

First Class Syntax, Semantics, and Their Composition

How to assemble a compiler on the fly

Marcos Viera

Instituto de Computación
Universidad de la República
Montevideo, Uruguay
mviera@ing.edu.uy

S. Doaitse Swierstra

Atze Dijkstra
Department of Computer Science
Utrecht University
Utrecht, The Netherlands
{doaitse,atze}@cs.uu.nl

Arthur Baars

ProS: Centro de Investigación en
Métodos de Producción de Software
Universidad Politécnica de Valencia
Valencia, Spain
abaars@pros.upv.es

Abstract

Having extensible languages is appealing and thus raises the question how to construct the associated extensible compilers. In recent years we have developed a collection of techniques which together now make this possible in Haskell, doing so *in a type safe way*: transformation of typed abstract syntax trees makes it possible to construct parsers on the fly, parser combinators make it possible to construct parsers dynamically, and first-class attribute grammars make it possible to define the associated semantics compositionally.

In this paper we show how, using a small example language and a single extension, all our techniques fit together in constructing compilers out of a collection of pre-compiled, statically type-checked “language-definition fragments”.¹

Categories and Subject Descriptors D.3.3 [Programming languages]: Language Constructs and Features; D.1.1 [Programming techniques]: Applicative (Functional) Programming

General Terms Design, Languages, Performance, Standardization

Keywords Attribute Grammars, Typed Transformations, Typed Grammars, Haskell, Arrows, GADTs, Compositional language Definitions, Syntax macros

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 else 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 how to compose a compiler, various lines of attack have been pursued. The most direct and least invasive one, which is so widely applied that one may not recognise 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 become 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 we characterize as the construction of *embedded domain specific languages* (EDSL). The possibility to define operators and precedences can be used to mimic syntactic extensions. By making (amazing and unforeseen) use of the Haskell class system we may even embed some form of parsing in the host language (Hinze and Paterson 2003; McBride and Paterson 2008; Swierstra 2008). Unfortunately not all programming languages really support this approach very well, given the flood of so-called modelling 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 (Dijkstra 2005; Dijkstra et al. 2008, 2009) 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 lose 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 how we can do better than only providing powerful abstraction mechanisms, but without opening up the whole source of the compiler. The most commonly found approach is to introduce so-called *syntax macros* (Leavenworth 1966), which enable the programmer 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 represen-

¹ The code of this paper is available as a package from <http://hackage.haskell.org/package/SyntaxMacros>

tation, 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. An infamous example of this problem is the Hugs complaint about “an unexpected semi-colon”, where the original source program does not contain a single semi-colon; the hidden semi-colon was inserted by a pre-processor in order to deal with the offside rule. Combinator languages are another place where the problem of reporting errors back in terms of the underlying language pops up. Type errors are often given in terms of the underlying, possibly quite complicated, types which thus leak out from the library implementing the EDSL; the problem can be partially remedied by providing proper type definitions shielding these implementations, but this requires extra work and partially defeats the advantages of having type inference. There are more fundamental approaches to deal with such leakage problems, but this requires the library implementer to script the type inferencer in such a way that error messages are somehow translated back into the concepts as understood by the EDSL-programmer. To implement such an approach, extensive support from the underlying compiler is required and should be taken into account when constructing the compiler (Heeren et al. 2003; Heeren 2005).

Given the above considerations we impose some quite heavy restrictions on ourselves. In the first place such extensions should go beyond merely syntactic extensions as is the case with the original syntax-macros (Leavenworth 1966), which only map new syntax onto existing syntax; we want also to access existing static semantics part of the compiler, e.g., in order to influence the way errors are reported such that *the way the compiler is eventually composed is not reflected in its behaviour*. Furthermore we are explicitly not satisfied with just being able to extend the compiler text in an easy and compositional way (which in our terminology would be to change a compiler); we can already do so using our attribute grammar system. Instead we are seeking 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 component(s), to which extra components can be added at will. The core question answered in this paper is *how to do this in a statically typed language* like Haskell.

The solution we present builds on several related developments:

- the introduction of typed abstract syntax (Baars and Swierstra 2002)
- 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 (Baars et al. 2009a)
- the description of typed grammar descriptions and the typed Left-Corner Transform to remove left-recursion (Baars et al. 2009b)
- the possibility to construct self-analysing, error correcting parser on the fly (Swierstra 2000, 2008)
- the possibility to deal with attribute grammars as first class Haskell values, which can be transformed, composed and finally evaluated. (Viera et al. 2009).

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, and in Section 3 show the corresponding static semantics parts. In Section 4 we show how we have used the techniques from the papers mentioned above. We close by discussing related work, future work and present our conclusions.

All the code to be written by the language definers is given in figures. Figure 1 shows the definition of the grammar of the original language and Figures 4, 5, 6 and 7 its semantics. The extension of

the language is shown in Figures 2, for the grammar, and 8, 9, 10 and 11, for its semantics.

2. Context-Free Grammar

In this section we show how to express the context free grammar parts. We start with a simple expression language, defined by the *initial grammar* :

```
exp ::= "let" var "=" exp "in" exp
      | term "+" exp | term
term ::= term "*" factor | factor
factor ::= int | var
```

Note that this concrete grammar uses the syntactic categories *term*, *exp* and *factor* to describe the operator precedences. All these non-terminals will later be associated with the same abstract non-terminal.

Figure 1 shows what a language implementer has to provide, using the provided combinator library, to describe how to parse this language fragment; we immediately recognise the context-free grammar just given in the structure of the code. Each non-terminal of the CFG is introduced (with *addNT*) by defining a list of productions, where each production is in an *applicative style*. For example, the production *term "*" factor* of the non-terminal *term* is defined as:

```
prd semMul $ ch_me1 ∈ term <> trm "*" <>
             ch_me2 ∈ factor <> prdEnd
```

An element of a production (the part after the \$) is a sequence of elements separated by <> symbols; an element is either a terminal (*trm "*"*), representing a literal to be recognized, or a non-terminal which is represented by a pair of values separated by an ∈ token. The left argument of ∈ is a label (introduced elsewhere, and all starting with *ch_*) for this position and the right hand side argument tells us which non-terminal occurs here in the production. The functions starting with *sem* (e.g. *semMul*) describe how to combine the semantic values of the non-terminals in the production to the semantic value of the left hand side. We call them *semantic functions*, since they give semantics to the language. They can refer to the children by using the labels of the children. In section 3 we show how to construct and adapt the semantic functions using a first-class attribute grammar library.

