separately defined in different modules which can be compiled separately, we claim that the term *first-class grammar fragments* is justified.

The extended language compiler is shown in Figure 9. The (left associative) operator (+>>) composes an initial grammar with its extension, returning a new (initial) grammar. The function *genCompiler* makes sure that all existing references to non-terminals eventually refer to the final version of the definitions for these non-terminals.

```
import GrammarExt2 (gramIni, gramExt, gramExt2)
import SemanticsImplExt2 (semIni, semExt, semExt2)
compiler = genCompiler (gramIni   semIni +>>
                        gramExt   semExt +>>
                        gramExt2  semExt2)
```

Figure 9. Extended Language Compiler

3. First-Class Syntax

In this section we introduce the `murder` library, which we use to define and combine grammars. The library is based on the typed representation of grammars and the typed transformations [4] of these grammars.

3.1 Grammar Representation

We use a representation of grammars as typed abstract syntax [5] based on the use of Generalized Algebraic Data Types [25]. The idea is to indirectly refer to non-terminals via references encoded as types. Such references type-index into an environment holding the actual collection of productions for non-terminals. This enforces that the productions occurring in an environment can only contain references to non-terminals that belong to the environment in question. A grammar is a value with type *Grammar a*, where the type *a* is the type of a witness of a complete successful parse starting from the root nonterminal of the grammar.

3.2 Grammar Extensions

Grammar definitions (Figure 2) and extensions (Figure 7) are typed transformations of values of type *Grammar*, implemented using the library `TTTAS`[3] (Typed Transformations of Typed Abstract Syntax). `TTTAS` enables us to represent typed transformation steps, (possibly) extending a typed environment. In other words, by using typed transformations when adding non-terminals and productions to a grammar, we will always construct a grammar that is assured to be well-typed again.

In `TTTAS` the transformations are represented as *Arrows* [14]. *Arrows* are a generalization of *Monads*, modeling a computation that takes inputs and produces outputs. In our case the computation maintains a state containing the environment mapping the non-terminals of the grammar onto the thus far defined productions. We use the input and output of the arrows to read and write data controlling the transformation process.

Both extensible grammars and grammar extensions have to export their starting point and their list of *exportable non-terminals*, which can be used and/or modified by future extensions. We encode this data in a value of type *Export*, which is constructed using the function *exportList*, and extended with *extendExport*.

The only difference between extensible grammars and grammar extensions is that a grammar extension has to import the list of non-terminals it will extend, while an initial grammar does not import anything.

Thus, the definition of an extensible grammar, like the one in Figure 2, has the following shape:

```
gramIni = proc () → do ...
                          exportNTs ≺ exported
```

[3] `http://hackage.haskell.org/package/TTTAS`

where the **proc** () part indicates that $gramIni$ is a typed transformation that takes just () as input and returns as output a value ($exported$) of type $Export$. With $exportNTs$ we inject the $Export$ value in the transformation in order to return it as output.

The definition of a grammar extension, like the one in Figure 7, has the shape:

$$gramExt = \mathbf{proc}\ imported \rightarrow \mathbf{do}\ ... \qquad exportNTs \prec exported$$

Now in order to extend a grammar with a grammar extension all we have to do is to compose both transformations by connecting the output of the first to the input of the second. This is the role of the operator (+>>), used in Figure 9, which is a (type) specialized version of the $Arrow$'s composition (>>>).

To add a new non-terminal to the grammar boils down to adding a new term to the environment using the transformation $addNT$. The input to $addNT$ is the initial list of alternative productions for the non-terminal and the output is a non-terminal symbol, i.e. the index of the newly added non-terminal in the new grammar. Thus, when in Figure 2 we write:

$$exp \leftarrow addNT \prec productions$$

we are adding the non-terminal for the expressions, with the list of productions $productions$ passed as a parameter, and we bind to exp a symbol holding the reference to the added non-terminal so it can be used in the definition of this or other non-terminals. The list of alternative $productions$ is expressed in an applicative style; i.e. in terms of $pure$, (<*>), (<*) and (<|>), or using the brackets $[\![$ and $]\!]$. These brackets are inspired by the $idioms$ approach as introduced by McBride [21]. The brackets $[\![$ and $]\!]$ are the LaTeX representations of the Haskell identifiers *iI* and *Ii*, which come with a collection of Haskell class and instance declarations which together allow us to write $[\![\ (semMul\ sf)\ term$ "*" $factor\ ]\!]$ instead of the more elaborate text:

$pure\ (semMul\ sf)$ <*> $sym\ term$ <* tr "*" <*> $sym\ factor$

Adding new productions to an existing non-terminal boils down to appending the extra productions to the list of existing productions of that non-terminal. Figure 7 contains an example of adding a production to the non-terminal $factor$. The transformation $addProds$ takes as input a pair with a reference to the non-terminal to be extended and the list of productions to add:

$$addProds \prec (nonterminal, productions)$$

In this case the output is irrelevant, since no new references are created as a result of this extension.

