Grammar Fragments Fly First-Class

Marcos Viera
S. Doaitse Swierstra
Atze Dijkstra

Technical Report UU-CS-2011-032
Nov 2011

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

ISSN: 0924-3275

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

Grammar Fragments Fly First-Class

Marcos Viera*, S. Doaitse Swierstra and Atze Dijkstra

December 14, 2011

Abstract

We present a Haskell library[1] for describing grammars explicitly, using typed abstract syntax with references. We can analyze, transform and finally generate parsers from this representation. What makes our approach special is that we can combine grammar fragments on the fly, i.e. after they have been compiled. Thus grammar fragments become real first-class values.

We show how by using this technique we can extend an initial, more limited grammar embedded in a compiler with extra syntactic constructs. Existing grammars can be freely extended by both adding new non-terminals and by adding new productions to the existing non-terminals, with no restrictions being imposed on the individual fragments, nor on the structure as a whole.

1 Introduction

We have many different ways to represent grammars and grammatical structures: be it in implicit form using *conventional parser combinators* which directly implement the parsing semantics or as *typed abstract syntax*. Each approach comes with its own advantages and disadvantages. The former, being a domain specific embedded language, makes direct use of the abstraction and naming system of the underlying host language. This implicit representation however also does have its disadvantages: we can only perform a limited form of grammar analysis and transformation. The latter approach, which does give us full access to the complete domain specific program, comes with a more elaborate naming system and transforming such programs necessitates to provide proofs (encoded in the Haskell type system) that the types remain correct during transformation.

There are several application areas for the latter approach. One of the cases where one wants to be able to compose grammar fragments is when a user extends the syntax of the base language with his own notation. In order to do so several steps have to be undertaken: he has to extend the underlying context-free grammar and he has to give the semantics of these new constructs. Once the extensions are given we have to construct the parser for the complete language, and have to be sure that the newly defined or redefined semantics become part of the semantics of the complete language. In a more limited way this is done by e.g. the quasi quoting mechanism of Template Haskell [7, 13]. This approach however has its limitations, since here the separate pieces are clearly separated from the host language, whereas we want to describe "invisible extensions".

In an earlier paper [17] we have shown how to define the final semantics of a composed language in terms of composible attribute grammar fragments and in [15] we have shown how to compose grammar fragments for a limited class of grammars, i.e. those which describe Haskell data types. These latter grammars have a convenient property: productions cannot derive the empty string, which is a necessary pre-condition for the Left-Corner Transform (LCT) which is to be applied later to remove potential left-recursion.

In general this restriction however does not hold and hence the questions remains how to make sure that the final context-free grammar, which is thus composed of a potentially very large

*Instituto de Computación, Universidad de la República, Montevideo, Uruguay

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

```
Grammar:

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

```
Haskell code:

    prds = proc () → do
      rec root   ← addNT ≺ [[ semRoot exp    ]]
          exp    ← addNT ≺ [[ semLet  "let" var "=" exp
                                      "in"  exp      ]]
                      <|> [[ semAdd  term "+" exp   ]]
                      <|> [[ id      term  ]]
          term   ← addNT ≺ [[ semMul  term "*" factor ]]
                      <|> [[ id      factor ]]
          factor ← addNT ≺ [[ semCst  int ]]
                      <|> [[ semVar  var ]]
      exportNTs ≺ exportList root $ export ntExp    exp
                                  . export ntTerm   term
                                  . export ntFactor factor

    gram = closeGram prds
```

Figure 1: Initial language

number of smaller CFG fragments, is parseable. There should be no need to restrict the user from abstaining from left-recursion, nor from the use of empty productions.

In this paper we describe an *unrestricted*, *applicative* interface for such grammar descriptions, describe how they can be combined, and how they can be transformed so they fulfill the initial requirements imposed by the Left-Corner Transform. After applying this transform, which we have described elsewhere, one finally gets the required parser by interpreting the final structure using a conventional combinator library.

In section 2 we describe the "user-interface" to our library, in section 3 we describe the type structures used internally to represent our grammar fragments and its applicative interface, whereas in section 4 we describe internal structures of the grammars which make it possible to extend the grammars. In section 5 present a transformation to remove the empty productions of a grammar fragment in order to be able to apply Left-Corner Transform. In section 6 we extend our grammar representation with a fixpoint-like combinator. Finally in section 7 we conclude and discuss some future work.

2 Context-Free Grammar

In this section we show how to express a part of a context free grammar. Our running example will be a simple expression language, to which we will refer to as the *initial grammar*. Figure 1 shows this initial grammar, together with the almost isomorphic Haskell code corresponding to this language fragment.

Note that this concrete grammar uses the syntactic categories *root*, *exp*, *term* and *factor* to describe the operator precedences.

A language implementer has to provide the Haskell code, expressing himself using our combinator library (of course one might generate this from the grammar description) and the arrow-interface; in the structure of the code we immediately recognize the context-free grammar just given. Each non-terminal of the CFG is introduced (using *addNT*) by defining a list of productions (alternatives) separated by `<|>` operators, where each production contains a sequence of elements to be recognised. The alternatives are expressed in so-called *applicative style*, using the idiomatic brackets $\llbracket$ and $\rrbracket$ which delineate the description of a production from the rest of the Haskell code. These brcakets are inspired by the *idioms* approach as introduced by Conor McBride [9][2]. We have made those definitions a bit more specific and have added extra instances which deal with special cases, such as single characters (`'*'`) and strings (`"let"`) in such definitions. These symbols define parts of a parser which do not bear any meaning. The result of their parsing is discarded and not taken into further account. The brackets $\llbracket$ and $\rrbracket$ are syntactic sugar for *iI* and *Ii*, respectively, where for example the production description:

$\llbracket$ *semMul term* `"*"` *factor* $\rrbracket$

is equivalent to:

pure semMul `<*>` *sym term* `<*` *tr* `"*"` `<*>` *sym exp*.

The operator (`<*`), applied in the case of terminals, is a sequence operator that ignores the value of the second argument. A naive implementation of (`<*`) is:

a `<*` $b = pure\ const$ `<*>` a `<*>` b

The functions starting with *sem* (e.g. *semMul*) describe how to combine the semantic values of the non-terminals in the right-hand side of the production into the semantic value of the left hand side of that production. We call them *semantic functions*, since they give a semantics to the production. We will not go into the details of how to construct such semantic functions in this paper. They have to be provided elsewhere, e.g. by writing plain Haskell code full of monad transformers. We prefer to generate such functions from an attribute grammar using the *uuagc*-system, or describe them directly in Haskell using attribute grammar descriptions embedded as a domain specific language in Haskell as described in [17]. In all these cases 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 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 (by convention of the form *nt*..., which is actually a single value of a specific type) and the collection of right hand sides. These right hand sides 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 a parser integrated with the semantics for the language starting from the first non-terminal, which in our case is *root*.

2.1 Language Extension

In this subsecion we show how to extend the language just defined with a couple extra productions; we add conditional expressions, conditions and the possibility to use parentheses to influence the way expressions are parsed:

exp ::= ...
  | `"if"` *cond* `"then"` *exp* `"else"` *exp*

[2] `http://www.haskell.org/haskellwiki/Idiom_brackets`

$$
\begin{aligned}
&cond \;\; ::= exp\ \texttt{"=="}\ exp \\
&\qquad\quad \mid exp\ \texttt{">"}\ \ exp \\
&factor ::= \;... \\
&\qquad\quad \mid \texttt{"("}\ exp\ \texttt{")"}
\end{aligned}
$$

This language extension $prds'$ is defined as a closed Haskell value by itself, which accesses an already existing set of productions (*imported*) and builds an extended set, as shown in Figure 2.

$$
\begin{aligned}
&prds' = proc\ imported \rightarrow \mathbf{do} \\
&\quad \mathbf{let}\ exp \quad\ = getNT\ ntExp \quad\ imported \\
&\quad \mathbf{let}\ factor = getNT\ ntFactor\ imported \\
&\quad \mathbf{rec}\ addProds \prec (exp, \quad [\![\ semIf \quad \texttt{"if"} \quad cond \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{"then"}\ exp \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{"else"}\ exp\ \ ]\!]) \\
&\qquad\quad cond \leftarrow addNT \ \prec\ \ [\![\ semEq\ exp\ \texttt{"=="}\ exp\ ]\!] \\
&\qquad\qquad\qquad\qquad\quad \texttt{<|>}\ [\![\ semGr\ exp\ \texttt{">"}\ \ exp\ ]\!] \\
&\qquad\quad addProds \prec (factor, [\![\ semPar\ \texttt{"("}\ exp\ \texttt{")"}\ ]\!]) \\
&\quad exportNTs \prec extendExport\ imported \\
&\qquad\qquad\qquad\qquad\qquad\qquad (export\ ntCond\ cond) \\
&gram' = closeGram\ (prds\ \texttt{+>>}\ prds')
\end{aligned}
$$

Figure 2: Language Extension

For each non-terminal to be extended 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 *exp* the new production for conditional is added by:

$$
\begin{aligned}
&\mathbf{let}\ exp = getNT\ ntExp\ imported \\
&addProds \prec (exp, [\![\ semIf\ \texttt{"if"} \quad cond \\
&\qquad\qquad\qquad\qquad\quad\ \texttt{"then"}\ exp \\
&\qquad\qquad\qquad\qquad\quad\ \texttt{"else"}\ exp\ ]\!])
\end{aligned}
$$

This code shows how to combine the previously defined productions with the newly defined productions into the extended grammar. New non-terminals can be added as well using *addNT*; in the example we add the non-terminal *cond* to represent some simple conditions:

$$
\begin{aligned}
&cond \leftarrow addNT \ \prec\ \ [\![\ semEq\ exp\ \texttt{"=="}\ exp\ ]\!] \\
&\qquad\qquad\qquad\quad \texttt{<|>}\ [\![\ semGr\ exp\ \texttt{">"}\ \ exp\ ]\!]
\end{aligned}
$$

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:

$$
\begin{aligned}
&exportNTs \prec extendExport\ imported \\
&\qquad\qquad\qquad\qquad\quad (export\ ntCond\ cond)
\end{aligned}
$$

Because both *prds* and $prds'$ are 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 here.

3 Grammar Representation

Having described how to define individual language fragments and how to combine them, in this and the following sections we will embark on the description of the internals of our library itself.

Since we do not want to put severe constraints on the use of the libraries when composing a context free grammar from a collection of individual fragments, we will have to cope with a large class of grammars; we require from all individual components that they can be safely composed. So we will have to deal with left-recursive grammars and grammars which are, once composed, for example not LALR(1). In previous work [2, 1, 3, 15] we have developed a serie of techniques to deal with such grammars, which are based on typed representation of grammars and typed transformations of these grammars, for example to remove left recursion. In this section we introduce a typed representation of grammars that provides an easy way to describe grammars and allows the use of these techniques.

We represent grammars as typed abstract syntax, using Generalised Algebraic Data Types [12]. The idea, proposed in [3], is to indirectly refer to non-terminals via references encoded as types. Such references type-index into an environment holding the actual descriptions of the 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 [10]. and was extended in [3] 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 into 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* defined by 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 *FinalEnv* forces environments *def* and *use* to coincide, thereby closing an environment and thus making sure that all references point to some definition, and that those definitions describe values of the expected types.

To express that an environment is close we introduce the type *FinalEnv* whch guarantees that the *use* and*def* are the same.