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 recognised 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 (with the form *nt...*) and the collection of right hand sides which can be used and modified in future extensions.

The function *closeGram* takes the list of productions, and converts it into a compiler; in our case that is parser plus semantics for the language starting from the first non-terminal *root*.

2.1 Language Extension

Next we extend the language with some extra productions; one for defining the square of a value, one for defining the sum of the squares of two values, and one for defining a substitution. Furthermore we add the possibility to use parentheses to influence the way expressions are parsed:

```
exp ::= ...
      | "square" exp
      | "pyth" factor factor
```

```

prds = proc () → do
  rec root ← addNT < [prd ('semAGItf'()) $ ch_expr ∈ exp <> prdEnd]
  exp ← addNT < [prd semLet $ trm "let" <> ch_lnm ∈ var <> trm "=" <> ch_val ∈ exp <>
                trm "in" <> ch_body ∈ exp <> prdEnd
                , prd semAdd $ ch_ae1 ∈ term <> trm "+" <> ch_ae2 ∈ exp <> prdEnd
                , prdId term ]
  term ← addNT < [prd semMul $ ch_me1 ∈ term <> trm "*" <> ch_me2 ∈ factor <> prdEnd
                , prdId factor]
  factor ← addNT < [prd semCst $ ch_cv ∈ int <> prdEnd
                , prd semVar $ ch_vnm ∈ var <> prdEnd]
  exportNTs < Export root $ ntExp ∈ exp ^|
            ntTerm ∈ term ^|
            ntFactor ∈ factor ^|
            ntNil

gram = closeGram prds

```

Figure 1. Initial language

```

prds' = proc (Export root nts) → do
  let exp = getNT ntExp nts
  addProds < (exp, [prd semSq $ trm "square" <> ch_se ∈ exp <> prdEnd
                  , prd semPyth $ trm "pyth" <> ch_pe1 ∈ factor <> ch_pe2 ∈ factor <> prdEnd
                  , prd semSubst $ ch_body ∈ exp <>
                    trm "[" <> ch_lnm ∈ var <> trm "|" <> ch_val ∈ exp <> trm "]" <> prdEnd])
  let factor = getNT ntFactor nts
  addProds < (factor, [prd semPar $ trm "(" <> ch_pe ∈ exp <> trm ")" <> prdEnd])
  exportNTs < (Export root nts)
  gram' = closeGram $ extendGram prds prds'

```

Figure 2. Language Extension

```

prds' :: (NTRecord (nts env) , GetNT NTExp (nts env) (Symbol AttExpr TNonT env)
        , GetNT NTFactor (nts env) (Symbol AttExpr TNonT env)
        ) ⇒ SyntaxMacro env (Export start nts) (Export start nts)

```

Figure 3. The type of $prds'$

```

| exp "[" var "|" exp "]"
factor ::= ...
| "(" exp ")"

```

This language extension $prds'$ is defined as a Haskell value by itself in Figure 2. For each non-terminal to be extended we retrieve its list of productions (using `getNT`) from the ‘state’ nts , and add new productions to this list using `addProds`. For example, for `factor` the new production for parentheses is added by:

```

let factor = getNT ntFactor nts
addProds < (factor, [prd semPar $
  trm "(" <> ch_pe ∈ exp <> trm ")" <> prdEnd])

```

This code shows how to combine the previously defined productions with the newly defined productions into a complete grammar. New non-terminals can be added as well using `addNT`, although this is not demonstrated in the example.

Because both $prds$ and $prds'$ are both proper Haskell values which are separately defined in different modules we claim that the term *first class syntax macros* is justified here.

Being able to define these language fragments separately raises the question how the type system is used to check the well-formedness of their composition? The crucial observation is that all assumptions about the other parts of the grammar are encoded using class constraints (see Figure 3); the type of the exported non-terminals ($nts\ env$) is restricted to be a list (here an indexed nested product) of lists of productions by $NTRecord\ (nts\ env)$. The second and third constraint express that this ($nts\ env$) should contain entries for the non-terminals labeled with $NTExp$ and $NTFactor$. The fact that the two ($Export\ start\ nts$) arguments of the resulting $SyntaxMacro\ env\ (Export\ start\ nts)\ (Export\ start\ nts)$ are the same, indicates that the root symbol does not change and that the list of non-terminals does not grow with this language extension. Parameters like $Symbol\ AttExpr\ TNonT\ env$ indicate that the semantic type associated with both non-terminals is described by the type $AttExpr$, i.e. abstract syntax trees belonging to these

non-terminals will be attributed like expressions. It is here that it becomes apparent that for the static semantics, i.e. after parsing, these non-terminals are treated the same.

3. Semantic Functions

In this section we complete the example by showing how the static semantics are defined for the initial language and how they can be redefined when the language is extended.

3.1 AspectAG

To define the static semantics of a language we use the AspectAG² (Viera et al. 2009) embedding of attribute grammars in Haskell and start out by defining the type of abstract syntax trees which are to be attributed, as shown in Figure 4.

```

data AGItf = AGItf { expr :: T_Expr }
data T_Expr
  = Cst { cv :: Int }
  | Var { vnm :: String }
  | Mul { me1 :: T_Expr, me2 :: T_Expr }
  | Add { ae1 :: T_Expr, ae2 :: T_Expr }
  | Let { lnm :: String, val :: T_Expr, body :: T_Expr }
  $ (deriveAG "AGItf")

```

Figure 4. Language Semantics: Abstract Syntax Tree

The Template Haskell function *deriveAG* generates the labels (i.e. names holding values of unique types) we used in Picture 1 to refer to the children of the productions: *ch_expr*, *ch_cv*, *ch_vnm*, *ch_me1*, *ch_me2*, *ch_ae1*, *ch_ae2*, *ch_lnm*, *ch_val* and *ch_body*.

Next we define for each attribute and for each position in the abstract syntax where this attribute has a defining position how it is to be computed. Figures 5 and 6 show the definitions for these attribute occurrences. We define attributes for the following aspects: pretty printing, realized by *spp*, which holds a pretty printed document of type *PP_Doc* (Figure 5), and expression evaluation, realized by *sval* of type *Int*, which holds the result of an expression, and *ienv* which holds the environment ($[(String, Int)]$) in which an expression is to be evaluated (Figure 6). The attributes *spp* and *sval* are *synthesized attributes*; they take their definition “from below”, using the values of the synthesized attributes of the children of the node and the inherited attributes of the node itself. The attribute *ienv* is an *inherited attribute*, which takes its definition “from above”, referring to the inherited attributes of its father and the synthesized attributes of its siblings.