3.3 Closed Grammars

We can run the transformation by closing the grammar; i.e. all references are made to point to the latest version of their corresponding non-terminals. Thus, a call to $closeGram$ starts with an empty grammar, and applies to it all the transformations defined in the grammar description to obtain the defined grammar.

$$genCompiler = (parse\ .\ generate\ .\ leftcorner)\ closeGram$$

The type of a closed grammar is $Grammar\ \ a$, where a is a phantom type [13] representing the type of the start non-terminal.

Since this grammar can be left-recursive we have to apply the $leftcorner$ [5] typed transformation in order to remove potential left-recursion:

$$leftcorner :: Grammar\ a \rightarrow Grammar\ a$$

The function $generate$ generates a parser integrated with the semantics for the language starting from the first non-terminal, which in our case is $root$.

$$generate :: Grammar\ a \rightarrow Parser\ Token\ a$$

Finally, $parse$ parses the input program while computing the meaning of that program. Currently we can generate either `uulib`[4] or `uu-parsinglib`[5] parsers.

$$parse :: Parser\ Token\ a \rightarrow [\,Token] \rightarrow ParseResult\ a$$

4. First-Class Semantics

In this section we complete the example by showing how we use attribute grammars to define the static semantics of the initial language and how such definitions can be redefined when the language is extended.

An Attribute Grammar describes for a context-free grammar how each node in a parse tree is to be decorated with a collection of values, called *attributes*. For each attribute we have a defining expression in which we may refer to other "nearby" attributes, thus defining a data-flow graph based on the abstract syntax tree. An attribute grammar evaluator schedules the computation of these expressions, such that the attributes we are interested in eventually get computed.

4.1 Definition of the Language Semantics

To define the static semantics of a language we use the `AspectAG`[6] embedding of attribute grammars in Haskell. In order to be able to redefine attributes or to add new attributes later, it encodes the lists of inherited and synthesized attributes of a non-terminal as an $HList$-encoded [17] value, indexed by types using the Haskell class mechanism. In this way the *closure test* of the attribute grammar (each attribute has exactly one definition) is realized through the Haskell class system. Thus, attribute grammar fragments can be individually type-checked, compiled, distributed and composed to construct a compiler. Albeit easy to use for the experienced Haskell programmer, it has a rather steep learning curve for the uninitiated. A further disadvantage is that the approach is relatively expensive: once the language gets complicated (in our Haskell compiler `UHC` [9] some non-terminals have over 20 attributes), the cost of accessing attributes may eventually overshadow the cost of the actual computations.

For those reasons in [35] we have defined an extension to the `uuagc` compiler [29], that generates `AspectAG` code fragments from original `uuagc` sources. This tool enables a couple of optimizations to the `AspectAG` code: we limit both our reliance on the $HList$-encoding, resulting in a considerable speed improvement, and allow existing uuagc code to be reused in a flexible environment.

With the `--aspectag` option we make `uuagc` generate `AspectAG` code out of a set of `.ag` files and their corresponding `.agi` files. 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).

In the rest of the paper we will show examples written in the `uuagc` language. Although another valid option would have been to implement the semantic functions directly in `AspectAG`, or to use a hybrid approach.

