Haskell, Do You Read Me?

Constructing and Composing Efficient Top-down Parsers at Runtime

Marcos Viera

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

Abstract

The Haskell definition and implementation of *read* is far from perfect. In the first place *read* is not able to handle the associativities defined for infix operators. Furthermore, it puts constraints on the way *show* is defined, and especially forces it to generate far more parentheses than expected. Lastly, it may give rise to exponential parsing times. All this is due to the compositionality requirement for *read* functions, which imposes a top-down parsing strategy.

We propose a different approach, based on typed abstract syntax, in which grammars describing the data types are composed dynamically. Using the transformation libraries described in a companion paper these syntax descriptions are combined and transformed into parsers at runtime, from which the required *read* function are constructed. In this way we obtain linear parsing times, achieve consistency with the defined associativities, and may use a version of *show* which generates far fewer parentheses, thus improving readability of printed values.

The described transformation algorithms can be incorporated

in a Haskell compiler, thus moving most of the work involved to compile time.

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 GADT, Haskell, Left-Corner Transform, Meta Programming, Parser Combinators, Type Systems, Typed Abstract Syntax, Typed Transformations

1. Introduction

In this paper we propose a solution to a few long standing, related problems in the design of the Haskell *Read* and *Show* classes. We start by explaining the current design, which was considered an optimal point in the design space available at the time of the design of Haskell98 (Peyton Jones 2003).

Consider the following data type, together with the fixity declarations of the operators involved:

Haskell'08, September 25, 2008, Victoria, BC, Canada.

S. Doaitse Swierstra Eelco Lempsink

Department of Computer Science
Utrecht University
Utrecht, The Netherlands
doaitse@cs.uu.nl, emlempsi@cs.uu.nl

```
infixl 5 :<:
infixr 6 :>:
data T1 = T1 :<: T1
        | T1 :>: T1
        | C1
        deriving (Read, Show)

v = C1 :>: C1 :<: C1 :<: C1
w = (read "C1 :>: C1 :<: C1 :<: C1" :: T1)
x = (read (show v)                  :: T1)
```

Given the fixity declarations, the definition of v is fine. Unfortunately the evaluation of w leads to a runtime error, because *read* is ignorant of the associativities of `:>:` and `:<:`. It is a sad observation that despite all the effort that went into the design of the language, we cannot just take a constant expression out of the program, put it in a file and read it back. Surprisingly, the definition of *show* is such that x is well-defined again.

The second problem relates to the efficiency of the standard implementation of *read*. In a GHC bug ticket (Petruzza et al.) it is explained why, with the current implementation of *read* and *show*, the following expression takes a long time to be processed, and on some systems may not run at all:

```
read "((((((((((C1))))))))))" :: T1
```

To understand what is going on we delve into the internals of the implementation, and the definitions of *read* and *show* from the Haskell98 Report, using a small example.

$$
\begin{array}{lllll}
T1\ (n) & \rightarrow & T1\ (5)\ \texttt{":<:"}\ T1\ (6) & & (n \leqslant 5) \\
 & | & T1\ (7)\ \texttt{":>:"}\ T1\ (6) & & (n \leqslant 6) \\
 & | & \texttt{"C1"} & & \\
 & | & \texttt{"("}\ T1\ (0)\ \texttt{")"} & &
\end{array}
$$

Figure 1. Grammar of the type *T1*

Consider the grammar of Figure 1, in which the parameter indicates the priority level at which the non-terminal may occur in an expression. Note how the associativity of the operators is encoded by this parameter: for the first alternative the second occurrence of *T1* in the right hand side has a higher priority.

A second observation is that for $n = 5$, this grammar is actually left recursive because of the first alternative, and thus cannot be parsed by conventional top-down parsing methods, based on recursive descent techniques.