The extensible records of *HList*³ (Kiselyov et al. 2004) are used by AspectAG to hold collections of attributes. Thus, the type (semantic domain) associated with *T_Expr* is a function from the collection of inherited attributes (in our case only *ienv*) to the collection of synthesized attributes (*spp* and *sval*):

```

type AttExpr
  = Record
    (HCons (LVPair (Proxy Att_ienv) [(String, Int)])
      HNil)
  → Record
    (HCons (LVPair (Proxy Att_spp) PP_Doc)
      (HCons (LVPair (Proxy Att_sval) Int)
        HNil))

```

²<http://hackage.haskell.org/package/AspectAG>

³<http://hackage.haskell.org/package/HList>

The function *syndefM* introduces a new synthesized attribute to be accessed by the given label. The second argument is the monad which constructs the value of the attribute; from within this monad one may refer to the other attributes defined with this same production. So the *sppAdd* from Figure 5 defines the attribute *spp* for the production *Add*:

```

sppAdd = syndefM spp $
  do e1 ← at ch_ae1
      e2 ← at ch_ae2
      return $ (e1 # spp) >|< " + " >|< (e2 # spp)

```

It combines the pretty printed child *ae1* with the string " + " and the pretty printed child *ae2*, using the pretty printing combinator ($>|<$), for horizontal (beside) composition, from the *uulib*⁴ library. The types used as label for *spp* and other attributes are generated by the Template Haskell function *attLabels*.

The function *inhdefM* introduces a new inherited attribute. In our example the *ienvLet* defines the computation of the attribute *ienv* (environment) for the production *Let*:

```

ienvLet = inhdefM ienv exprNT $
  do lnm ← at ch_lnm
      val ← at ch_val
      lhs ← at lhs
      return $ ch_val .=. lhs # ienv .*.
              ch_body .=. (lnm, val # sval) :
              lhs # ienv .*.
  emptyRecord

```

An attribute is being defined for each child of which semantic category is in the list *exprNT*, storing these definitions in an extensible record (given the combinators $(.=)$ for field definition and $(.*)$ for record extension) labeled by the names of the children. The *ienv* value coming from the parent (*lhs* stands for “left-hand side”) is copied to the *ienv* position of the child *ch_val*. On the other hand, in the case of *ch_body*, the environment is extended with a pair composed by the value of the child *ch_lnm* and the value of the attribute *sval* of the child *ch_val*.

Explicitly giving all rules soon becomes cumbersome, so some shortcuts are available: **copy** rules and **use** rules. A *copy* rule copies an inherited attribute from a parent to all its children. The function *copy* takes the name of the attribute and an heterogeneous list of semantic categories for which the attribute has to be defined, and generates *copy* rules for these. For example, we can define a *copy* rule for *ienv*:

```

ienvRule = copy ienv exprNT

```

so we can declare:

```

ienvAdd = ienvRule

```

instead of having to write:

```

ienvAdd = inhdefM ienv exprNT $
  do lhs ← at lhs
      return $ ch_ae1 .=. lhs # ienv .*.
              ch_ae2 .=. lhs # ienv .*.
  emptyRecord

```

A *use* rule introduces a synthesized attribute that collects information from some of the children. The function *use* takes the following arguments: the attribute being defined, the list of semantic categories for which the attribute is to be defined, a monoidal operator which combines the attribute values, and a unit value to be used in those cases where none of the children has such an attribute. The

⁴<http://hackage.haskell.org/package/uulib>

```

$(attLabels ["spp"])
sppAGItf = syndefM spp $ liftM (#spp) (at ch_expr)
sppCst   = syndefM spp $ liftM pp (at ch_cv)
sppVar   = syndefM spp $ liftM pp (at ch_vnm)
sppMul   = syndefM spp $ do e1 ← at ch_me1
                          e2 ← at ch_me2
                          return $ e1 # spp >|< " * " >|< e2 # spp
sppAdd   = syndefM spp $ do e1 ← at ch_ae1
                          e2 ← at ch_ae2
                          return $ e1 # spp >|< " + " >|< e2 # spp
sppLet   = syndefM spp $ do lnm ← at ch_lnm
                          val ← at ch_val
                          body ← at ch_body
                          return $ "let " >|< pp lnm >|< " = " >|< val # spp >|< " in " >|< body # spp

```

Figure 5. Language Semantics: Pretty Printing

```

$(attLabels ["ienv", "sval"])
exprNT = nt_T_Expr .* hNil
allNT  = nt_AGItf .* exprNT
ienvRule = copy ienv exprNT
ienvAGItf = inhdefM ienv exprNT $ do return (ch_expr .=. ([] :: [(String, Int)]) .* emptyRecord)
ienvCst   = ienvRule
ienvVar   = ienvRule
ienvMul   = ienvRule
ienvAdd   = ienvRule
ienvLet   = inhdefM ienv exprNT $ do lnm ← at ch_lnm
                          val ← at ch_val
                          lhs ← at lhs
                          return $ ch_val .=. lhs # ienv .*
                          ch_body .=. (lnm, val # sval) : lhs # ienv .*
                          emptyRecord

svalRule f = use sval allNT f (0 :: Int)
svalAGItf = svalRule ((* :: Int → Int → Int)
svalCst   = syndefM sval $ liftM id (at ch_cv)
svalVar   = syndefM sval $ do vnm ← at ch_vnm
                          lhs ← at lhs
                          return $ fromJust (lookup vnm (lhs # ienv))
svalMul   = svalRule ((* :: Int → Int → Int)
svalAdd   = svalRule ((+ :: Int → Int → Int)
svalLet   = syndefM sval $ liftM (#sval) (at ch_body)

```

Figure 6. Language Semantics: Evaluation

attribute *sval* can now be endowed with such a collecting property by:

```
svalRule f = use sval allNT f (0 :: Int)
```

and subsequently we use this property in:

```
svalAdd = svalRule ((+) :: Int → Int → Int)
```

instead of:

```

svalAdd = syndefM sval $
  do e1 ← at ch_ae1
     e2 ← at ch_ae2
     return $ (e1 # sval) + (e2 # sval)

```

In Figure 7 we show how the semantic functions of the example are defined. For each production, its attributes have to be combined using the function *ext*. Finally, the function *knit* is applied to the combined attributes for the production.