```
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 the visible part of its type.

```
data EG
data CG
data Grammar s a
```

$$= \forall\ env.Grammar\ (Ref\ a\ env)\ (FinalEnv\ (Productions\ s)\ env)$$

newtype *Productions s a env*
$= PS\ \{unPS :: [Prod\ s\ a\ env]\}$

The type s represents the state of the grammar, that is: EC if the grammar can contain empty productions and CG if the grammar does not contain empty productions.

We represent productions differently from [3] and [15], where a production is a sequence of symbols terminated with a function representing the semantics. Here we represent productions in an *applicative-style*; i.e. with a couple of constructors *Pure* and *Star* analogous to the *pure* function and <*> operator of applicative functors:

$$
\begin{array}{l}
\textbf{data}\ Prod\ s\ a\ env\ \textbf{where} \\
\quad \begin{array}{lll}
Pure & :: & a \to Prod\ s\ a\ env \\
Star & :: & Prod\ s\ (a \to b)\ env \\
 & & \to Prod\ s\ a\ env \to Prod\ s\ b\ env \\
Sym & :: & Symbol\ a\ t\ env \to Prod\ s\ a\ env \\
 & & \\
FlipStar & :: & Prod\ CG\ a\ env \\
 & & \to Prod\ CG\ (a \to b)\ env \to Prod\ CG\ b\ env
\end{array}
\end{array}
$$

FlipStar is a variant of *Star* with its arguments in the reverse order. By imposing s to be CG, we restrict *FlipStar* to be included only in grammars without empty productions. *Sym* is a special case of *pure* that lifts a symbol to a production. 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 an environment labelled with *env*. A normal terminal contains the literal string it represents. 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.

$$
\begin{array}{l}
\textbf{data}\ TTerm \\
\textbf{data}\ TNonT \\
\textbf{data}\ TAttT \\
\textbf{data}\ Symbol\ a\ t\ env\ \textbf{where} \\
\quad \begin{array}{llll}
Term & :: String \to & Symbol\ String & TTerm\ env \\
Nont & :: Ref\ a\ env \to & Symbol\ a & TNonT\ env \\
TermInt & :: & Symbol\ Int & TAttT\ env \\
TermChar & :: & Symbol\ Char & TAttT\ env \\
TermVarid & :: & Symbol\ String & TAttT\ env \\
TermConid & :: & Symbol\ String & TAttT\ env \\
TermOp & :: & Symbol\ String & TAttT\ env
\end{array}
\end{array}
$$

The type parameter t indicates, at the type-level, whether a *Symbol* is a terminal (type *TTerm*) for which the result is (usually) discarded, a non-terminal (*TNonT*) or an attributed terminal (*TAttT*) in the value of which we are interested.

In order to make our code more readable we introduce the following smart constructors for terminals:

$$
\begin{array}{ll}
trm & = Term \\
int & = TermInt \\
char & = TermChar \\
var & = TermVarid \\
con & = TermConid \\
op & = TermOp
\end{array}
$$

3.1 From Grammar to Parser

A grammar can be compiled into a parser, which can then be used to *parse* a String into a *ParseResult* containing a semantic value of type a.

$$
\begin{aligned}
&compile :: Grammar\ CG\ a \to Parser\ a\\
&parse\quad :: Parser\ a \to String \to ParseResult\ a
\end{aligned}
$$

We translate to the `uu-parsinglib` parser combinator library [14], that has an *Applicative* (and *Alternative*) interface. Thus, *compile* translates a *Productions* list as a sequence of parsers combined by `<|>`. The *Prod* constructors *Star*, *FlipStar* and *Pure* are translated to `<*>`, `<**>` and *pure*, respectively. Terminals are translated to terminal parsers and non-terminal references are looked-up into an environment containing the translated productions for each non-terminal.

$$
\begin{aligned}
&\mathbf{newtype}\ Const\ f\ a\ s = C\ \{unC :: f\ a\}\\
&compile :: Grammar\ CG\ a \to Parser\ a\\
&compile\ (Grammar\ (start :: Ref\ a\ env)\ rules)\\
&\quad = unC\ (lookupEnv\ start\ result)\ \mathbf{where}\\
&\quad result =\\
&\qquad mapEnv\\
&\qquad (\lambda(PS\ ps) \to C\ (foldr1\ (\texttt{<|>})\ [comp\ p \mid p \leftarrow ps]))\\
&\qquad rules\\
&\quad comp :: \forall\ t.Prod\ CG\ t\ env \to Parser\ t\\
&\quad comp\ (Star\ \ x\ y) \qquad\quad = comp\ x\ \texttt{<*>}\ \ comp\ y\\
&\quad comp\ (FlipStar\ x\ y) \quad = comp\ x\ \texttt{<**>}\ comp\ y\\
&\quad comp\ (Pure\ x) \qquad\quad\ = pure\ x\\
&\quad comp\ (Sym\ (Term\ t)) \quad = pTerm\ t\\
&\quad comp\ (Sym\ (Nont\ n)) \quad = unC\ (lookupEnv\ n\ result)\\
&\quad comp\ (Sym\ TermInt) \quad\ = pInt\\
&\quad comp\ (Sym\ TermChar) \ \ = pChr\\
&\quad comp\ (Sym\ TermVarid) = pVar\\
&\quad comp\ (Sym\ TermConid) = pCon\\
&\quad comp\ (Sym\ TermOp) \quad\ = pOp\\
&mapEnv\ ::\ \ (\forall\ a.f\ a\ s \to g\ a\ s)\\
&\qquad\qquad \to Env\ f\ s\ env \to Env\ g\ s\ env\\
&mapEnv\ \_\ Empty \quad\ = Empty\\
&mapEnv\ f\ (Ext\ r\ v) = Ext\ (mapEnv\ f\ r)\ (f\ v)
\end{aligned}
$$

Since the `uu-parsinglib` performs a breadth-first search it will parse a large class of grammars without any further *try* or *cut*-like annotations; the only requirement it imposes is that the grammar is not left-recursive (which holds since we will apply the LCT before compiling the grammar into a parser) and that the grammar is unambiguous. This latter property is unfortunately undecidable; fortunately it is trivial to generate a parser version which can handle ambiguous grammars too, since the `uu-parsinglib` library contains provisions for this.