Figure 10 shows the `.agi` file for the semantics of our initial language. Notice that the grammar defined here is not exactly the same as the context-free grammar of the language, since our attribute grammars are built on top of the *abstract syntax* of the language. We define attributes for the following aspects: pretty printing, realized by the synthesized attribute spp, which holds a pretty

[4] `http://hackage.haskell.org/package/uulib`

[5] `http://hackage.haskell.org/package/uu-parsinglib`

[6] `http://hackage.haskell.org/package/AspectAG`

```
SemanticsImpl.agi
  DATA Root  | Root decls : Decls main : Expr
  DATA Decls | Decl name : String val : Expr rest : Decls
             | NoDecl
  DATA Expr  | Add al : Expr ar : Expr
             | Mul ml : Expr mr : Expr
             | Cst value : Int
             | Var var : String


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

Figure 10. Language semantics

printed document of type PP_Doc, and expression evaluation, realized by the synthesized attribute $sval$ of type Int, which holds the result of an expression, and an inherited attribute $ienv$ which holds the environment ($[(String, Int)]$) in which an expression is to be evaluated. *Synthesized attributes* take their definition "from below", using the values of the synthesized attributes of the children of the node the attribute is associated with and the inherited attributes of the node itself. An *inherited attribute* is defined "from above": in its defining expression we may refer to the inherited attributes of its parent and the synthesized attributes of its siblings.

Keep in mind that we chose these trivial semantics in order to keep the example simple, and focus on the features of the technique. A real compiler should involve more complex tasks such as type-checking, optimization and code generation.

Figure 11 shows the `.ag` file including the implementation of the attributes declared above. In a **SEM** block we specify how attributes of a production are to be computed out of the attributes from the left hand side and children of the production. The defining expressions at the right hand side of the = signs are almost plain Haskell code, using minimal syntactic extensions to refer to attributes. 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 $\mathbf{lhs}.attribute$. 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 ($val.sval$).

The pretty-printing attribute is defined for each production by combining the pretty printed children using the pretty printing combinators from the `uulib` library: (`<#>`) for horizontal (beside) composition, (`<->`) for vertical (above) composition, and pp to pretty print a string.

The semantics of the expression evaluation ($sval$) is intuitive. Variables of the main expression are located in an environment constructed as follows:

- the declarations sub-tree ($decls$) receives an empty environment $ienv$ and extends it through the list of declarations with the values resulting from the evaluation of the expression in the right hand side of each declaration
- the complete environment is passed "up" to the root in the attribute $senv$
- this environment is distributed into the $main$ expression as $ienv$

```
SemanticsImpl.ag
 SEM Root
   | Root    lhs.spp  = decls.spp <->
                        "main =" <#> main.spp
 SEM Decls
   | Decl    lhs.spp  = name <#> "=" <#> val.spp <->
                        rest.spp
   | NoDecl  lhs.spp  = empty
 SEM Expr
   | Add     lhs.spp  = al.spp <#> "+" <#> ar.spp
   | Mul     lhs.spp  = ml.spp <#> "*" <#> mr.spp
   | Cst     lhs.spp  = pp (show value)
   | Var     lhs.spp  = pp var

 SEM Root
   | Root    lhs.sval = main.sval
 SEM Expr
   | Add     lhs.sval = al.sval + ar.sval
   | Mul     lhs.sval = ml.sval * mr.sval
   | Cst     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 Decls
   | Decl    val.ienv   = []
             rest.ienv  = (name, val.sval) : lhs.ienv

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

Figure 11. Language semantics

The rules to describe the computation of the attribute $ienv$ for the productions Add and Mul of the non-terminal $Expr$ are omitted. In this case, rules that copy the attribute (unchanged) to the children are inserted automatically by `uuagc`. The library `AspectAG` includes a function $copy$ that implements the same behaviour.

Notice that the expressions of the declarations ($Decl$) should be closed, since they are (in our current definition) evaluated in an empty environment.

A semantic function (sem_Prod) is generated for each production ($Prod$) of the grammar. Thus, to complete our initial language of Section 2 we only need to construct the record $semIni$ with these semantic functions:

```
semIni = SemLang { semRoot    = sem_Root
                 , semDecls   = semDecls
                 , semNoDecl  = sem_NoDecl
                 , semAdd     = sem_Add
                 , semMul     = sem_Mul
                 , semCst     = sem_Cst
                 , semVar     = sem_Var }
```

```
SemanticsImplExt.agi
  EXTENDS "SemanticsImpl"
  ATTR Root Decls Expr SYN serr
                           USE {++} {[]} : [String]
```

```
SemanticsImplExt.ag
  SEM Decls | Decl lhs.serr
     = (case lookup name lhs.ienv of
              Just _    → [name ++ " duplicated"]
              Nothing → [ ]) ++ val.serr ++ rest.serr
  SEM Expr | Var lhs.serr
     = case  lookup var lhs.ienv of
              Just _    → [ ]
              Nothing → [var ++ " undefined"]
```

Figure 12. Semantics Extension: Errors

4.2 Extending the Semantics

Having first-class attribute grammars enables us to have a compiled definition of the semantics of a language and to introduce relatively small extensions to it later, without the need to either reconstruct the whole compiler, or to require the sources of the core language to be available.