The function *knit* takes the combined attribute computations and the semantic functions of the children in a record, labeled by the name of each child preceded by “*ch_*”, and computes the semantic function. Thus, for example, a correct call to *semAdd* is:

```

semAdd (ch_ae1 .=. e1 .*
       ch_ae2 .=. e2 .*
       emptyRecord)

```

```

attsAGItf = sppAGItf 'ext' ienvAGItf 'ext' svalAGItf
attsCst   = sppCst   'ext' ienvCst   'ext' svalCst
attsVar   = sppVar   'ext' ienvVar   'ext' svalVar
attsMul   = sppMul   'ext' ienvMul   'ext' svalMul
attsAdd   = sppAdd   'ext' ienvAdd   'ext' svalAdd
attsLet   = sppLet   'ext' ienvLet   'ext' svalLet

semAGItf = knit attsAGItf
semCst   = knit attsCst
semVar   = knit attsVar
semMul   = knit attsMul
semAdd   = knit attsAdd
semLet   = knit attsLet

```

Figure 7. Language Semantics: Semantic Functions

where $e1$ and $e2$ are the semantic functions of $ae1$ and $ae2$, respectively. In subsection 4.1 we will explain how this call is made out of the code of Figure 1. Note that the labels ch_ae1 and ch_ae2 used here are also used in Figure 1 when introducing the production for the addition.

3.2 Attribute Redefinitions

Our definition of substitution, in Figure 2, is just an alternative way to write a **let** expression. Thus, we could naively define $semSubst$ as:

```
semSubst = semLet
```

But this has the disadvantage sketched before; if we pretty print the expression:

```
x [x | 10]
```

we get:

```
let x = 10 in x
```

So, we have to *redefine* the pretty printing attribute as shown in Figure 8.

```

sppSubst = synmodM spp $
  do lnm ← at ch_lnm
     val ← at ch_val
     body ← at ch_body
     return $ (body # spp) >|<
              [" >|< (pp lnm) >|< " | " >|<
               (val # spp) >|< "]"
semSubst = knit (sppSubst 'ext' attsLet)

```

Figure 8. Semantics Extension: Substitution

The function $synmodM$ modifies the definition of an existing synthesized attribute, taking the name of the attribute and the new computation. The AspectAG library also provides a function $inhmodM$ ⁵ to modify the definition of an inherited attribute for a collection of semantic categories. Note that the arguments of $synmodM$ and $inhmodM$ play the same rôle as those of $syndefM$ and $inhdefM$. To apply a modification we have to redefine the already defined attributes using ext .

Notice that the programmer of the extensions does not need to know the details of the implementation of every attribute. In order to implement a redefinition for a production only the names of the

⁵The functions $synmodM$ and $inhmodM$ were introduced for this work.

children of the production are needed, which are provided in the definition of the abstract syntax tree (Figure 4 in the example).

In the rest of this section we show how the semantic functions of the other extensions of the example are implemented.

The square of a value is the multiplication of this value by itself. Thus, the semantics of multiplication can be used as a basis, by passing to it the semantics of the only child (ch_se) of the square production both as ch_me1 and ch_me2 . Again, we have to redefine the pretty printing attribute.

```

$ (chLabel "sqe" "T_Expr)
sppSq = synmodM spp $
  do me1 ← at ch_me1
     return $ "square " >|< (me1 # spp)
semSq r = knit (sppSq 'ext' aspMul)
          (ch_me1 .=. (r # ch_sqe) .*.
           ch_me2 .=. (r # ch_sqe) .*.
           emptyRecord)

```

Figure 9. Semantics Extension: Square

When defining the extension for substitution, in Figure 2, we used the same labels of the children of the **let** expression (ch_lnm , ch_val and ch_body). In the case of square we introduce a new label ch_sqe , which is associated to ch_me1 and ch_me2 in the definition of $semSq$.

For the sum of squares, we use the semantic of the addition as a basis, redefining the pretty printing (spp) and evaluation ($sval$) attributes.

```

$ (chLabels ["pe1", "pe2"] 'T_Expr)
sppPyth = synmodM spp $
  do ae1 ← at ch_ae1
     return $ "pyth " >|< (ae1 # spp)
svalPyth = synmodM sval $
  do ae1 ← at ch_ae1
     ae2 ← at ch_ae2
     return $ (λp1 p2 → p1 * p1 + p2 * p2)
              (ae1 # sval) (ae2 # sval)
semPyth r = knit (sppPyth 'ext' svalPyth 'ext' aspAdd)
          (ch_ae1 .=. (r # ch_pe1) .*.
           ch_ae2 .=. (r # ch_pe2) .*.
           emptyRecord)

```

Figure 10. Semantics Extension: Sum of Squares

In the case of parenthesis, we do not use any other production's semantics definition as a basis. We just define them from scratch:

```

$ (chLabel "pe" "T_Expr)
sppPar = syndefM spp $
  do pe ← at ch_pe
     return $ "(" >|< pe # spp >|< ")"
ienvPar = copy ienv exprNT
svalPar = syndefM sval $ liftM (#sval) (at ch_pe)
semPar = knit (sppPar 'ext' ienvPar 'ext' svalPar)

```

Figure 11. Semantics Extension: Parenthesis

4. Syntax Macros Library

In this section the syntax macros library is presented. The library is based on typed representation of grammars and typed transformations (Baars et al. 2009a) of these grammars.

4.1 Grammar Representation

We use the representation of grammars as typed abstract syntax proposed in (Baars et al. 2009b), based on the use of Generalised Algebraic Data Types (Peyton Jones et al. 2006). The idea is to indirectly refer to non-terminals via references encoded as types. Such references type-index into an environment holding the actual trees for non-terminals.

A *Ref* encodes a typed reference to an environment containing values of different types. It is labeled with the type *a* of the referenced value and the type *env* of the environment (a nested Cartesian product extending to the right) which contains the value.

```
data Ref a env where
  Zero :: Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)
```

The constructor *Zero* expresses that the first element of the environment has to be of type *a*. The constructor *Suc* remembers a position in the rest of the environment. It ignores the first element in the environment by being polymorphic in the type *b*.