We define the relation *equality under compilation* ($\overset{c}{\equiv}$) as:

$$
a \overset{c}{\equiv} b \Leftrightarrow compile\ a \equiv compile\ b
$$

Since *Parser* is an applicative functor, we can translate its laws to elements of type *Prod* under compilation:

$$
\begin{aligned}
&Pure\ id\ \text{`}Star\text{`}\ v && \overset{c}{\equiv} v\\
&Pure\ (.)\ \text{`}Star\text{`}\ u\ \text{`}Star\text{`}\ v\ \text{`}Star\text{`}\ w && \overset{c}{\equiv} u\ \text{`}Star\text{`}\ v\ \text{`}Star\text{`}\ w\\
&Pure\ f\quad \text{`}Star\text{`}\ Pure\ x && \overset{c}{\equiv} Pure\ (f\ x)\\
&u\qquad\ \ \text{`}Star\text{`}\ Pure\ y && \overset{c}{\equiv} Pure\ (\$y)\ \text{`}Star\text{`}\ u
\end{aligned}
$$

3.2 Applicative Interface

We want the type *Productions* to be an instance of *Applicative* and *Alternative*, in order to have an applicative interface to describe productions. However, this is impossible due to the order of its type parameters; we need a to be the last parameter[3]. Thus, we define the type *PreProductions* for descriptions of alternative productions. Notice that the productions cannot include *FlipStar*s.

$$\textbf{newtype}\ \mathit{PreProductions}\ \mathit{env}\ a$$
$$\quad = \mathit{PP}\ \{\mathit{unPP} :: [\mathit{Prod}\ \mathit{EG}\ a\ \mathit{env}]\}$$

The translation from *PreProductions* to *Productions* is trivial:

$$\mathit{prod} :: \mathit{PreProductions}\ \mathit{env}\ a \rightarrow \mathit{Productions}\ \mathit{EG}\ a\ \mathit{env}$$
$$\mathit{prod}\ (\mathit{PP}\ \mathit{ps}) = \mathit{PS}\ \mathit{ps}$$

Now we can define the instances of *Applicative* and *Alternative* for (*PreProductions env*):

$$\begin{array}{l}\textbf{instance}\ \mathit{Applicative}\ (\mathit{PreProductions}\ \mathit{env})\ \textbf{where} \\ \quad \mathit{pure}\ f = \mathit{PP}\ [\mathit{Pure}\ f] \\ \quad (\mathit{PP}\ f)\ \texttt{<*>}\ (\mathit{PP}\ g) = \mathit{PP}\ [\mathit{Star}\ f'\ g' \mid f' \leftarrow f, g' \leftarrow g] \\ \textbf{instance}\ \mathit{Alternative}\ (\mathit{PreProductions}\ \mathit{env})\ \textbf{where} \\ \quad \mathit{empty} = \mathit{PP}\ [\,] \\ \quad (\mathit{PP}\ f)\ \texttt{<|>}\ (\mathit{PP}\ g) = \mathit{PP}\ (f +\!\!+\ g)\end{array}$$

Note that we are dealing with lists of alternative productions. Thus, the alternative operator (<|>) takes two lists of alternatives and just appends them. In the case of sequential application (<*>) a list of productions is generated with all the possible combinations of the operands joined with a *Star*.

We also defined smart constructors for symbols: *sym* for the general case and *tr* for the special case where the symbol is a terminal.

$$\begin{array}{l}\mathit{sym}\ :: \mathit{Symbol}\ a\ t\ \mathit{env} \rightarrow \mathit{PreProductions}\ \mathit{env}\ a \\ \mathit{sym}\ s = \mathit{PP}\ [\mathit{Sym}\ s] \\ \mathit{tr}\quad :: \mathit{String} \rightarrow \mathit{PreProductions}\ \mathit{env}\ \mathit{String} \\ \mathit{tr}\quad s = \mathit{PP}\ [\mathit{Sym}\ (\mathit{Term}\ s)]\end{array}$$

4 Extensible Grammars

In this section we present the library to define and combine *extensible grammars* (like the one in Figure 1) and *grammar extensions* (Figure 2). The key idea is to see the definition, and possibly future extensions, of a grammar as a typed transformation that introduces new non-terminals into a typed grammar.

4.1 TTTAS

Grammar definitions and extensions are defined as typed transformations of values of type *Grammar*. For example, both *prds* and *prds*$'$ of Figures 1 and 2 are typed transformations: while *prd* starts with an empty context-free grammar and transforms it by adding the non-terminals *root*, *exp*, *term* and *factor*, the grammar extension *prd*$'$ continues the transformation started by *prd* and modifies some of the non-terminals. Notice that a *Grammar* is a collection of mutually recursive

[3]We cannot just redefine *Productions* with this order, because we need the original one for the transformations we will introduce later.

typed structures; thus, performing transformations while maintaining the whole collection well-typed is non-trivial. The rest of this sub-section is a short introduction to the API of TTTAS[4] (Typed Transformations of Typed Abstract Syntax), the library we use to implement our transformations. TTTAS is based on the *Arrow* type *Trafo* [5], which represents typed transformation steps, (possibly) extending an environment *Env*.

$$\mathbf{data}\ \mathit{Trafo}\ m\ t\ s\ a\ b$$

The arguments are the types of: the meta-data m (i.e., state other than the environment we are constructing), the terms t stored in the environment, the final environment s, the arrow-input a and arrow-output b. Thus, instances of the classes *Category* and *Arrow* are implemented for ($\mathit{Trafo}\ m\ t\ s$), which provides a set of functions for constructing and combining *Trafo*s. Some of these functions which we will refer to are:

- Identity arrow (like *return* in monads)

  $$\mathit{returnA} :: \mathit{Arrow}\ a \Rightarrow a\ b\ b$$

- Lifting a function to an arrow

  $$\mathit{arr} :: \mathit{Arrow}\ a \Rightarrow (b \rightarrow c) \rightarrow a\ b\ c$$

- Left-to-right composition

  $$(\ggg) :: \mathit{Category}\ \mathit{cat} \Rightarrow \mathit{cat}\ a\ b \rightarrow \mathit{cat}\ b\ c \rightarrow \mathit{cat}\ a\ c$$

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

$$\mathit{loop} :: a\ (b, d)\ (c, d) \rightarrow a\ b\ c$$

There also exists a convenient notation [11] for *Arrow*s, which is inspired by the **do**-notation for *Monad*s.

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.

$$\begin{aligned}\mathit{runTrafo} &:: (\forall\ s.\mathit{Trafo}\ m\ t\ s\ a\ (b\ s)) \rightarrow m\ () \rightarrow a\\ &\rightarrow \mathit{Result}\ m\ t\ b\end{aligned}$$

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 (hence the use of *FinalEnv*).

$$\begin{aligned}&\mathbf{data}\ \mathit{Result}\ m\ t\ b\\ &\quad = \forall\ s.\mathit{Result}\ (m\ s)\ (b\ s)\ (\mathit{FinalEnv}\ t\ s)\end{aligned}$$

New terms can be added to the environment by using the function *newSRef*. It takes the term of type $t\ a\ s$ to be added as input and yields as output a reference of type $\mathit{Ref}\ a\ s$ that points to this term in the final environment:

$$\begin{aligned}&\mathit{newSRef} :: \mathit{Trafo}\ \mathit{Unit}\ t\ s\ (t\ a\ s)\ (\mathit{Ref}\ a\ s)\\ &\mathbf{data}\ \mathit{Unit}\ s = \mathit{Unit}\end{aligned}$$

[4] http://hackage.haskell.org/package/TTTAS

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

Functions ($FinalEnv\ t\ s \to FinalEnv\ t\ s$) 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.

$$
\begin{aligned}
&updateFinalEnv :: Trafo\ m\ t\ s \\
&\qquad\qquad\qquad\qquad (FinalEnv\ t\ s \to FinalEnv\ t\ s)\ ()
\end{aligned}
$$

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.