In this subsection we show, by using some simple examples, how extensions can be defined.

The use of variables and declarations in the example language can be erroneous. Thus we introduce in Figure 12 an extra synthesized attribute ($serr$) in which we collect error messages corresponding to duplicated definitions and referring to undefined variables. This extension to the language corresponds to the kind of extensions described in Figure 5, because it only involves a change at the semantic level; no new syntax is added.

The keyword **EXTENDS** in Figure 12 is used to indicate which attribute grammar is being extended. The **USE** clause included in the declaration of the synthesized attribute $serr$ indicates that, for the productions where the definition is omitted, the attribute will be computed by *collecting* the synthesized attributes $serr$ of the children of the production. If the collection is empty ($NoDecl$ and Cst) the value of the attribute is $[\,]$. In the other case ($Root$, Add and Mul) the values are combined with the operator $(+\!\!+)$. The same can be done in AspectAG using the function use.

In Section 2.1 we extended the initial language with a conditional expression. The implementation of the semantics of this extension, which corresponds to the extensions depicted in Figure 4, is shown in Figure 13. In this case not only new attributes are added, but we also extend the abstract syntax with a new kind of node, and define a new production for the existing non-terminal $Expr$. Since semantics extensions are pairwise incremental, we also have to define the computation of the attribute $serr$ for the newly included productions.

AspectAG enables the *redefinition* of already existing attributes. In uuagc (extended to generate AspectAG) we use $:=$, instead of $=$, to declare attribute redefinitions. For example, in the extension of Figure 14, the attribute $ienv$ is redefined to allow the use of variables in the expressions of the declarations. Notice how a very small change to the attribute grammar definitions may influence the overall language considerably.

Usually we do not want to define the complete semantics of a syntactic extension from scratch. If we limited ourselves to a syntax-macro like mechanism, where new syntax is mapped onto existent syntax, it would be useful to have a way to express this mapping at the semantic level. In [33] we extended AspectAG with

```
SemanticsImplExt2.agi
  EXTENDS "SemanticsImplExt"
  DATA Expr | If cnd : Cond thn : Expr els : Expr
  DATA Cond | Eq el : Expr er : Expr
            | Gr gl : Expr gr : Expr
  ATTR Cond SYN sval : Bool
  ATTR Cond SYN spp  : PP_Doc
  ATTR Cond SYN serr USE {++} {[]} : [String]
```

```
SemanticsImplExt2.ag
  SEM Expr | If   lhs.sval = if cnd.sval then thn.sval
                                         else  els.sval
  SEM Cond | Eq lhs.sval = el.sval ≡ er.sval
                lhs.spp  = el.spp <#> "==" <#> er.spp
           | Gr lhs.sval = gl.sval > gr.sval
                lhs.spp  = gl.spp <#> ">" <#> gr.spp
```

Figure 13. Semantics Extension: If

```
SemanticsImplExt3.agi
  EXTENDS "SemanticsImplExt2"
```

```
SemanticsImplExt3.ag
  SEM Decls | Decl val.ienv := rest.senv
```

Figure 14. Semantics Extension: Variables in declarations

```
SemanticsImplExt4.agi
  EXTENDS "SemanticsImplExt3"
  DATA Expr
     | Sq se : Expr              ⇒ (Mul se se)
     | Pyth pl : Expr pr : Expr  ⇒ (Add (Sq pl) (Sq pr))
     | Db de : Expr              ⇒ (Mul (Cst 2) de)
```

Figure 15. Semantics Extension: Sq, Pyth and Db

an $agMacro$ combinator that enables us to define the attribute computations of a new production in terms of the attribute computations of existing productions. Thus, we can define the extensions Sq, computing the square of an expression, $Pyth$ for the sum of the squares of two expressions, and Db to double an expression as in Figure 15. The fragment $Sq\ se : Expr \Rightarrow (Mul\ se\ se)$ defines a production Sq with a child se, where the computation of its semantics is based on the computation of the semantics of the production Mul, but mapping both children to se. In the case of $Pyth$, macros are used recursively to define the mapping of the children of the production Add.