This encoding was introduced by Pasalic and Linger (Pasalic and Linger 2004). This idea is extended in (Baars et al. 2009b) such that environments *Env* consist of a collection of possibly mutually recursive definitions. Instead of containing values of different types, an environment contains terms describing those values. These terms can also contain typed references to other terms. Thus, the type of a term is *t a use*, where the type parameter *a* is the type of the described value and *use* the environment to which references to other terms occurring in the term may point.

```
data Env t use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use
        → Env t use (def', a)
```

The type parameter *def* contains the type labels *a* of the terms of type *t a use* occurring in the environment. When a term is added to the environment using *Ext*, its type label is included as the first component of *def*. The type *use* describes the types that may be referred to from within terms of type *t a use* using *Ref a use* values. The type *FinalEnv* forces environments *def* and *use* to coincide, thereby closing an environment. References in the terms in a *FinalEnv* do not point to values outside this *FinalEnv*.

```
type FinalEnv t usedef = Env t usedef usedef
```

A grammar consists of a closed environment, containing a list of alternative productions for each non-terminal, and a reference (*Ref a env*) to one of these non-terminals which is the start symbol. The type *a* is the type of the witness of a complete successful parse. The type *env* is hidden using existential quantification, so changes to the structure of the grammar can be made, by adding or removing non-terminals, without having to change its type.

```
data Grammar a
  = ∀ env. Grammar (Ref a env)
    (FinalEnv Productions env)
newtype Productions a env
  = PS {unPS :: [Prod a env]}
```

A production is a sequence of symbols terminated with an *End f* element, with *f* representing the semantics, usually a function taking the results of the earlier elements as arguments.

```
data Prod a env where
  Seq :: Symbol b t env → Prod (b → a) env
      → Prod a env
  End :: a → Prod a env
```

A symbol is either a terminal or a non-terminal. A non-terminal is encoded by a reference pointing to one of the elements of the environment *env*. A normal terminal contains the literal string it represents and has *DTerm* as its witness. We define a category of *attributed terminals*, which are not fixed by a literal string. Every attributed terminal refers to a lexical structure. Although in the case of terminals the parsed value is ignored when evaluating semantics, in attributed terminals the parsed values are used, so the type *a* instantiates to the type of the parsed value.

```
data DTerm = DTerm
data TTerm
data TNonT
data TAttT
data Symbol a t env where
  Term  :: String → Symbol DTerm TTerm env
  Nont  :: Ref a env → Symbol a TNonT env
  TermInt  :: Symbol Int TAttT env
  TermChar :: Symbol Char TAttT env
  TermVarid :: Symbol String TAttT env
  TermConid :: Symbol String TAttT env
  TermOp   :: Symbol String TAttT env
```

We extended the type *Symbol* proposed in (Baars et al. 2009b) with a type parameter *t* that indicates, at type-level, if a *Symbol* is a terminal (type *TTerm*), non-terminal (*TNonT*) or attributed terminal (*TAttT*).

As we have seen in section 3, the semantic functions defined with AspectAG take as argument an heterogeneous record with the semantics of every child of the production. Thus, for example, the production for the addition operation can be encoded with the value:

```
let f = λe2 DTerm e1 → semAdd (ch_ae1 .=. e1 .*
                               ch_ae2 .=. e2 .*
                               emptyRecord)
in Seq term $ Seq (Term "+") $ Seq exp $ End f
```

This notation is somewhat cumbersome to use directly, so we have introduced additional right-associative combinators to make the definitions for the productions look prettier. The combinators construct the sequence of symbols (terminals and non-terminals), while collect and group the arguments for the semantic function. So, with the additional combinators the above production for the addition looks like:

```
prd semAdd $ ch_ae1 ∈ term <> trm "+" <>
             ch_ae2 ∈ exp <> prdEnd
```

Let us see how the combinators are defined. We start by defining the following smart constructors for the terminals:

```
trm = Term
int  = TermInt
char = TermChar
var  = TermVarid
con  = TermConid
op   = TermOp
```

So we can write for example *trm "+"* instead of *Term "+"*.

The type *LSPair* is defined to associate a label *nt* with a symbol (*Symbol a t env*). It is similar to the field definition

LVPair in *HList*, but here we restrict the value to be a *Symbol* and we also lift the environment *env* to the type.

```
newtype LSPair nt a t env
  = LSPair { symLSPair :: (Symbol a t env) }
infixr 6 ∈
(∈) _ = LSPair
```

The type *nt* is used to describe a label as a type-level value. As we have said before, labels are generated by Template Haskell functions (*deriveAG* in Figure 4). An example of a label is:

```
data Ch_ae1
ch_ae1 = ⊥ :: Proxy (Ch_ae1, T_Expr)
```

The type *Ch_ae1* has no inhabitants at the value-level, since it will be only used to perform computations at the type-level. For example, having *term* :: *Symbol Att_Expr TNonT env*, the code:

```
ch_ae1 ∈ term
```

has type:

```
LSPair (Proxy (Ch_ae1, T_Expr)) Att_Expr TNonT env
```

and the value \perp of *ch_ae1* is never evaluated.

The type *PreProd* is the type of the “pre-productions”, i.e. a function that constructs a production *Prod a env* given a function $r \rightarrow a$.

```
data PreProd r a env
  = PreProd ((r → a) → (Prod a env))
```

Thus, given a semantic function *sem* with type $(r \rightarrow a)$, where *r* is a record with the semantic functions of the children and *a* the semantic domain, and a pre-production *PreProd r a env*, a production is obtained just by applying the pre-production to *sem*.

```
prd :: (r → a) → PreProd r a env → Prod a env
prd sem (PreProd pp) = pp sem
```

A basic pre-production is one that generates an empty production, that is, with no symbols:

```
prdEnd :: PreProd (Record HNil) a env
prdEnd = PreProd $ λf → End (f emptyRecord)
```

Since this production has no children, its semantic function *f* takes an empty record.

We define a type-class *ProdSeq* for the right-associative combinators (*<>*) to construct a pre-production.

```
class ProdSeq s r a r' a' | s r' a' → r a where
  (<>) :: s env → PreProd r a env → PreProd r' a' env
```

The combinator (*<>*) constructs a sequence from a symbol *s* and a pre-production *PreProd r a env* representing the rest of the sequence.