4.2 Grammar Extensions

In this subsection the API of the library to define and combine *extensible grammars* (like the one in Figure 1) and *grammar extensions* (Figure 2) is presented. A grammar extension can be seen as a serie of typed transformation steps that can add new non-terminals to a typed grammar and/or modify the definition of already existing non-terminals.

We define an extensible grammar type (*ExtGram*) for constructing intitial grammars from scratch and a grammar extension type (*GramExt*) as a typed transformation that extends a typed extensible grammar. In both cases a *Trafo* uses the *Productions* as the type of terms defined in the environment being carried.

```
type ExtGramTrafo = Trafo Unit (Productions EG)
type ExtGram env              start′ nts′
  = ExtGramTrafo env ()
                    (Export start′ nts′ env)
type GramExt env start nts start′ nts′
  = ExtGramTrafo env (Export start  nts  env)
                    (Export start′ nts′ env)
```

4.2.1 Exportable non-terminals

Both extensible grammars and grammar extensions have to export the starting point $start'$ and a list of *exportable non-terminals* nts' to be used in future extensions. The only difference between them is that a grammar extension has to import the elements (*start* and *nts*) exported by the grammar it will extend, while an extensible grammar, given that it is an initial grammar, does not import anything.

The exported (and imported, in the case of grammar extensions) elements have type *Export start nts env*, including the starting point (with type (*Symbol start TNonT env*), thus a non-terminal) and the list of exportable non-terminals (*nts env*).

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

The list of exportable non-terminals has to be passed in a *NTRecord*, which is an implementation of extensible records very similar to the one in the *HList* library [6], with the difference that it has a type parameter *env* for the environment where the non-terminals point into. So, we define data types to represent a list-like structure both at the value and type level.

$$
\begin{array}{l}
\textbf{data}\ \mathit{NTCons}\ \mathit{nt}\ v\ l\ \mathit{env} \\
\quad = \mathit{NTCons}\ (\mathit{LSPair}\ \mathit{nt}\ v\ \mathit{TNonT}\ \mathit{env})\ (l\ \mathit{env}) \\
\textbf{data}\ \mathit{NTNil}\ \mathit{env} = \mathit{NTNil}
\end{array}
$$

Each element of the list is a field of type *LSPair*, that associates a label *nt* (a phantom type [4]) with a symbol of type (*Symbol a t env*).

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{LSPair}\ \mathit{nt}\ a\ t\ \mathit{env} \\
\quad = \mathit{LSPair}\ \{\mathit{symLSPair} :: (\mathit{Symbol}\ a\ t\ \mathit{env})\} \\
\textbf{infixr}\ 6 \in \\
(\in)\ \_ = \mathit{LSPair}
\end{array}
$$

Labels are used as type-level values; note that when constructing a field using $(\in)$ we just ignore the real value. For each label we have to define a unique type and a $\perp$ value to lift this type. The labels of our example (Figure 1) are:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{NTRoot}; & \mathit{ntRoot} & = \perp :: \mathit{NTRoot} \\
\textbf{data}\ \mathit{NTExp}; & \mathit{ntExp} & = \perp :: \mathit{NTExp} \\
\textbf{data}\ \mathit{NTTerm}; & \mathit{ntTerm} & = \perp :: \mathit{NTTerm} \\
\textbf{data}\ \mathit{NTFactor}; & \mathit{ntFactor} & = \perp :: \mathit{NTFactor}
\end{array}
$$

We have defined some functions to construct *Export* values:

$$
\begin{array}{ll}
\mathit{exportList}\ r\ \mathit{ext} & = \mathit{Export}\ r\ (\mathit{ext}\ \mathit{ntNil}) \\
\mathit{export}\ l\ \mathit{nt} & = \mathit{NTCons}\ (l \in \mathit{nt})
\end{array}
$$

Thus, the export list in Figure 1:

$$
\begin{array}{lll}
\mathit{exportList}\ \mathit{root}\ \$\ \mathit{export}\ \mathit{ntExp} & \mathit{exp} \\
\qquad\qquad\quad .\ \mathit{export}\ \mathit{ntTerm} & \mathit{term} \\
\qquad\qquad\quad .\ \mathit{export}\ \mathit{ntFactor} & \mathit{factor}
\end{array}
$$

is equivalent to:

$$
\begin{array}{l}
\mathit{Export}\ \mathit{root}\ (\mathit{NTCons}\ (\mathit{ntExp} \in \mathit{exp}) \\
\qquad\qquad (\mathit{NTCons}\ (\mathit{ntTerm} \in \mathit{term}) \\
\qquad\qquad (\mathit{NTCons}\ (\mathit{ntFactor} \in \mathit{factor}) \\
\qquad\qquad\ \mathit{NTNil})))
\end{array}
$$

Given that *expr*, *term* and *factor* in the example have types (*Symbol AttExpr TNonT env*), (*Symbol AttTerm TNonT env*) and (*Symbol AttFactor TNonT env*), respectively, where each *AttNT* is the semantic domain associated to the respective *NT*, the type of the list of exportable non-terminals is:

$$
\begin{array}{l}
\mathit{NTCons}\ \mathit{NTExpr}\ \mathit{AttExpr} \\
\qquad (\mathit{NTCons}\ \mathit{NTTerm}\ \mathit{AttTerm} \\
\qquad\qquad (\mathit{NTCons}\ \mathit{NTFactor}\ \mathit{AttFactor} \\
\qquad\qquad\qquad (\mathit{NTNil}\ \mathit{env}) \\
\qquad\qquad\qquad \mathit{env}) \\
\qquad\qquad \mathit{env}) \\
\qquad \mathit{env}
\end{array}
$$

If we want this list to be a record, it should be ensured at compile time it does not contain two elements with the same label. This is accomplished by the class *NTRecord*:

$$
\begin{array}{l}
\textbf{class } \mathit{NTRecord\ nts} \\
\textbf{instance } \mathit{NTRecord}\ (\mathit{NTNil\ env}) \\
\textbf{instance } (\mathit{NTRecord}\ (\mathit{nts\ env}), \mathit{IsNotElem\ nt}\ (\mathit{nts\ env})) \\
\qquad\qquad \Rightarrow \mathit{NTRecord}\ (\mathit{NTCons\ nt\ v\ nts\ env})
\end{array}
$$

A type r is a *NTRecord* 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 *NTRecord* and the label *nt* does not belong to it:

$$
\begin{array}{l}
\textbf{class } \mathit{Fail\ err} \\
\textbf{data } \mathit{Duplicated\ nt} \\
\textbf{class } \mathit{IsNotElem\ nt\ nts} \\
\textbf{instance } \mathit{IsNotElem\ nt}\ (\mathit{NTNil\ env}) \\
\textbf{instance } \mathit{Fail}\ (\mathit{Duplicated\ nt}) \\
\quad \Rightarrow \mathit{IsNotElem\ nt}\ \ (\mathit{NTCons\ nt}\ \ \mathit{v\ nts\ env}) \\
\textbf{instance } \mathit{IsNotElem\ nt1}\ (\mathit{l\ env}) \\
\quad \Rightarrow \mathit{IsNotElem\ nt1}\ (\mathit{NTCons\ nt2\ v\ nts\ env})
\end{array}
$$

Overlapping instance detection[5] is used to decide whether the *IsNotElem* 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.

$$
\begin{array}{l}
\textbf{class } \mathit{GetNT\ nt\ nts\ v} \mid \mathit{nt\ nts} \to v \ \textbf{where} \\
\quad \mathit{getNT} :: \mathit{nt} \to \mathit{nts} \to v \\
\textbf{data } \mathit{NotFound\ nt} \\
\textbf{instance } \mathit{Fail}\ (\mathit{NotFound\ nt}) \\
\quad \Rightarrow \mathit{GetNT\ nt}\ (\mathit{NTNil\ env})\ r \\
\quad \textbf{where } \mathit{getNT} = \bot \\
\textbf{instance } \mathit{GetNT\ nt}\ \ (\mathit{NTCons\ nt\ v\ l\ env}) \\
\qquad\qquad\qquad\quad (\mathit{Symbol\ v\ TNonT\ env}) \\
\quad \textbf{where } \mathit{getNT}\ \_\ (\mathit{NTCons}\ f\ \_) = \mathit{symbolNTField}\ f \\
\textbf{instance } \mathit{GetNT\ nt1}\ (\mathit{l\ env})\ r \\
\quad \Rightarrow \mathit{GetNT\ nt1}\ (\mathit{NTCons\ nt2\ v\ l\ env})\ r \\
\quad \textbf{where } \mathit{getNT\ nt}\ (\mathit{NTCons}\ \_\ l) = \mathit{getNT\ nt\ l}
\end{array}
$$

We will not go into further details here, but its implementation is similar to the *IsNotElem* 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.