The Haskell98 Report describes how left recursion is avoided by using a modified grammar, in which the priorities of the children are always higher than the priority of the left hand side of the production. So the language which is actually recognised by the generated *read* function is described by the non left-recursive grammar:

$$
\begin{aligned}
\mathit{T1}\ (n) \rightarrow\ & \mathit{T1}\ (6)\ \texttt{":<:"}\ \mathit{T1}\ (6) && (n \leqslant 5)\\
|\ & \mathit{T1}\ (7)\ \texttt{":>:"}\ \mathit{T1}\ (7) && (n \leqslant 6)\\
|\ & \texttt{"C1"}\\
|\ & \texttt{"("}\ \mathit{T1}\ (0)\ \texttt{")"}
\end{aligned}
$$

Note that this grammar treats all operators as non-associative. The derived instance for *read* is:

```
left_prec  = 5
right_prec = 6
app_prec   = 10

instance Read T1 where
  readsPrec n r
    = readParen (n > left_prec)
        (λr → [(u :<: v, w) |
           (u, s) ← readsPrec (left_prec + 1) r,
           (":<:", t) ← lex s,
           (v, w) ← readsPrec (left_prec + 1) t]
        ) r
    ++ readParen (n > right_prec)
        (λr → [(u :>: v, w) |
           (u, s) ← readsPrec (right_prec + 1) r,
           (":>:", t) ← lex s,
           (v, w) ← readsPrec (right_prec + 1) t]
        ) r
    ++ readParen (n > app_prec)
        (λr → [(C1, s) |
           ("C1", s) ← lex r]
        ) r
```

The function *readParen* requires a pair of parentheses around its parser argument, if its first argument evaluates to *True*. The price we have to pay for avoiding left-recursive grammars, is that we have to place many more parentheses in our expressions. The good news, and the reason that the aforementioned x is well-defined, is that the derived *show* function generates these extra parentheses; the derived *read* is helped to perform its task by the derived *show*, such that $read\ .\ show = id$.

By taking a closer look at this code we can now understand the source of inefficiency; all three alternatives happily start by accepting a `"("`-symbol – the first one expecting to see a `:<:` after having seen the corresponding closing parenthesis, the second one expecting a `:>:`, and the third one expecting nothing– and if the second input symbol is a `"("` too, all three have three more ways to proceed, leading to an exponential growth in parsing time.

Now consider a expression of the form `C1 :>: (C1 :>: (...))`. Here we do not have the problem of the opening parentheses, but for expressions with more than 10 `C1`s the parsing time grows exponentially too. What is happening? If we split the grammar according to the precedences we can see the problem:

$$
\begin{array}{lll}
T1\ (0..5) & \rightarrow & T1\ (6)\ \texttt{":<:"}\ T1\ (6) \mid T1\ (6) \\
T1\ (6) & \rightarrow & T1\ (7)\ \texttt{":>:"}\ T1\ (7) \mid T1\ (7) \\
T1\ (7..10) & \rightarrow & \texttt{"C1"} \\
 & \mid & \texttt{"("}\ T1\ (0)\ \texttt{")"}
\end{array}
$$

Due to the division of the non-terminal $T1$ into three non-terminals, new alternatives pointing directly to the next level have to be added to $T1\ (0..5)$ and $T1\ (6)$. Nonterminals $T1\ (0..5)$ and $T1\ (6)$ have a common prefix into their productions. So, each `"C1"` will be parsed twice before making a decision between the alternatives

$T1\ (7)\ \texttt{":>:"}\ T1\ (7)$ and $T1\ (7)$; and, even worse, this process is performed twice before deciding between $T1\ (6)\ \texttt{":<:"}\ T1\ (6)$

and T (6).

One might expect that there is a simple cure for these problems, since the Haskell compiler itself is able to parse the equivalent expression. In the example case a compiler could indeed spend a bit more time in analysing the data type and constructing an equivalent grammar which does not have the identified shortcomings. This leads, using the applicative parser interface (McBride and Paterson 2007), straightforwardly to the following combinator based parser for $T1$, using the parser combinators *pChainl* and *pChainr* which are defined in appendix A:

```
infixr 7 `pChainl`, `pChainr`
pT1 = (":<:", (:<:)) `pChainl`
      (":>:", (:<:)) `pChainr`
                     (pParens pT1 <|> pToken "C1")
```

Both combinators combine an operator, described by its string representation and a binary function defining its semantics, and a parser for the operands into a parser which recognises a sequence of operands separated by operators. When parsing is completed the combinator *pChainl* builds the result for a left-associative operator and *pChainr* for a right-associative operator.