If the symbol is a non-terminal (associated to a label *nt*), the sequence is constructed with the symbol as the first element and the rest of the sequence resulting from applying the pre-production *pp* to a function *f*, that takes a record *r* and extends it with the value of the argument *x* before passing it to *fl*. Here we can see how the semantic function *fl* that “comes from the left” receives the record *r* constructed “from the right”.

```
instance (HExtend (LVPair nt v) r r')
  ⇒ ProdSeq (LSPair nt v TNonT)
    r (v → a)
    r' a where
  s <> pp = PreProd $
```

```
λfl → let f r = λx → fl (labelLSPair s .=. x .* r)
in Seq (symLSPair s) $ prd f pp
```

When the symbol corresponds to a terminal, *f* takes a parameter of type *DTerm*, but does not add it to the record.

```
instance ProdSeq (Symbol DTerm TTerm)
  r (DTerm → a)
  r a where
```

```
s <> pp = PreProd $
λfl → let f r = λDTerm → fl r
in Seq s $ prd f pp
```

There is also a combinator for the case when the symbol is an attributed terminal. In this case the semantic function of attributed terminals ($\lambda(\text{Record HNil}) \rightarrow x$) is added to the record. This function takes an empty input and returns the value of the attribute.

```
instance (HExtend (LVPair nt (Record HNil → v)) r r')
  ⇒ ProdSeq (LSPair nt v TAttT)
    r (v → a)
    r' a where
```

```
s <> pp = PreProd $
λfl → let f r = λx → fl (labelLSPair s .=.
  (λ(Record HNil) → x)
  .* r)
in Seq (symLSPair s) $ prd f pp
```

Finally, a function is included to create productions that only include a non-terminal and do not change its semantics.

```
prdId :: Symbol a TNonT env → Prod a env
prdId nt = Seq nt $ End id
```

4.2 TTTAS

Grammar definitions and extensions are defined as typed transformations of values of type *Grammar*, implemented using the library TTTAS⁶ (Typed Transformations of Typed Abstract Syntax). TTTAS is based on the type *Trafo*, which represents typed transformation steps, (possibly) extending an environment *Env*.

```
data Trafo m t s a b
```

The arguments are the types of: the meta-data *m*, the terms *t* stored in the environment, the final environment *s*, the input *a* and output *b*. The type *Trafo* is an *Arrow* (Hughes 2000). Thus, instances of the classes *Category* and *Arrow* are implemented for (*Trafo m t s*). That provides a set of functions for constructing and combining *Trafos*. Some of these functions are:

- Identity arrow (like *return* in monads)

```
returnA :: Arrow a ⇒ a b b
```

- Lifting a function to an arrow

```
arr :: Arrow a ⇒ (b → c) → a b c
```

- Left-to-right composition

```
(>>>) :: Category cat ⇒ cat a b → cat b c → cat a c
```

The class *ArrowLoop* is instantiated to provide feedback loops with its member:

```
loop :: a (b, d) (c, d) → a b c
```

There also exists a convenient notation (Paterson 2001) for *Arrows*, which is inspired by the **do**-notation for *Monads*.

⁶<http://hackage.haskell.org/package/TTTAS>

A transformation is run with *runTrafo*, starting with an empty environment and an initial value of type *a*. The universal quantification over the type *s* ensures that transformation steps cannot make any assumptions about the type of the (yet unknown) final environment.

```
runTrafo :: (∀ s. Trafo m t s a (b s)) → m () → a
          → Result m t b
```

The result of running a transformation is encoded by the type *Result*, containing the meta-data, the output type and the final environment. It is existential in the final environment, because in general we do not know how many definitions are introduced by a transformation and which are their types. Note that the final environment has to be closed.

```
data Result m t b
  = ∀ s. Result (m s) (b s) (FinalEnv t s)
```

New terms can be added to the environment by using the function *newSRef*. It takes the term to be added as input and yields as output a reference that points to this term in the final environment.

```
newSRef :: Trafo Unit t s (t a s) (Ref a s)
data Unit s = Unit
```

The type *Unit* is used to represent that this transformation does not record any meta-information.

Functions (*FinalEnv t s* → *FinalEnv t s*) for updating the final environment of a transformation can be lifted into the *Trafo* and composed using *updateFinalEnv*. All functions lifted using *updateFinalEnv* will be applied to the final environment once it is created.

```
updateFinalEnv :: Trafo m t s
                (FinalEnv t s → FinalEnv t s) ()
```

If we have, for example:

```
proc () → do
  ...
  updateFinalEnv < upd1
  ...
  updateFinalEnv < upd2
  ...
```

the function (upd2 . upd1) will be applied to the final environment, produced by the transformation.

4.3 Grammar Extensions

In this subsection the library to define and combine *extensible grammars* (like the one in Figure 1) and *syntax macros* (Figure 2) is presented. The idea is to see grammar extensions as typed transformations to extend typed grammars.

We define an extensible grammar (*ExtGram*) and a syntax macro (*SyntaxMacro*) as a typed transformation that constructs a typed grammar. That is, a *Trafo* with *Productions* as the type of terms.

```
type GramTrafo = Trafo Unit Productions
type ExtGram env exp
  = GramTrafo env () (exp env)
type SyntaxMacro env imp exp
  = GramTrafo env (imp env) (exp env)
```

Both extensible grammars and syntax macros have to export a list of non-terminals *exp* to be used in future extensions. The only difference between them is that a syntax macro has to import the list of non-terminals exported by the grammar it will extend, while an extensible grammar, given that it is an initial grammar, does not import anything.