Since the exportable non-terminals are wrapped into an *Export* value, we include an instance to lookup a non-terminal from its list of exportable non-terminals:

$$
\begin{array}{l}
\textbf{instance } \mathit{GetNT\ nt}\ (\mathit{nts\ env})\ r \\
\quad \Rightarrow \mathit{GetNT\ nt}\ (\mathit{Export\ start\ nts\ env})\ r \\
\quad \textbf{where } \mathit{getNT\ nt}\ (\mathit{Export}\ \_\ \mathit{nts}) = \mathit{getNT\ nt\ nts}
\end{array}
$$

To be able to finally export the starting point and the exportable non-terminals we chain an *Export* value through the transformation in order to return it as output.

$$
\begin{array}{l}
\mathit{exportNTs} :: \mathit{NTRecord}\ (\mathit{nts\ env}) \\
\qquad\qquad \Rightarrow \mathit{ExtGramTrafo\ env}\ (\mathit{Export\ start\ nts\ env})
\end{array}
$$

[5]We did it to keep the code as simple as possible, alternatives to avoid overlapping can be found in [6].

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad (\mathit{Export}\ \mathit{start}\ \mathit{nts}\ \mathit{env}) \\
\mathit{exportNTs} = \mathit{returnA}
\end{array}
$$

Thus, the definition of an extensible grammar, like the one in Figure 1, has the following shape [6]:

$$
\begin{array}{l}
\mathit{prds} = \mathit{proc}\ () \rightarrow \mathbf{do} \\
\quad ... \\
\quad \mathit{exportNTs} \prec \mathit{export}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{prds}' = \mathit{proc}\ (\mathit{imported}) \rightarrow \mathbf{do} \\
\quad ... \\
\quad \mathit{exportNTs} \prec \mathit{export}
\end{array}
$$

where *imported* and *export* are both of type *Export*. We have defined a function to extend (imported) exportable lists:

$$\mathit{extendExport}\ (\mathit{Export}\ r\ \mathit{nts})\ \mathit{ext} = \mathit{Export}\ r\ (\mathit{ext}\ \mathit{nts})$$

4.2.2 Adding Non-terminals

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

$$
\begin{array}{l}
\mathit{addNT} :: \mathit{ExtGramTrafo}\ \mathit{env}\ (\mathit{PreProductions}\ \mathit{env}\ a) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{Symbol}\ a\ \mathit{TNonT}\ \mathit{env}) \\
\mathit{addNT} = \mathit{proc}\ p \rightarrow \mathbf{do} \\
\qquad\qquad\qquad r \leftarrow \mathit{newSRef} \prec \mathit{prod}\ p \\
\qquad\qquad\qquad \mathit{returnA} \prec \mathit{Nont}\ r
\end{array}
$$

The input to *addNT* is the initial list of alternative productions (a *PreProductions*) 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:

$$
\begin{array}{l}
\mathit{term} \leftarrow \mathit{addNT}\ \prec\ [\![\ \mathit{semMul}\ \mathit{term}\ \texttt{"*"}\ \mathit{factor}\ ]\!] \\
\qquad\qquad\qquad\quad \texttt{<|>} [\![\ \mathit{id}\qquad\quad \mathit{factor}\ ]\!]
\end{array}
$$

we are adding the non-terminal *term* for terms, with the productions $[\![\ \mathit{semMul}\ \mathit{term}\ \texttt{"*"}\ \mathit{factor}\ ]\!]$ and $[\![\ \mathit{id}\ \mathit{factor}\ ]\!]$, and we bind to *term* a symbol holding the reference to the added non-terminal which can be used in the definition of this or other non-terminals. Because *Trafo* instantiates *ArrowLoop*, we can define mutually recursive non-terminals using the keyword **rec**.

4.2.3 Adding Productions

Adding new productions to an existing non-terminal translates into the concatenation of the new productions to the existing list of productions of the non-terminal.

$$
\begin{array}{l}
\mathit{addProds} :: \mathit{ExtGramTrafo}\ \mathit{env} \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{Symbol}\ a\ \mathit{TNonT}\ \mathit{env} \\
\qquad\qquad\qquad\qquad\qquad\quad , \mathit{PreProductions}\ \mathit{env}\ a) \\
\qquad\qquad\qquad\qquad\qquad\quad () \\
\mathit{addProds} = \mathit{proc}\ (\mathit{nont}, \mathit{prds}) \rightarrow \mathbf{do} \\
\quad \mathit{updateFinalEnv} \prec \\
\qquad \mathit{updateEnv}\ (\lambda \mathit{ps} \rightarrow \mathit{PS}\ \$\ (\mathit{unPP}\ \mathit{prds}) +\!\!+ (\mathit{unPS}\ \mathit{ps})) \\
\qquad\qquad\qquad (\mathit{getRefNT}\ \mathit{nont})
\end{array}
$$

In Figure 2 we have seen examples of adding productions to the non-terminals *exp* and *factor*.

[6]Using arrow's syntax [11]

4.2.4 Grammar Extension and Composition

To extend a grammar is to compose two transformations, the first one constructing an extensible grammar and the second one representing a grammar extension.

$$
\begin{aligned}
&(\texttt{+>>}) :: (\mathit{NTRecord}\ (\mathit{nts}\ \mathit{env}), \mathit{NTRecord}\ (\mathit{nts}'\ \mathit{env}))\\
&\qquad \Rightarrow \mathit{ExtGram}\ \mathit{env}\ \mathit{start}\ \mathit{nts}\\
&\qquad \rightarrow \mathit{GramExt}\ \mathit{env}\ \mathit{start}\ \mathit{nts}\ \mathit{start}'\ \mathit{nts}'\\
&\qquad \rightarrow \mathit{ExtGram}\ \mathit{env}\qquad\qquad \mathit{start}'\ \mathit{nts}'\\
&g\ \texttt{+>>}\ \mathit{sm} = g\ \texttt{>>>}\ \mathit{sm}
\end{aligned}
$$

We defined (+>>) to restrict the types of the composition. Two grammar extensions can be composed just by using (>>>).

If we want to compose two extensible grammars *g1* and *g2* (their non-terminals sets are disjoint), we have to sequence them, obtain their start points *s1* and *s2*, and finally add the new starting point; a non-terminal *s* that references to *s1* and *s2* as its alternatives.

$$
\begin{aligned}
&(\texttt{<++>}) :: (\mathit{NTUnion}\ \mathit{nts1}\ \mathit{nts2}\ \mathit{nts})\\
&\qquad \Rightarrow \mathit{ExtGram}\ \mathit{env}\ \mathit{start}\ \mathit{nts1}\\
&\qquad \rightarrow \mathit{ExtGram}\ \mathit{env}\ \mathit{start}\ \mathit{nts2}\\
&\qquad \rightarrow \mathit{ExtGram}\ \mathit{env}\ \mathit{start}\ \mathit{nts}\\
&\mathit{g1}\ \texttt{<++>}\ \mathit{g2} = \mathit{proc}\ () \rightarrow \mathbf{do}\\
&\qquad\qquad (\mathit{Export}\ \mathit{s1}\ \mathit{ns1}) \leftarrow \mathit{g1} \prec ()\\
&\qquad\qquad (\mathit{Export}\ \mathit{s2}\ \mathit{ns2}) \leftarrow \mathit{g2} \prec ()\\
&\qquad\qquad s \leftarrow \mathit{addNT} \prec \mathit{sym}\ \mathit{s1}\ \texttt{<|>}\ \mathit{sym}\ \mathit{s2}\\
&\qquad\qquad \mathit{returnA} \prec \mathit{Export}\ s\ (\mathit{ntUnion}\ \mathit{ns1}\ \mathit{ns2})
\end{aligned}
$$

5 Closed Grammars

To close a grammar we run the *Trafo*, in order to obtain the grammar to which we apply the Left-Corner Transform. By applying *leftcorner* we prevent the resulting grammar to be left-recursive, so it can be parsed by a top-down parser. Such a step is essential since we cannot expect from a large collection of language fragments, that the resulting grammar will be e.g. LALR(1) or non-left-recursive. The type of the start non-terminal *a* is the type of the resulting grammar.