Unfortunately however the situation is not always so easy to solve. Consider the following definition:

```
infix 5 :+:
infix 6 :*:

data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
```

When deriving *read* for $T2$, a Haskell implementation does not generate a parser, but a function that maps a parser (coming from the *Read* dictionary) recognising values of some parameter type a, to a parser which recognises values of type $T2\ a$. In this way we deal with the situation that the complete grammar is not always at hand when building parsers: the parameter of $T2$ might be defined in another module, or may not be defined at all.

It now also becomes clear why the strategy chosen in Haskell works; we have limited our languages to a class for which we can build parsers by composing parsers whenever we define new languages by composing languages. Each module happily generates its own instances of the class *Read*, and these values can straightforwardly be combined into the required parser. So the question we answer in this paper is:

> *"How can we construct efficient parsers for the language of data types in a compositional way?"*.

In the rest of this paper we show how these problems can be overcome, using a library for transforming typed abstract syntax, the design of which has been described in an accompanying paper (Baars and Swierstra 2008).

Before delving into the technical details we start out by sketching the solution. Parser generators usually perform some form of grammar analysis, and unfortunately the result of such analyses cannot easily be combined into the analysis result for a combined grammar (Bravenboer 2008; Bouwers et al. 2008). Since there is no easy way to compose parsers, we take one step back and compose grammars instead, and thus we have to represent grammars as Haskell values. Once the grammars are composed we can build the required parser.

In order to make grammars first-class values we introduce a polymorphic data type $DGrammar\ a$ (DataGrammar), describing grammatical structures which in their turn describe *String* values corresponding to values of type a. By making values of this data type member of a class:

```
class Gram a where
  grammar :: DGrammar a
```

we can now provide the type of our *read* function, *gread*:

```
read  :: Read a ⇒ String → a    -- the original
gread :: Gram a ⇒ String → a    -- this paper
```

In Section 2 we give a top-level overview of the steps involved. In Section 3 we describe how to represent grammars by typed abstract syntax, thus preparing them for the transformations in Section 4. In Section 5 we spend some words on the efficiency of the overall approach and describe a few open problems and details to pursue further, whereas in Section 6 we conclude.

2. A Better Read

We obtain a parser for rules of data type t by taking the following steps.

deriveGrammar Generate an instance of the class *Gram*. We provide a function *deriveGrammar*, defined using Template Haskell (Sheard and Peyton Jones 2002), which performs this step, although we would eventually expect a compiler to take care of this. The instance *Gram T1*, describing the structure of the type *T1* is generated by calling:

```
$ (deriveGrammar "T1)
```

In this generated description precedences and associativities are reflected by annotating uses of non-terminals in the right hand side with the precedence of the position at which they occur, and by annotating productions with the level at which they may be applied (as in Figure 1). This is similar to the description given in the Haskell98 report.

group When a grammar refers to other grammars, which are generated separately and probably in a different module, we have to remove these references by combining the separate grammars into a single complete grammar; this corresponds to the dictionary passing for *Read*. Once this is done we know all the precedences of all the non-terminals involved, and we may construct a new grammar using a sufficient number of new non-annotated non-terminals, in which the precedences and associativities are represented by the grammar itself.

leftcorner For all resulting left-recursive grammar (or parts thereof) we perform the Left-Corner transform (Baars and Swierstra 2008). The LC-transform is a relatively straightforward transformation which maps a grammar onto an equivalent grammar which is not left-recursive.

leftfactoring Apply left-factoring to the resulting grammar, in order to remove the source of inefficiencies we have seen in section 1.

compile Convert the grammar into a parser. We use the parser combinators included in the UU library (Swierstra 2008) package in order to construct a fast parser. The function *compile* is defined in appendix B.

parse Add a call to this parser, a check for a successful result and the generation of an error message in case of failure.