Extensible grammar *ExtGram* and syntax macro *SyntaxMacro* export their starting point (with type (*Symbol start TNonT env*), thus a non-terminal) and their list of *exportable non-terminals* (*nts env*), which can be used and/or modified in future extensions.

```
data Export start nts env
  = Export (Symbol start TNonT env) (nts env)
```

The list of exportable non-terminals has to be passed in a *NTRRecord*, which is an implementation of extensible records very similar to the one in *HList*, with the difference that it has a type parameter *env* for the environment where the non-terminals point to. The idea of this kind of *type-level programming* is based on the use of types to represent type-level values, and classes to represent type-level types and functions. So, we define data types to represent a list-like structure both at the value and type level.

```
data NTCons nt v l env
  = NTCons (LSPair nt v TNonT env) (l env)
data NTNil env = NTNil
infixr 4 ^|
(^|) = NTCons
```

Each element (*Symbol a TNonT env*) of the list has an associated label *nt*. A type for each label has to be defined. The labels of our example (Figure 1) are:

```
data NTRoot
ntRoot = ⊥ :: NTRoot
data NTExp
ntExp = ⊥ :: NTExp
data NTTerm
ntTerm = ⊥ :: NTTerm
data NTFactor
ntFactor = ⊥ :: NTFactor
```

Given that *expr*, *term* and *factor* in the example have type *Symbol AttExpr TNonT env*, then the type of the list of exportable non-terminals is:

```
NTRoot NTExp AttExpr
(NTRoot NTTerm AttExpr
 (NTCons NTFactor AttExpr
 (NTNil env)
 env)
 env)
```

If we want this list to be a record, it should be ensured at compile time that there are no elements with the same label in it. This is accomplished by the class *NTRRecord*.

```
class NTRRecord nts
instance NTRRecord (NTNil env)
instance (NTRRecord (nts env), NotBelong nt (nts env))
  ⇒ NTRRecord (NTCons nt v nts env)
```

A type *r* is a *NTRRecord* if it is an empty list (*NTNil env*) or is a (*NTCons nt v nts env*) where the rest of the list (*nts env*) is a *NTRRecord* and the label *nt* does not belong to it.

```
class Fail err
data Duplicated nt
class NotBelong nt nts
instance NotBelong nt (NTNil env)
instance Fail (Duplicated nt)
  ⇒ NotBelong nt (NTCons nt v nts env)
```

```

instance NotFind nt1 (l env)
  ⇒ NotBelong nt1 (NTCons nt2 v nts env)

```

Overlapping instance detection⁷ is used to decide whether the *NotBelong* check fails. Verification of absence of duplicate labels proceeds recursively until it arrives at the empty list or at an instance where the labels match. When that happens a message about duplicate labels is generated by relying on the absence of an instance for class *Fail*: *Fail* doesn't have any instances at all, hence compilation terminates yielding an error message like:

```

No instance for (Fail (Duplicated nt)) ...

```

The class *GetNT* is used to lookup a non-terminal in a record.

```

class GetNT nt nts v | nt nts → v where
  getNT :: nt → nts → v
data NotFound nt
instance Fail (NotFound nt) ⇒ GetNT nt (NTNil env) r
  where getNT = ⊥
instance GetNT nt (NTCons nt v l env)
  (Symbol v TNonT env)
  where getNT _ (NTCons f _) = symbolNTField f
instance GetNT nt1 (l env) r
  ⇒ GetNT nt1 (NTCons nt2 v l env) r
  where getNT nt (NTCons _ l) = getNT nt l

```

We will not go into more details here, but its implementation is similar to the *NotBelong* case with the differences that *GetNT* fails when the label is not found (the search reaches *NTNil*), and when the label is found the non-terminal is returned.

To export the defined non-terminals is to inject an *Export* in the transformation in order to return it as output.

```

exportNTs :: NTRecord (nts env)
  ⇒ GramTrafo env (Export start nts env)
  (Export start nts env)
exportNTs = returnA

```

Thus, the definition of an extensible grammar, like the one in Figure 1, has the following shape⁸:

```

prds = proc () → do
  ...
  exportNTs < export

```

where *export* is a value of type *Export*. The definition of a syntax macro, like the one in Figure 2, has the shape:

```

prds' = proc (Export start nts) → do
  ...
  exportNTs < (Export start' nts')

```

To add a new non-terminal to the grammar is to add a new term to the environment.

```

addNT :: GramTrafo env [Prod a env]
  (Symbol a TNonT env)
addNT = proc p → do
  r ← newSRef < PS p
  returnA < Nont r

```

The input to *addNT* is the list of alternative productions for the non-terminal and the output is a non-terminal symbol, i. e. a reference to the non-terminal in the grammar. Thus, when in Figure 1 we write:

⁷ We did it to keep the code as simple as possible, alternatives to avoid overlapping can be found in (Kiselyov et al. 2004).

⁸ Using arrow's syntax (Paterson 2001)

```

exp ← addNT < [...]

```

we are adding the non-terminal for the expressions, with the list of 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.

Adding new productions to an existing non-terminal translates to concatenating the new productions with the existing list of productions of the non-terminal.

```

addProds :: GramTrafo env
  (Symbol a TNonT env, [Prod a env]) ()
addProds = proc (Nont nt, prds) → do
  updateFinalEnv <
  updateEnv (λ(PS ps) → PS $ prds ++ ps) nt

```

In Figure 2 there are examples of adding productions to the non-terminals *exp* and *factor*.

A grammar extension is the composition of two transformations, the first one representing an extensible grammar and the second one representing a syntax macro.

```

extendGram :: (NTRecord (nts env)
  , NTRecord (nts' env))
  ⇒ ExtGram env (Export start nts)
  → SyntaxMacro env (Export start nts)
  (Export start' nts')
  → ExtGram env (Export start' nts')
extendGram g sm = g >>> sm

```

We defined *extendGram* to restrict the types of the composition. Two syntax macros can be composed just by using (>>>).

To close a grammar is to run the *Trafo*, in order to obtain the Grammar, and apply the optimization functions *leftcorner* and *leftfactoring*. The type of the start non-terminal *a* is the type of the resulting grammar.

```

closeGram :: (∀ env. ExtGram env (Export a nts))
  → Grammar a
closeGram prds = case runTrafo prds Unit () of
  Result _ (Export (Nont r) _) gram
  → (leftfactoring.leftcorner) $ Grammar r gram

```

Once the grammar is closed the following functions are used to obtain the associated parser and parse.

```

compile :: Grammar a → Parser Token a
parse :: Parser Token a → [Token] → ParseResult a

```

5. Related Work

Although syntax-macros are not commonly supported in typed languages, there is a long tradition in languages like Lisp (Weise and Crew 1993), Scheme (Fisher and Shivers 2006), Prolog (Abramson 1984), and more recently Stratego (Bravenboer 2008). 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 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-languages, that is, that lack Scheme’s syntax extension ability- we find that we miss it greatly*”. Having made this observation they introduce the Ziggurat 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 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 Java-script, that lack Haskell’s type and class system- we find that we miss it greatly*”.

One might object that in the code 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, but the good news is that the complexity is neatly hidden in a couple of libraries, thus avoiding the need to know about the underlying type level wizardry.