$$
\begin{aligned}
&\mathit{closeGram} :: (\forall\ \mathit{env}.\mathit{ExtGram}\ \mathit{env}\ a\ \mathit{nts})\\
&\qquad\qquad \rightarrow \mathit{Grammar}\ \mathit{CG}\ a\\
&\mathit{closeGram}\ \mathit{prds} = \mathbf{case}\ \mathit{runTrafo}\ \mathit{prds}\ \mathit{Unit}\ ()\ \mathbf{of}\\
&\quad \mathit{Result}\ \_\ (\mathit{Export}\ (\mathit{Nont}\ r)\ \_)\ \mathit{gram}\\
&\qquad \rightarrow (\mathit{leftCorner}.\mathit{removeEmpties})\ (\mathit{Grammar}\ r\ \mathit{gram})
\end{aligned}
$$

The *leftcorner* function is an adaptation of our representation of *Prod* of the transformation proposed in [1]. The Left Corner transform does not accept grammars with either empty productions or productions which start with a possibly empty element, since such production do not have a well-defined collection of left-corner symbols, i.e., symbol which have be recognized first before the left-hand side symbol can be recognized. Thus, we need to introduce a preprocessing step which removes such empty productions.

$$
\begin{aligned}
&\mathit{removeEmpties} :: \mathit{Grammar}\ \mathit{EG}\ a \rightarrow \mathit{Grammar}\ \mathit{CG}\ a\\
&\mathit{leftCorner}\qquad :: \mathit{Grammar}\ \mathit{CG}\ a \rightarrow \mathit{Grammar}\ \mathit{CG}\ a
\end{aligned}
$$

The function *removeEmpties* takes a grammar that can have empty productions (*Grammar EG a*) and returns an equivalent grammar (*Grammar CG a*) without empty productions and without left-most empty elements. Since this transformation does not introduce new non-terminals, we do not need to use *TTTAS* to implement it.

First, the possibly empty production of each non-terminal is located using the function *findEmpties*, that takes the environment with the productions of the grammar and returns an isomorphic environment with values of type *HasEmpty*. If a non-terminal has an empty production, then the position of the resulting environment corresponding to this non-terminal contains a value *HasEmpty f*, where *f* is the semantic value associated to this empty case. If a non-terminal does not have empty productions then the environment contains a *HasNotEmpty* on its corresponding position.

After locating the empty productions we remove them from the grammar using the function *removeEmptiesEnv*, where the empty production of each non-terminal is removed and added to the contexts where the non-terminal is referenced. Thus, if the root symbol has an empty production, allowing the parsing of the empty string, this behavior will not be present after the removal. For simplicity reasons we avoid this situation by disallowing empty productions for the root symbol of the grammars we deal with, and yield an error message in this case. It is easy to remove this constraint by adding a production (*Pure f*) to the start non-terminal of the grammar resulting from the whole (*leftCorner.removeEmpties*) transformation, where (*HasEmpty f*) is the result of looking-up the start point in the environment of empty productions. But in practice we do not expect this to be necessary.

```
data HasEmpty a env = Unknown
                    | HasNotEmpty
                    | HasEmpty a

removeEmpties :: Grammar EG a → Grammar CG a
removeEmpties (Grammar start prds) =
  let empties = findEmpties prds
  in  case lookupEnv start empties of
        HasNotEmpty
          → Grammar start $
                        removeEmptiesEnv empties prds
        _ → error "Empty prod at start point"
```

In the following sub-sections we will explain the empty productions removal algorithm on more detail. For that we will use the following example grammar:

```
A → pure fA <*> tr "a" <*> sym B
B → sym C <*> sym D <*> pure fB
C → pure fC1 <*> sym C <*> tr "c" <|> pure fC2
D → pure fD <|> tr "d"
```

5.1 Finding Empty Productions

The function *findEmpties* constructs an environment with values of type *HasEmpty*. It starts with an initial environment of found empties without information, created by *initEmpties*, and iterates updating this environment until a fixpoint is reached. The function *stepFindEmpties* implements one step of this iteration, returning a triple with: the environment with the found empty productions thus far, a Boolean value indicating whether this step introduced changes to the environment, and a Boolean value that tells us whether the returned environment still has any *Unknown* non-terminals.

```
type GramEnv s = Env (Productions s)

findEmpties :: GramEnv EG env env
```

```
              → Env HasEmpty env env
findEmpties prods
  = findEmpties' prods (initEmpties prods)
findEmpties' prds empties =
  case stepFindEmpties empties prds empties of
  (empties', True,  _    ) → findEmpties' prds empties'
  (empties', False, False) → empties'
  (_,        False, True ) → error "Incorrect Grammar"
```

If we arrive at a fixpoint, and still have remaining *Unknown* non-terminals, then the grammar is incorrect, so we get a soundness check for the grammar for free. Such non-terminals will not be able to derive a finite sentence, as the following example shows:

```
A → term "a" <*> sym A
```

The initial environment of the algorithm is an environment with an *Unknown* value for each non-terminal of the grammar. In the example, the initial environment is the one of the `Step 0` in Figure 3.

```
initEmpties :: GramEnv EG use def
            → Env HasEmpty use def
initEmpties Empty         = Empty
initEmpties (Ext nts _)   = Ext (initEmpties nts)
                                Unknown
```

On each *step* we take the actual environment of found empty productions and we go through all the non-terminals of the grammar, trying to find out if the information about the existence of empty productions for this non-terminal can be updated (*updEmpty*).

```
stepFindEmpties :: Env HasEmpty use use
                → GramEnv EG use def
                → Env HasEmpty use def
                → (Env HasEmpty use def, Bool, Bool)
stepFindEmpties _         Empty          Empty
  = (Empty, False)
stepFindEmpties empties (Ext rprd prd) (Ext re e)
  = let (re', rchg, runk)
             = stepFindEmpties empties rprd re
        (e',  chg,  unk)
             = updEmpty empties prd e
    in  (Ext re' e', chg ∨ rchg, unk ∨ runk)
```

We only have to update the *HasEmpty* value associated with a non-terminal if in the actual environment it is *Unknown*. In the other cases we already know whether this non-terminal has any empty productions.

```
updEmpty :: Env HasEmpty use use
         → Productions EG a use
         → HasEmpty a use
         → (HasEmpty a use, Bool, Bool)
updEmpty empties prds Unknown
  = case hasEmpty empties prds of
         Unknown → (Unknown, False, True )
         e       → (e,       True,  False)
updEmpty _ _ e = (e, False, False)
```

Step 0	Step 1
$A \to$ *Unknown* $B \to$ *Unknown* $C \to$ *Unknown* $D \to$ *Unknown*	$A \to$ *HasNotEmpty* $B \to$ *Unknown* $C \to$ *HasEmpty fC2* $D \to$ *HasEmpty fD*
Step 2	**Step 3**
$A \to$ *HasNotEmpty* $B \to$ *HasEmpty* (*fC2 fD fB*) $C \to$ *HasEmpty fC2* $D \to$ *HasEmpty fD*	$A \to$ *HasNotEmpty* $B \to$ *HasEmpty* (*fC2 fD fB*) $C \to$ *HasEmpty fC2* $D \to$ *HasEmpty fD*

Figure 3: Results of Finding Empties Steps for the Example

The new *HasEmpty* information for a non-terminal is computed out of the previous environment and the list of productions of the non-terminal. The *HasEmpty* information is retreived for each production using *isEmpty*, and those results are combined. If any of the productions is empty then we have found that the non-terminal may derive the empty string. If we find more than one empty production the grammar is ambiguous. If all the productions are not empty, then we return *HasNotEmpty*. In other cases, the information for this non-terminal remains still *Unknown*.

```
hasEmpty :: Env HasEmpty env env
          → Productions EG a env → HasEmpty a env
hasEmpty empties (PS ps)
  = foldr (λp re → combine (isEmpty p empties) re)
          HasNotEmpty ps
combine   :: HasEmpty a env → HasEmpty a env
               → HasEmpty a env
combine (HasEmpty _) (HasEmpty _)
  = error "Ambiguous Grammar"
combine _              (HasEmpty f) = HasEmpty f
combine (HasEmpty f) _              = HasEmpty f
combine HasNotEmpty HasNotEmpty = HasNotEmpty
combine _              _              = Unknown
```

An empty production is obtained from: a production (*Pure a*), a reference to a non-terminal that has an empty production, or a sequence of two empty productions. In this case we construct a *HasEmpty a* value with a the semantic action associated to this empty alternative; for a sequential composition of actions f and x the associated semantic action is $(f\ x)$, given that *Pure f* ‘*Star*‘ *Pure x* $\overset{c}{\equiv}$ *Pure* $(f\ x)$. If a production is a terminal symbol, a reference to a non-terminal that has no empty production, or a sequence of two productions where at least one of them is not empty, then this production is not empty and we return *HasNotEmpty*. We obtain the information of the referenced non-terminals from the environment created thus far. Thus, it can happen that in a certain step there is not enough information to take a decision about a production, remaining it *Unknown*. This is the case of a reference to a non-terminal that is still *Unknown* and a sequence of two productions where the first production is empty and the second *Unknown* or the first is *Unknown* and the second not empty.

```
isEmpty :: Prod EG a env → Env HasEmpty env env
         → HasEmpty a env
isEmpty (Pure a)          _          = HasEmpty a
```

```
isEmpty (Sym (Nont r)) empties = lookupEnv r empties
isEmpty (Sym _)        _       = HasNotEmpty
isEmpty (Star pl pr)   empties
 = case isEmpty pl empties of
   HasNotEmpty → HasNotEmpty
   HasEmpty f  →
     case isEmpty pr empties of
       HasEmpty x → HasEmpty (f x)
       e          → e
   Unknown     →
     case isEmpty pr empties of
       HasNotEmpty → HasNotEmpty
       _           → Unknown
```

Let us look at our example grammar; in Figure 3 we show the results of the steps taken to find the empty productions.