All these steps are visible as individual functions in *gread*:

$$
\begin{aligned}
&gread :: (Gram\ a) \Rightarrow String \rightarrow a\\
&gread = (\ parse\ .\ compile\\
&\qquad\qquad .\ leftfactoring\ .\ leftcorner\\
&\qquad\qquad .\ group)\ grammar
\end{aligned}
$$

```
instance Gram T1 where
  grammar = DGrammar _0 envT1
envT1 :: Env DGram (T1, ()) (T1, ())
envT1 = consD (nonts _0) Empty
  where
    nonts _T1 = DLNontDefs
      [(DRef (_T1, 5)
       , DPS [dNont (_T1, 5) .#. dTerm ":<:" .#.
                dNont (_T1, 6) .#. dEnd infixL]
       )
      , (DRef (_T1, 6)
       , DPS [dNont (_T1, 7) .#. dTerm ":>:" .#.
                dNont (_T1, 6) .#. dEnd infixR]
       )
      , (DRef (_T1, 10)
       , DPS [dTerm "C1" .#. dEnd (const C1)
              , dTerm "(" .#. dNont (_T1, 0) .#.
                dTerm ")" .#. dEnd parenT]
       )
      ]
    infixL e1 _ e2 = e2 :<: e1
    infixR e1 _ e2 = e2 :>: e1
```

Figure 2. Representation of the grammar of type *T1*

Since all these steps, except the first one, are performed at runtime, we have achieved true runtime compositionality. Modules can be compiled separately, and the final parsing function is generated just in time. In the next subsections we look at each step in more detail.

2.1 Deriving *Gram*

The data type *DGrammar* describes grammars, and we postpone its detailed discussion to Section 3. In Figure 2 we give the instance of the class *Gram*, containing a value of type *DGrammar T1*, which is generated for the data type *T1* from Figure 1. Without going into the implementation details, it is easy to see the direct relationship between the data type *T1* and its *DGrammar T1* representation. For example the part of the grammar:

$$| \; T1 \; (7) \; \texttt{":>:"} \; T1 \; (6) \qquad\qquad (n \leqslant 6)$$

which corresponds to the second alternative ($n \leqslant 6$) in the data type definition, is represented by the pair:

$$\begin{array}{l} (DRef \; (\_T1, 6) \\ , DPS \; [dNont \; (\_T1, 7) \; .\#. \; dTerm \; \texttt{":>:"} \; .\#. \\ \qquad\quad dNont \; (\_T1, 6) \; .\#. \; dEnd \; infixR] \\ ) \end{array}$$

In the first component of this pair we specify the non-terminal and its precedence level (which corresponds to a guard behind a set of production rules), while in the second component we find the set of corresponding productions (in this case a singleton list). Each right-hand side consists of a sequence of terminals ($dTerm$) and non-terminals ($dNont$), separated by an operator .#. indicating sequential composition. The sequence finishes with a call to $dEnd\ f$, where f (in this case $infixR$) is a function which takes the parsing results of the right-hand side elements into a value of type $T1$.

$$
\begin{array}{lllll}
T2\ (n) & \rightarrow & T2\ (6)\ \texttt{":+:"}\ T2\ (6) & & (n \leqslant 5) \\
 & \mid & A\ (7)\ \ \texttt{":*:"}\ T2\ (7) & & (n \leqslant 6) \\
 & \mid & \texttt{"C2"} & & \\
 & \mid & \texttt{"("}\ T2\ (0)\ \texttt{")"} & &
\end{array}
$$

Figure 3. Grammar of the type $T2\ a$

2.2 Grouping

The first transformation we apply to the grammar is to split it according to precedences actually used. The result of grouping the grammar for the type $T1$ (Figure 1) is:

$$
\begin{array}{l}
A \rightarrow A\ \texttt{":<:"}\ B \mid B \\
B \rightarrow C\ \texttt{":>:"}\ B \mid C \\
C \rightarrow \texttt{"C1"} \mid \texttt{"("}\ A\ \texttt{")"}
\end{array}
$$

where A groups all non-terminals from level 0 to 5, B corresponds to the non-terminal of level 6 and C all non-terminals from level 7 up-to 10. The original reference to $T1\ (0)$ between parentheses is mapped to a reference to A. For non-terminals representing levels less than 10 (A and B) a new alternative that points to the next level is added.