Sometimes we will need to define a special semantics for certain attributes of the production. For example, with the definition of Sq of Figure 15, if we pretty print the expression **pyth** 3 4 the result will be $3 * 3 + 4 * 4$, since that was the abstract syntax tree to which it was mapped in order to compute its semantics. This however is likely not to be the desired behavior. Fortunately we are able to redefine attribute computations in such cases! Thus, in the corresponding `.agi` file (Figure 16) we redefine the semantics of the pretty printing aspect.

```
SemanticsImplExt4.ag

  SEM Expr
    | Sq   lhs.spp := "sq" <#> se.spp
    | Pyth lhs.spp := "pyth" <#> pl.spp <#> pr.spp
    | Db   lhs.spp := "db" <#> de.spp
```

Figure 16. Pretty printing redefinition

5. Related Work

Although syntax extensions are not commonly supported in typed languages, there is a long tradition in languages like Lisp [36], Scheme [2], Prolog [1], and more recently Stratego [6]. For these syntactically very parsimonious languages a pressing need for such a facility exists, and the absence of a rich type system does not provide a burden for its implementation. We quote Fisher and Shivers [11] who say *"Once one has become accustomed to such a powerful tool, it is hard to give up. When we find ourselves writing programs in languages such as Java, SML, or C, that is, that lack Scheme's syntax extension ability- we find that we miss it greatly"*. Having made this observation they introduce the Ziggurat [12] system, which aims at the same goal as this paper; the underlying technology is completely different though. They use a delegation based system with which the semantics associated with the node in an abstract syntax tree can be updated. By using Lisp as their implementation language they do not have to cope with the problems posed by the Haskell type system; on the other hand the users of the Ziggurat system do not have the advantages associated with having a typed implementation language. We believe that having a statically typed implementation language is a great advantage, and we happily rephrase the above quote: *"Once one has become accustomed to the advantages of a static type system, it is hard to give up. When we find ourselves writing programs in languages such as Lisp, PHP, Ruby and JavaScript, that lack Haskell's type and class system- we find that we miss it greatly"*.

Another distinguishing feature is that our underlying technology for describing the static semantics is based on attribute grammars. Attribute grammars have proven themselves extremely useful for compositional language definitions. Adams [3] proposed a set of tools for modular syntax and modular attribute grammars in an untyped setting. Among many others, the attribute grammars systems LISA [22], JastAdd [10], Silver [30] and Kiama [26], have successfully tackled the problem of defining modular extensible compilers in a typed context.

Most of these systems, like `uuagc`, have a generative approach to compositionality; i.e. take the sources of all the composing modules and generate a monolithic system in a host language. Therefore, they do not provide separate compilation. An exception is Kiama, which is embedded as a library in Scala, and supports composition by using *mixins* and *traits*. From this point of view, Kiama is closely related to `AspectAG`, although the former is not able to perform well-formedness checks (such as the closure test) to a composed grammar, unless the grammar is declared as non-extensible. The design of `AspectAG` is inspired by [8], which represents attributions using Rémy-style records, instead of the type-level programming techniques.

LISA and Silver include parser generators to construct parsers out of the composed grammars. Since we do not have access to the source of the composing grammars, we use typed grammar transformations and parser combinators to generate (left-recursion free) top-down parsers *on the fly*. Neither Kiama nor JastAdd provides support for concrete syntax specification and parsing.

All the systems support synthesized and inherited attributes, but some of them extended the model with some new features. Silver includes *forwarding*, to allow productions to implicitly define the computation of some attributes by translation. This functionality is very similar to the provided in `AspectAG` by the combination of *agMacro*s and attribute redefinitions. JastAdd and Kiama support *reference attributes*, i.e. attributes that refer to other tree nodes. This is useful in writing compilers, because it allows one to model language relations (such as the use and declaration of variables and types) as references inside the abstract syntax tree. We do not support this kind of attribute.

Finally, we use Haskell, a strongly-typed pure functional programming language, to define the attribute computations. We think it fits perfectly to the declarative nature of attribute grammars. In cases where imperative languages like Java (JastAdd, LINDA) are used, it becomes impossible to control the absence of side-effects. Silver defines its own language which is declarative and strongly-typed, although more limited.