In `Step 1` we find useful information for non-terminals A, C and D. In the case of A we have only one production:

Pure fA ‘*Star*‘ *Sym* (*Term* `"a"`) ‘*Star*‘ *Sym B*

looking at the left part of the sequence:

Pure fA ‘*Star*‘ *Sym* (*Term* `"a"`)

we have another sequence with an empty left part and a non-empty right part. Since one of its components is not empty, the whole sequence is not empty; and the same applies to its containing sequence.

In the cases of C and D, it can be seen that both have two productions. In both cases one production is empty and the other is not empty; hence we have immediately located an empty production for both non-terminals.

The non-terminal B has a single production:

Sym C ‘*Star*‘ *Sym D* ‘*Star*‘ *Pure fB*

if we look at the left part:

Sym C ‘*Star*‘ *Sym D*

it is a sequence of two non-terminals. Thus we have to look for the information we have from the previous step, in this case the `Step 0` (the initial environment). For both C and D we still do not have any information, thus the information of the left part of the production is *Unknown*. Since the right part of the production is empty (*Pure fB*), we cannot assume anything yet about the existence of empty productions for B.

In `Step 2` we take another look at B. Now we know that C has an empty production with semantic action *fC2* and D has an empty production with semantic action *fD*. Therefore from the left part of the sequence we can construct a *HasEmpty* (*fC2 fD*), having finally found the empty production *HasEmpty* (*fC2 fD fB*).

The `Step 3` does not perform any changes to the environment of found empty productions, since no non-terminal is *Unknown*. Thus, we have found the empty productions of the grammar.

5.2 Removal of Empty Productions

Once the empty productions are found, we can proceed to remove them. The function *removeEmptiesEnv* traverses the environment with the productions of the non-terminals, removing the empty productions, transforming productions which start with an empty element into productions starting with non-empty elements, and transforming the contexts where the non-terminals with empty productions are referenced.

$$\begin{aligned}
&\mathit{removeEmptiesEnv} :: \mathit{Env}\ \mathit{HasEmpty}\ \mathit{use}\ \mathit{use} \\
&\qquad\qquad\qquad\quad \to \mathit{GramEnv}\ \mathit{EG}\ \mathit{use}\ \mathit{def} \\
&\qquad\qquad\qquad\quad \to \mathit{GramEnv}\ \mathit{CG}\ \mathit{use}\ \mathit{def} \\
&\mathit{removeEmptiesEnv}\ _ \qquad \mathit{Empty} \\
&\quad = \mathit{Empty} \\
&\mathit{removeEmptiesEnv}\ \mathit{empties}\ (\mathit{Ext}\ \mathit{rprds}\ \mathit{prds}) \\
&\quad = \mathit{Ext}\ (\mathit{removeEmptiesEnv}\ \mathit{empties}\ \mathit{rprds}) \\
&\qquad\quad (\mathit{removeEmpty}\ \mathit{empties}\ \mathit{prds})
\end{aligned}$$

To remove the possibly empty production from a non-terminal, we apply the function *splitEmpty* to every production, concatenating the resulting alternative productions.

$$\begin{aligned}
&\mathit{removeEmpty} :: \mathit{Env}\ \mathit{HasEmpty}\ \mathit{env}\ \mathit{env} \\
&\qquad\qquad\quad \to \mathit{Productions}\ \mathit{EG}\ a\ \mathit{env} \\
&\qquad\qquad\quad \to \mathit{Productions}\ \mathit{CG}\ a\ \mathit{env} \\
&\mathit{removeEmpty}\ \mathit{empties}\ (\mathit{PS}\ \mathit{prds}) \\
&= \mathit{PS}\ \$\ \mathit{foldr}\ ((+\!\!+).\mathit{remEmptyProd})\ [\,]\ \mathit{prds} \\
&\quad \mathbf{where}\ \mathit{remEmptyProd}\ \mathit{prd} = \\
&\qquad \mathbf{let}\ (\mathit{prd}', _) = \mathit{splitEmpty}\ \mathit{empties}\ \mathit{prd} \\
&\qquad \mathbf{in}\ \ \mathit{prd}'
\end{aligned}$$

The function *splitEmpty* takes a production, and the environment of found empty productions, and returns a pair with a list of alternative productions generated from removing the empty part of the input production, and the possibly empty part of the production. While removing the empty productions in *removeEmpty* we use the generated non-empty productions and ignore the empty part.

$$\begin{aligned}
&\mathit{splitEmpty} :: \mathit{Env}\ \mathit{HasEmpty}\ \mathit{env}\ \mathit{env} \to \mathit{Prod}\ \mathit{EG}\ a\ \mathit{env} \\
&\qquad\qquad\quad \to ([\mathit{Prod}\ \mathit{CG}\ a\ \mathit{env}], \mathit{Maybe}\ (\mathit{Prod}\ \mathit{CG}\ a\ \mathit{env}))
\end{aligned}$$

In the case of non-terminal symbols, the generated non-empty production is a refernce to the symbol itself, since the algorithm will remove its possible empty production. The empty production, if it exists, is looked-up in the environment of found empty productions.

$$\begin{aligned}
&\mathit{splitEmpty}\ \mathit{empties}\ (\mathit{Sym}\ (\mathit{Nont}\ r)) \\
&\quad = \mathbf{case}\ \mathit{lookupEnv}\ r\ \mathit{empties}\ \mathbf{of} \\
&\qquad \mathit{HasEmpty}\ f \to ([\mathit{Sym}\ \$\ \mathit{Nont}\ r], \mathit{Just}\ (\mathit{Pure}\ f)) \\
&\qquad _ \qquad\qquad\ \to ([\mathit{Sym}\ \$\ \mathit{Nont}\ r], \mathit{Nothing})
\end{aligned}$$

Terminal symbols are non-empty productions, thus the generated non-empty production is the symbol itself, not having any empty part. On the other hand, a *Pure a* production is an empty production without non-empty part.

$$\begin{aligned}
&\mathit{splitEmpty}\ _\ (\mathit{Sym}\ s)\ \ = ([\mathit{Sym}\ s], \mathit{Nothing}) \\
&\mathit{splitEmpty}\ _\ (\mathit{Pure}\ a) = ([\,], \mathit{Just}\ (\mathit{Pure}\ a))
\end{aligned}$$

In the example, when removing the empty productions of D, *splitEmpty* is invoked for the alternative productions $(\mathit{Pure}\ \mathit{fD})$ and $(\mathit{Sym}\ (\mathit{Term}\ \texttt{"d"}))$, that result in the respective pairs $([\mathit{Sym}\ (\mathit{Term}\ \texttt{"d"})], \mathit{Nothing})$ and $([\,], \mathit{Just}\ (\mathit{Pure}\ \mathit{fD}))$. Thus, after the removal D only has the production $(\mathit{Sym}\ (\mathit{Term}\ \texttt{"d"}))$.
The non-empty productions generated by the transformation of a sequence f `<*>` g are:

- *fne_gne* Sequences of the combination of the non-empty productions generated from f and g.

- *fne_ge* Sequences of the non-empty productions generated from *f* and the empty production of *g*.
- *fe_gne* Sequences of the empty production of *f* and the non-empty productions generated from *g*. Here we introduce the FlipStar "reversed" sequence, translating (*fe* `<*>` *gne*) to (*gne* `<**>` *fe*), in order to move the non-empty part of the sequence to the left. Thus we avoid introducing left-most empty elements.