When a grammar contains references to non-terminals of other grammars, we include all the referred grammars. Hence, if we have the grammar of $T2\ a$ (Figure 3), the result of grouping $T2\ T1$ is:

$$
\begin{array}{l}
A \rightarrow B\ \texttt{":+:"}\ B \mid B \\
B \rightarrow F\ \texttt{":*:"}\ C \mid C \\
C \rightarrow \texttt{"C2"} \mid \texttt{"("}\ A\ \texttt{")"} \\
D \rightarrow D\ \texttt{":<:"}\ E \mid E \\
E \rightarrow F\ \texttt{":>:"}\ E \mid F \\
F \rightarrow \texttt{"C1"} \mid \texttt{"("}\ D\ \texttt{")"}
\end{array}
$$

Note that the non-terminal names of the split grammar of $T1$ have changed from A, B and C to D, E and F, respectively.

Of course a compiler could do this statically for those types for which all necessary information is already available; but in the general case this is something which has to be done dynamically.

2.3 LC-Transformation

Consider the grammar of the data type $T1$ after applying *group*. The production:

$$A \rightarrow A\ \texttt{":<:"}\ B \mid B$$

is left-recursive. So, this grammar cannot be parsed by a top-down parser. We remedy this by applying a Left-Corner transformation (Johnson 1998), for which a typed implementation is given in (Baars and Swierstra 2008). Since the complete implementation is given in this companion paper, we only give a short description of this transformation, in order to make this paper self-contained.

We use the following notational convention for grammar meta-variables. Lower-case letters (a, b, etc.) denote terminal symbols. Low-order upper-case letters (A, B, etc.) denote non-terminals, while high-order upper-case letters (X, Y, Z) denote symbols that can either be terminals or non-terminals. Greek lower-case symbols (α, β, etc.) denote sequences of terminals and non-terminals.

A *direct left-corner* of a non-terminal A is a symbol X so that there exists a production for A with X as the left-most symbol on the right-hand side. The *left-corner* relation is defined as the transitive closure of the direct left-corner relation. So, a non-terminal being left-recursive is equivalent to being a left-corner of itself.

For each (left-recursive) non-terminal A of the original grammar, the function *leftcorner* applies the following rules to build new productions for A and productions for new non-terminals

A_X, where X is a left-corner of A and a non-terminal A_X stands for that part of an A after having seen an X.

1. For each production $A \rightarrow X \;\; \alpha$ of the source grammar add $A\_X \rightarrow \alpha$ to the target grammar, and add X to the set of left-corners found for A.
2. For each newly found left-corner X of A:
   (a) If X is a terminal symbol b add $A \rightarrow b \;\; A\_b$ to the transformed grammar.
   (b) If X is a non-terminal B then for each original production $B \rightarrow Y \;\; \beta$ add the production $A\_Y \rightarrow \beta \;\; A\_B$ to the transformed grammar and add Y to the left-corners of A.

The left-corner transformation for the type *T1* yields the grammar:

$$
\begin{array}{lll}
A & \rightarrow & \texttt{"C1"}\ A\_C1 \mid \texttt{"("}\ A\_(\\
A\_A & \rightarrow & \texttt{":<:"}\ B\ A\_A \mid \texttt{":<:"}\ B \\
A\_B & \rightarrow & A\_A \mid \epsilon \\
A\_C & \rightarrow & \texttt{":>:"}\ B\ A\_B \mid A\_B \\
A\_C1 & \rightarrow & A\_C \\
A\_(& \rightarrow & A\ \texttt{")"}\ A\_C \\
B & \rightarrow & \texttt{"C1"}\ B\_C1 \mid \texttt{"("}\ B\_(\\
B\_C & \rightarrow & \texttt{":>:"}\ B \mid \epsilon \\
B\_C1 & \rightarrow & B\_C
\end{array}
$$

$$
\begin{array}{ll}
B\_(& \rightarrow A \texttt{")"}\ B\_C \\
C & \rightarrow \texttt{"C1"}\ C\_C1 \mid \texttt{"("}\ C\_(\\
C\_C1 & \rightarrow \epsilon \\
C\_(& \rightarrow A \texttt{")"}
\end{array}
$$

2.4 Left-Factoring

Looking at the grammar of *T1* after the LC-transform, we see that a common prefix has appeared in the productions for the non-terminal A_A. This overlap leads to inefficient parsers, since we have to parse the same part of the input more than once. The function *leftfactoring* removes such common prefixes by applying the following rule until all left-factors have been removed.