6. Conclusions and Future Work

With the combination of the techniques we have developed over the years our dream is close to becoming true: the possibility to construct a complete compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments.

We tackled the problem of how to construct a composable compiler both at syntactic and semantic level.

By dealing with context-free grammars as Haskell values we are able to compose, analyze and transform them (to apply for example the left-corner transform) to construct efficient parsers. With the use of typed transformations we mantain a type correct representation of the grammars during the transformation processes. It is important to point out that grammar fragments are combined on the fly; i.e. after they have been compiled.

At the semantic level, by using first-class attribute grammars, attribute computations can be defined and composed; different attributes can be stored in different modules, compiled and then composed. An importan characteristic of this approach is that attribute computations can also be redefined, in order to specialize (or modify the behaviour of) parts of already defined (and compiled) semantics.

With the combination of techniques described in this paper we have established a firm bridge-head. So what problems are left and how should we proceed from here?

In the first place the organization of the collection of attributes in a linear structure, such as `HList` is costly. It is our experience however that a compiler spends most of its time in the auxiliary code for type-checking and -inferencing and (global) optimization. Thus for a modest language defined by a limited set of attributes we think the approach is not prohibitively costly. For more complicated languages, which use many attributes for their definition, there are several ways to alleviate this problem. Most attributes are not defined in isolation since most aspects are described using a collection of attributes. This is something we can exploit; do not place all attributes in a single linear `HList`, but group them in an tree-like structure [20], thus lowering the nesting depth of the top `HList` products.

Building the complete compiler from scratch as a collection of syntax extensions and fine-grained aspect definitions is probably not always the optimal approach; large parts of the compiler will be shared by all users, and there is no reason to use the relatively expensive techniques enabling extensibility all over the compiler, as long as the core compiler remains extensible. In this way we plan to define an extensible Haskell compiler, where the already existing attribute-grammar based description of `UHC` can be used to generate such an extensible core compiler. Therefore we provide default definitions for all aspects, each of which can be redefined.

An additional benefit of this approach is that we prevent unwanted or illogical combinations of aspects. For example, we may inhibit circumvention of the basic type-checking part of the compiler by simply not exporting that part of the interface.

A second point for improvement is the way attribute evaluation is scheduled. In the description above we use a very straightforward approach which uses Haskell's lazy evaluation; a tree attribution is seen as a single large data flow graph, with attributes in the nodes and semantic functions for defining the values of the nodes [7, 8, 15, 18]. Unfortunately this elegant approach breaks down when large trees are to be attributed; a lazy evaluation scheduling first builds a large dependency graph in memory, and only starts doing some real work when this large graph has been constructed. This resembles the application of function *foldr* to a very long list, usually remedied by using $foldl'$ instead. Unfortunately there is no similar simple transformation which alleviates this problem for an arbitrary attribute grammar, since this requires a global flow analysis of the attribute dependencies [16]. However, the `UUAGC` already performs these analyses and can generate strict implementations containing explicitly scheduled code, and thus an efficient version for the sketched core compiler can be generated. Interfacing with this core compiler will be a bit more cumbersome, since the dependencies between the attributes now have become visible. Since these dependencies usually reflect the way the compiler programmer thinks about his attribute grammars [23] we expect this extra burden to be bearable.

A third problem arises from the way we construct our parsers and combine our aspects. With the current Haskell implementations *every time* we use the compiler the complete parser and attribute grammar is reconstructed from scratch; the individual grammar components are constructed first (*gramIni* and *gramExt*), then they are merged into a single large grammar (the calls to `+>>`) and references are resolved (*closeGram*); subsequently this large grammar is analysed and subjected to the Left-Corner Transform, and finally out of this resulting grammar the actual parser is constructed. A similar sequence of steps is done for the aspects. The final parser and evaluator, however, do not depend on the input of the compiler; they are global constant Haskell values; i.e. are in constant applicative form (CAF). Having such values repeatedly being constructed is not a problem of our approach alone, but occurs whenever some form of composition, analysis and transformation is taking place. We expect this to occur more often once the expressiveness of our techniques become more widely known and we think this problem is to be solved at the Haskell level in a generic way, e.g., by making it possible to save evaluated global values just before a program quits (using pragmas), and reading them back when the program is run for the next time; in this way the evaluation of CAFs is memoized over different runs of the program.