The possible empty production generated from *f* `<*>` *g* is *fe_ge*, a production *Pure* (*fv gv*) where *fv* and *gv* are the semantic actions associated to the empty productions of *f* and *g*, respectively. Notice the use of the *Maybe Monad*.

```
splitEmpty empties (Star f g)
  = let (fne, fe)  = splitEmpty empties f
        (gne, ge) = splitEmpty empties g
        fne_gne = [Star fv gv | fv ← fne, gv ← gne]
        fne_ge  = case ge of
                    Nothing → [ ]
                    Just gv → [Star fv gv | fv ← fne]
        fe_gne  = case fe of
                    Nothing → [ ]
                    Just fv → [FlipStar gv fv | gv ← gne]
        fe_ge   = do
                    (Pure fv) ← fe
                    (Pure gv) ← ge
                    return $ Pure (fv gv)
    in (fne_gne ++ fne_ge ++ fe_gne, fe_ge)
```

The function *splitEmpty* takes a production of type *Prod EG a env* as argument, and thus productions of the form (*FlipStar g f*) are not possible as input. However, as we have seen before, this kind of productions can be generated out of the transformation (case *fe_gne*) because the returned productions have type *Prod CG a env*. In the example, during the removal of the empty production of *B*, we call *splitEmpty* for:

```
Sym C `Star` Sym D `Star` Pure fB
```

that calls *splitEmpty* for *Sym C ‘Star‘ Sym D* and *Pure fB*. Let us see what happens in the evaluation for the first sub-production. The function *splitEmpty* is again called for *Sym C* and *Sym D*, resulting in:

```
fne ⇒ [Sym C]
fe  ⇒ Just (Pure fC2)
gne ⇒ [Sym D]
ge  ⇒ Just (Pure fD)
```

Thus, the empty part of the sub-production is (*Pure* (*fC2 fD*)), and the non-empty generated productions are:

```
[Sym C `Star`     Sym D
,Sym C `Star`     Pure fD
,Sym D `FlipStar` Pure fC2]
```

Finally, given that the result of *splitEmpty* for (*Pure fB*) is ([ ], *Just* (*Pure fB*)), the empty part of *B* coincides with the one found with *findEmpties* and the transformed *B* is:

```
B → PS [Sym C `Star`     Sym D    `Star` Pure fB
       ,Sym C `Star`     Pure fD  `Star` Pure fB
       ,Sym D `FlipStar` Pure fC2 `Star` Pure fB]
```

Note that now B: does not contain any empty production, includes productions for the empty and non-empty part of every referenced non-terminal, and has no left-most empty element.

The result of the transformation over the whole grammar example, using the smart constructors to make it easier to read, is:

```
A → tr "a" <**> pure fA  <*> sym B
  <|> tr "a" <**> pure fA  <*> pure (fC2 fD fB)
B → sym C <*> sym D    <*> pure fB
  <|> sym C <*> pure fD  <*> pure fB
  <|> sym D <**> pure fC2 <*> pure fB
C → sym C <**> pure fC1 <*> tr "c"
D → tr "d"
```

Notice how our brute-force approach generates grammars which have productions which start with the same sequence of elements. These will be taken care of by the left-factoring which is done as the last step of the Left Corner Transform. A slight different approach would be to extend our formalism to allow for nested structures, where we have a special kind of non-terminals, i.e. those which are only referenced once, and which we substitute directly in the grammar. This will lead to a grammar with a rule:

```
A → tr "a" <**> pure fA <*> (sym B
                             <|> pure (fC2 fD fB)
                             )
```

Unfortunately this will make the formulation of the Left Corner Transform more complicated.

6 Fixpoint Production

In order to be able to define recursive productions, we have added a sort of fixpoint combinator to our productions representation. The data type *Prod* is extended with the constructors *Fix*, for the fixpoint combinator, and *Var*, for references to the fixed point.

```
data FL a
data Prod s a env where
  ...
  Fix :: Productions (FL a) a env
       → Prod EG      a env
  Var :: Prod (FL a) a env
```

The type parameter s is used to restrict: *Fix* to be used only at "top-level", *Var* to be used only at "fixpoint-level", productions *Var* to have the same type of their containing *Fix*.

Thus, by defining some smart constructors:

```
varPrd :: PreProductions (FL a) env a
varPrd = PP [Var]
fixPrd :: PreProductions (FL a) env a
        → PreProductions EG     env a
fixPrd p = PP [(Fix.prod) p]
```

we can, for example, represent the useful EBNF-like combinators *pSome* and *pMany*.

```
pSome :: PreProductions (FL [a]) env a
       → PreProductions EG       env [a]
pSome p = fixPrd (one <|> more)
  where one  = (:[]) <$> p
```

```
              more = (:)   <$> p <*> varPrd

pMany :: PreProductions (FL [a]) env a
      → PreProductions EG        env [a]
pMany p = fixPrd (none <|> more)
  where none = pure []
        more = (:)   <$> p <*> varPrd
```

Another useful combinator is *pFoldr*, which is a generalized version of *pMany*, where the semantic functions have to be passed as an argument.

```
pFoldr :: (a → b → b, b)
       → PreProductions (FL b) env a
       → PreProductions TL     env b
pFoldr (c, e) p = fixPrd (none <|> more)
  where none = pure e
        more = c <$> p <*> varPrd
```

6.1 Fixpoint Removal

The semantics of *Fix* and *Var* are provided by a new transformation *removeFix*, that we add to the grammar closing pipeline.

```
closeGram :: (∀ env.ExtGram env a nts)
                → Grammar CG a
closeGram prds = case runTrafo prds Unit () of
  Result _ (Export (Nont r) _) gram
    →           (leftCorner.removeEmpties.removeFix)
                (Grammar r gram)
```

The function *removeFix* takes a grammar which can have *Fix* and *Var* productions, and returns a new grammar without them.

```
removeFix :: Grammar EG a → Grammar EG a
```

What we basically do is to traverse the input environment returning a copy of the productions in every case but *Fix*. When a (*Fix prds*) is found, we use *addNT* to add a new non-terminal to the grammar and return its reference. The productions we add to this non-terminal is the result of replacing *Var* by the non-terminal reference in *prds*. Thus, doing for example:

```
rec A ← addNT ≺ fixPrd
      (pure fA1 <|> pure fA2 <* trm "a" <*> varPrd)
```

is equivalent to do:

```
rec R ← addNT  ≺  pure fA1
                  <|> pure fA2 <* trm "a" <*> sym R
    A ← addNT  ≺  sym R
```

7 Conclusions and Future Work

We have shown how we can use typed abstract syntax to represent grammar fragments, how to analyze them and how to transform them using the TTTAS library. The algorithms we have shown are well-known as such, but have never been formulated in such a fully typed way. By formulating the transformations as we did we have given a partial correctness proof of the algorithms.

There are many places where, by making the representations a bit more involved, a more efficient algorithm can be had. Unfortunately the resulting descriptions will become quite a bit more involved too. It is part of our future work to find out whether such refinements will be needed in practical situations.

It should not go unnoticed that for our approach to work one should rely on an underlying combinator library which has a very general parsing strategy (as the `uu-parsinglib` has), and which is preferably able to handle ambiguous grammars. It is our experience that being able to handle such grammars, be it only to provide feedback to the language designer that the final grammar is ambiguous, is indispensable. In our case this can be easily achieved by lifting all semantic domains, by inserting *amb* combinators in the generated parsers, and lifting all semantic functions to Kleisli-compositions. Our next step will be to integrate a whole collection of techniques, of which this paper described only one, into the Utrecht Haskell Compiler (UHC); amongst these other techniques are the generation of first-class attribute grammar fragments by the *uuagc* compiler such that the first class language extensions can plug into them as described in [16], the efficient handling of larger collections of attributes as described in [8].

References

[1] 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, 2009. ACM.

[2] 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.

[3] 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, 2009.

[4] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

[5] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[6] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[7] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[8] Bruno Martínez, Marcos Viera, and Pardo Alberto. Just do it while compiling! fast extensible records in haskell. Technical report, Instituto de Computación, Universidad de la República, Montevideo, Uruguay, 2011.

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

[10] 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.

[11] 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.

[12] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[13] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[14] S. Doaitse Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, , and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Spinger, 2009.

[15] Marcos Viera, Doaitse Swierstra, S., and Eelco Lempsink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 63–74, New York, NY, USA, 2008. ACM.

[16] Marcos Viera, Doaitse Swierstra, S., and Arie Middelkoop. Uuag meets aspectag. Technical report, Instituto de Computación, Universidad de la República, Montevideo, Uruguay, 2011.

[17] 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.