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 (Kastens and Waite 1994; Gray et al. 1992; Swierstra et al. 1999; Ekman and Hedin 2007a,b; Mernik and Žumer 2005; Van Wyk 2007); the more aspects are being combined the more the attribute grammar approach for describing static semantics is to be preferred. Our experience with the Utrecht Haskell Compiler (UHC) (Dijkstra et al. 2009; Dijkstra 2005), which is completely structured as a composition of separately defined aspects and variants, has shown us the usefulness of this approach. Some non-terminals of the abstract syntax tree inside of UHC have over 15 different attributes defined for the non-terminal, with intricate data flow patterns between them. Compilers for modern languages have to deal with many aspects; to be able to deal with those aspects separately facilitates both incremental development as well as relatively easy experimentation and extension. Changes often are restricted to a few files dealing with a single aspect.

6. Conclusions and Future Work

With the combination of the techniques we have developed over the years our dream has finally come true: the possibility to construct a complete compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments. 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 *HList* is costly since getting at an individual attribute incurs selection from a (possibly deeply) nested Cartesian product. It is however our experience that a compiler spends most of its time in the auxiliary code for type-checking and -inferencing and (global) optimization, and thus for a modest language defined by a limited set of attributes our 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 *HTree* like structure, thus lowering the nesting depth of the top *HList* products.

Building the complete compiler from scratch as a collection of syntax-macros and fine-grained aspect definitions is probably not always the best approach; large parts of the compiler will be shared by all users, and there is no reason to use the relatively expensive techniques for extendability all over the compiler as long as the core compiler remains extensible. The situation then becomes similar to the way \LaTeX is distributed and used: a single large collection of standard definitions in a preprocessed format file, and a large col-

lection of packages which may be imported at will. In the same way the already existing attribute-grammar based description of UHC can be used to generate such an extensible core compiler. In this way we provide default definitions for all aspects, each of which can be redefined. For example, a pretty-printing attribute *pp* may be redefined by adding an extra aspect *updated_pp*, which borrows its default definition from the *pp* aspect in the core compiler. 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.

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 (Johnsson 1987; Kuiper and Swierstra 1986; de Moor et al. 2000; Oege de Moor and Swierstra 2000). Unfortunately this elegant approach fails when large trees are to be attributed; a lazy evaluation scheduling first builds a large dependency graph in memory, and only when this large graph has been constructed some real work is done. 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 analysis (Kastens 1980). However, the *uuage* 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 become visible. Since these dependencies usually reflect the way the compiler programmer thinks about his attribute grammars (Middelkoop et al. 2010) we expect this extra burden to be bearable.

A third problem arises from the way we construct our parsers and combine our aspects. *Every time* we use the compiler the complete parser and attribute grammar is reconstructed from scratch; the individual grammar components are constructed first (*prds* and *prds’*), then they are merged into a single large grammar (the calls to *extendGram*) 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 and similar sequence of steps is done for the aspects. The final parser and evaluator however do in no way depend on the input of the compiler; they are a global constant Haskell values (i.e. are in constant applicative form). Having such values repeatedly evaluated 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 the 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 ran once more; in this way the evaluation of *caf*’s is memoised over different runs of the program.

A final, quite subtle, problem lies in the current implementation of the Left-Corner Transform (Baars et al. 2009b). In the way this transform is formulated (and implemented) it is assumed that the non-terminals which occur at the beginning of a production do not produce the empty string. By extending the current grammar analysis this can in principle be discovered (Swierstra 2000), and the grammar can be transformed into an equivalent form without such offending non-terminals. This code still has to be written. For the time being an error message is generated when this situation occurs.

We invite those who like our approach, but do think that the notation is a bit too cumbersome, to suggest a nice syntax for defining the language fragments. Your favorite notation can be accommodated from now on!

7. Acknowledgments

We want to thank Arie Middelkoop and Chris Eidhof for commenting on the paper. We want to thank Joost Rommes for his initial exploration of attribute redefinitions in 2003.

References

- Harvey Abramson. Definite clause translation grammars. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1984.
- Arthur Baars, S. Doaitse Swierstra, and Marcos 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, 2009a. ACM.
- Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002. ISBN 1-58113-487-8.
- I. Baars, Arthur, Doaitse Swierstra, S., and Marcos 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, 2009b.
- Martin Bravenboer. *Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2008.
- Oege de Moor, L. Peyton Jones, Simon, and Van Wyk, Eric. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag. ISBN 3-540-41172-0.
- Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.
- Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg*, volume 5083, pages 57–74, Berlin, Heidelberg, 2008. Springer-Verlag.
- Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6.
- Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007a. ACM.
- Torbjörn Ekman and Görel Hedin. The jastadd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3): 14–26, 2007b. ISSN 0167-6423.
- David Fisher and Olin Shivers. Static analysis for syntax objects. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 111–121, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3.
- Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992. ISSN 0001-0782.
- Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, 2005.
- Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
- Ralf Hinze and Ross Paterson. Derivation of a typed functional lr parser, 2003.
- John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000. ISSN 0167-6423.
- Thomas 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.
- U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Inf.*, 31(7):601–627, 1994. ISSN 0001-5903.
- Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13: 229–256, 1980.
- Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4.
- M. F. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. RUU-CS 86-16, Department of Computer Science, 1986.
- B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966. ISSN 0001-0782.
- Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.
- Marjan Mernik and Viljem Žumer. Incremental programming language development. *Computer languages, Systems and Structures*, 31:1–16, 2005.
- Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Visit Functions for the Semantics of Programming Languages. In *Workshop on Generative Programming (to appear)*, March 2010.
- Kevin Backhouse Oege de Moor and S. Doaitse Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. ISSN 0350-5596. Special Issue: Attribute grammars and Their Applications.
- Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, volume LNCS 3286, pages 136 – 167, October 2004.
- Ross Paterson. A new notation for arrows. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 229–240, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.
- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gads. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340.
- S. Doaitse Swierstra. Parser combinators: from toys to tools. In Graham Hutton, editor, *Haskell Workshop*, 2000.
- S. Doaitse Swierstra. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, pages 252–300, 2008.
- S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João A. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of LNCS, pages 150–206. Springer-Verlag, 1999.
- Eric Van Wyk. Implementing aspect-oriented programming constructs as modular language extensions. *Sci. Comput. Program.*, 68(1):38–61, 2007. ISSN 0167-6423.
- Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 245–256, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.
- Daniel Weise and Roger Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4.