- For each set of productions $C = \{A \rightarrow X\ \alpha_1, ..., A \rightarrow X\ \alpha_n\}$, with $n > 1$, add the productions ($A \rightarrow X\ A\_\_X$, $A\_\_X \rightarrow \alpha_1, ..., A\_\_X \rightarrow \alpha_n$) to the grammar, and remove the productions in C.

So, by applying *leftfactoring* to the grammar after the LC-transform we obtain its optimised version:

$$
\begin{array}{ll}
A & \rightarrow \texttt{"C1"}\ A\_C1 \mid \texttt{"("}\ A\_(\\
A\_A & \rightarrow \texttt{":<:"}\ A\_A\_\_lt \\
A\_A\_\_lt & \rightarrow B\ A\_A\_\_lt\_\_B \\
A\_A\_\_lt\_\_B & \rightarrow A\_A \mid \epsilon \\
A\_B & \rightarrow A\_A \mid \epsilon \\
A\_C & \rightarrow \texttt{":>:"}\ B\ A\_B \mid A\_B \\
A\_C1 & \rightarrow A\_C \\
A\_(& \rightarrow A \texttt{")"}\ A\_C \\
B & \rightarrow \texttt{"C1"}\ B\_C1 \mid \texttt{"("}\ B\_(\\
B\_C & \rightarrow \texttt{":>:"}\ B \mid \epsilon \\
B\_C1 & \rightarrow B\_C
\end{array}
$$

$$
\begin{array}{ll}
B\_(& \rightarrow A \texttt{")"}\ B\_C \\
C & \rightarrow \texttt{"C1"}\ C\_C1 \mid \texttt{"("}\ C\_(\\
C\_C1 & \rightarrow \epsilon \\
C\_(& \rightarrow A \texttt{")"}
\end{array}
$$

3. Representing Data Type Grammars

We represent the grammars as *typed abstract syntax*, encoded using Generalised Algebraic Data Types (Peyton Jones et al. 2006). In the following subsections we introduce this representation and the issues involved in deriving it from a data type. The main problem to be solved is how to represent the typed references, and how to maintain a type correct representation during the transformation processes.

3.1 Typed References and Environments

Pasalic and Linger (Pasalic and Linger 2004) introduced an encoding *Ref* of typed references to an environment containing values of different type. A *Ref* is labeled with the type of the referenced value and the type of an environment (a nested Cartesian product) the value lives in:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ref} :: * \rightarrow * \rightarrow * \textbf{where} \\
\quad \mathit{Zero} :: \mathit{Ref}\ a\ (a, env') \\
\quad \mathit{Suc}\ \ :: \mathit{Ref}\ a\ env' \rightarrow \mathit{Ref}\ a\ (x, env')
\end{array}
$$

The constructor *Zero* expresses that the first element of the environment has to be of type a. The constructor *Suc* does not care about the type of the first element in the environment (it is polymorphic in the type x), and remembers a position in the rest of the environment.

Baars and Swierstra (Baars and Swierstra 2004, 2008) extend this idea such that environments do not contain values of mixed type but terms (expressions) describing such values instead; these terms take an extra type parameter describing the environment to which references to other terms occurring in the term may point. In this way we can describe typed terms containing typed references to other terms. As a consequence, an *Env* may be used to represent an environment, consisting of a collection of possibly mutually recursive definitions (in our case grammars). The environment stores a heterogeneous list of terms of type $t\ a\ use$, which are the right-hand expressions of the definitions. References to elements are represented by indices in the list.

$$
\begin{aligned}
&\textbf{data}\ Env :: (* \to * \to *) \to * \to * \to * \textbf{where} \\
&\quad Empty :: Env\ t\ use\ () \\
&\quad Cons\ \ :: t\ a\ use \to Env\ t\ use\ \ \ \ def' \\
&\qquad\qquad\qquad\quad \to Env\ t\ use\ (a, def')
\end{aligned}
$$

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 in environment using *Cons*, its type label is included as the first component of *def*. The type *use* describes the types that may be referenced from within terms of type $t\ a\ use$ using $Ref\ a\ use$ values. When the types *def* and *use* coincide the type system ensures that the references in the terms do not point to values outside the environment.