One might object that library code used in this paper goes far beyond the normal use of the Haskell type system, and that our type-level programming is not for the everyday Haskell programmer. We agree completely, although some of the complexity is already hidden in the libraries. Moreover, we believe type-level programming is a promising research area, which has broad interest in the (functional) programming languages community. Another possible line of future work is to explore the implementation of our techniques in a dependently-typed language, such as Agda or Coq.

References

[1] H. Abramson. Definite clause translation grammars. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1984.

[2] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, Sept. 1998. ISSN 0362-1340.

[3] S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., 1991.

[4] A. I. Baars, S. D. Swierstra, and M. Viera. Typed Transformations of Typed Abstract Syntax. In *TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

[5] A. I. Baars, S. D. Swierstra, and M. Viera. Typed Transformations of Typed Grammars: The Left Corner Transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

[6] M. Bravenboer. *Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2008.

[7] O. de Moor, K. Backhouse, and S. D. Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. ISSN ISSN 0350-5596. Special Issue: Attribute grammars and Their Applications.

[8] O. de Moor, L. Peyton Jones, Simon, and V. Wyk, Eric. Aspect-oriented compilers. In *Proceedings of the 1st Int. Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag. ISBN 3-540-41172-0.

[9] A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6.

[10] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007. ISSN 0167-6423.

[11] D. Fisher and O. Shivers. Static analysis for syntax objects. In *Proceedings of the eleventh International Conference on Functional Programming*, pages 111–121, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3.

[12] D. Fisher and O. Shivers. Building language towers with ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, Sept. 2008. ISSN 0956-7968.

[13] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

[14] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000. ISSN 0167-6423.

[15] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 154–173, London, UK, 1987. Springer-Verlag. ISBN 3-540-18317-5.

[16] U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13: 229–256, 1980.

[17] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 Workshop on Haskell*, pages 96–107. ACM Press, 2004.

[18] M. F. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. RUU-CS 86-16, Department of Computer Science, 1986.

[19] B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966. ISSN 0001-0782.

[20] B. Martínez, M. Viera, and A. Pardo. Just Do It While Compiling!: Fast Extensible Records in Haskell. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM'13)*, 2013.

[21] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007. .

[22] M. Mernik and V. Žumer. Incremental programming language development. *Computer languages, Systems and Structures*, 31:1–16, 2005.

[23] A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Visit Functions for the Semantics of Programming Languages. In *Workshop on Generative Programming*, 2010.

[24] R. Paterson. A new notation for arrows. In *Proceedings of the 6th Int. Conference on Functional Programming*, pages 229–240, New York, USA, 2001. ACM. ISBN 1-58113-415-0.

[25] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9): 50–61, 2006. ISSN 0362-1340.

[26] A. M. Sloane, L. C. L. Kats, and E. Visser. A pure object-oriented embedding of attribute grammars. In *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications*, March 2009.

[27] S. D. Swierstra. Parser combinators: from toys to tools. In G. Hutton, editor, *Haskell Workshop*, 2000.

[28] S. D. 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*, pages 252–300. Spinger, 2009.

[29] S. D. Swierstra, P. R. Azero Alcocer, and J. A. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[30] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75 (1–2):39–54, January 2010.

[31] M. Viera and S. Swierstra. Compositional compiler construction: Oberon0. Technical Report UU-CS-2012-016, Department of Information and Computing Sciences, Utrecht University, 2012.

[32] M. Viera and S. D. Swierstra. Attribute grammar macros. In *XVI Simpósio Brasileiro de Linguagens de Programação*, LNCS, pages 150–165, 2012.

[33] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell. In *Proceedings of the 14th Int. Conf. on Functional Programming*, pages 245–256, New York, USA, 2009. ACM. ISBN 978-1-60558-332-7.

[34] M. Viera, S. D. Swierstra, and A. Dijkstra. Grammar Fragments Fly First-Class. In *Proceedings of the 12th Workshop on Language Descriptions Tools and Applications*, pages 47–60, 2012.

[35] M. Viera, S. D. Swierstra, and A. Middelkoop. UUAG Meets AspectAG. In *Proceedings of the 12th Workshop on Language Descriptions Tools and Applications*, 2012.

[36] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the 1993 conference on Programming Language Design and Implementation*, pages 156–165, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4.

[37] N. Wirth. *Compiler construction*. International computer science series. Addison-Wesley, 1996. ISBN 978-0-201-40353-4.