The function *lookupEnv* takes a reference and an environment. The reference is used as an index in the environment to locate the referenced value. The types guarantee that the lookup succeeds, and that the value found is indeed labeled with the type with which the *Ref* argument was labeled:

$$
\begin{aligned}
&lookupEnv :: Ref\ a\ env \to Env\ t\ s\ env \to t\ a\ s \\
&lookupEnv\ Zero\ \ \ \ \ \ (Cons\ p\ \_)\ \ = p \\
&lookupEnv\ (Suc\ r)\ (Cons\ \_\ ps) = lookupEnv\ r\ ps
\end{aligned}
$$

3.2 Typed Grammar Representations

Baars and Swierstra introduce a data type *Grammar* for representing grammatical structures. A *Grammar* consists of a root symbol, represented by a value of type *Ref a...*, where *a* is the type of the witness of a successful parse, and an environment *Env*, containing

for each non-terminal of the grammar its list of alternative productions. As we require grammars to be closed, we pass the parameter *env* both at the *use* and the *def* position and because the internal structure of the grammar is not of interest it is made into an existential. This enables us to add or remove non-terminals without changing the type of the grammar as such.

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

A production is a sequence of symbols, and a symbol is either a terminal with *Token* as its witness or a non-terminal, encoded by a reference.

```
data Token = Keyw String
           | Open
           | Close
data Symbol a env where
  Nont :: Ref a env → Symbol a     env
  Term :: Token     → Symbol Token env
data Prod a env where
  Seq  :: Symbol b env → Prod (b → a) env
```

$$\rightarrow Prod\ a \qquad env$$
$$End :: a \qquad \rightarrow Prod\ a \qquad env$$

The right hand side sequence of symbols terminated by an *End f* element. The function *f* accepts the parsing results of the right hand side elements as arguments, and builds the parsing result for the left-hand side non-terminal.

3.3 Typed Grammar Representations for Data Types

For a grammar corresponding to a Haskell data type the situation is a bit different, since we actually have a whole collection of non-terminals: for each non-terminal the set is indexed by the precedences. Furthermore in productions of a non-terminal we can have references to non-terminals of both the grammar (i.e. data type) being defined as well as other grammars, corresponding to parameters of the data type. For example, the grammar of the type $T2\ a$ (Figure 3) has a reference to the 7th precedence level of the grammar of the type parameter a.

We coin the non-terminal we are finally interested in the *main non-terminal*, and our new grammar representation type *DGrammar* starts with a reference to the main non-terminal in the environment. Note that this is the only non-terminal that can be referred to from outside the grammar!

```
data DGrammar a
   = ∀ env . DGrammar (Ref a env)
                      (Env DGram env env)
data DGram a env = DGD (DLNontDefs a env)
                 | DGG (DGrammar a)
```

Other non-terminals definitions may be included in the environment as further *DGD*'s, and all the non-terminals labeled by *DGD* can be mutually recursive. In order to be able to refer to other grammars (such as introduced by a type parameter) we introduce an extra kind of non-terminal (DGG), which is the starting symbol of a completely new grammar. This imposes a tree like hierarchy on our non-terminals, with the $DGrammar$ nodes representing mutually recursive sets of non-terminals.

A reference to a non-terminal has to indicate the place in the environment where the non-terminal is defined (which can either be an internal non-terminal or another grammar) and the level of precedence at the referring position:

$$\mathbf{newtype}\ DRef\ a\ env = DRef\ (Ref\ a\ env, Int)$$

A non-terminal is defined by a list of productions available at each precedence level. An occurrence $(DRef\ (r, n), prods)$ tells us that the alternatives $prods$ of the non-terminal r are available for the levels from 0 to n. For efficiency reasons we order the list in increasing order of precedence.

```
newtype DLNontDefs a env
   = DLNontDefs [(DRef a env, DProductions a env)]
```

The list of alternative productions $DProductions$ is defined similar to $Productions$.

```
newtype DProductions a env
   = DPS{unDPS :: [DProd a env]}
data DProd a env where
  DSeq :: DSymbol b env → DProd (b → a) env
                        → DProd a env
  DEnd :: a             → DProd a env
data DSymbol a env where
  DNont :: DRef a env → DSymbol a env
  DTerm :: Token → DSymbol Token env
```

In order to make our grammar definitions look a bit nicer we introduce:

$$
\begin{array}{l}
\textbf{infixr}\ 5\ .\#. \\
\begin{array}{lll}
(.\#.) & & = DSeq \\
consG\ \ g\ es & & = Cons\ \ (DGG\ g)\ es \\
consD\ \ g\ es & & = Cons\ \ (DGD\ g)\ es \\
dNont\ \ nt & & = DNont\ (DRef\ nt) \\
dTerm\ \ t & \mid t \equiv \texttt{"("} & = DTerm\ Open \\
 & \mid t \equiv \texttt{")"} & = DTerm\ Close \\
 & \mid otherwise & = DTerm\ (Keyw\ t) \\
dEnd\ \ f & & = DEnd\ \ f \\
parenT\ p1\ e\ p2 & & = e
\end{array} \\
\_0 = Zero \\
\_1 = Suc\ \_0 \\
\_2 = Suc\ \_1
\end{array}
$$

Figure 4 shows the *DGrammar* $(T2\ a)$ representation of the grammar $T2\ a$ (Figure 3). It consists of an environment with the production of $T2\ a$ represented at position $\_0$ and the grammar of the type a at position $\_1$. So $DRef\ (\_0, n)$ refers to $T2\ a$ at level n and $DRef\ (\_1, n)$ refers the grammar of the type a at level n. Due to the type signature of the environment, the type system guarantees that the *grammar* we store as the second component in the environment is of type $DGrammar\ a$.

3.4 Representing Mutually Recursive Data Types

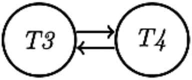
When performing the grammar transformations, we expect the grammars to be complete, i.e. all referred grammars are inlined in the grammar from which we want to derive a *gread*. In case of mutually recursive data types, like $T3$ and $T4$ of Figure 5, if we derive the instances:

```
instance Gram T3 where
  grammar = DGrammar _0 envT3
instance Gram T4 where
  grammar = DGrammar _1 envT4
```

```
instance Gram a ⇒ Gram (T2 a) where
  grammar = DGrammar _0 envT2
envT2 :: (Gram a) ⇒ Env DGram (T2 a, (a, ()))
                                    (T2 a, (a, ()))
envT2 = consD (nonts _0 _1) $
          consG grammar Empty
  where
    nonts _T2 _A = DLNontDefs
      [(DRef (_T2, 5)
       , DPS [dNont (_T2, 6) .#. dTerm ":+:" .#.
               dNont (_T2, 6) .#. dEnd infixP]
       )
      , (DRef (_T2, 6)
       , DPS [dNont (_A,  7) .#. dTerm ":*:" .#.
               dNont (_T2, 7) .#. dEnd infixT]
       )
      , (DRef (_T2, 10)
       , DPS [dTerm "C2"  .#. dEnd (const C2)
             , dTerm "("   .#. dNont (_T2, 0) .#.
               dTerm ")"   .#. dEnd parenT]
       )
      ]
    infixP e1 _ e2 = e2 :+: e1
    infixT e1 _ e2 = e2 :*: e1
```

Figure 4. Representation of the grammar of type *T2 a*

```
data T3 = T3 T4 | C3
data T4 = T4 T3 | C4
```




```
data T5 = T5 T6 | C5
data T6 = T6 T7
data T7 = T7_ T5
```

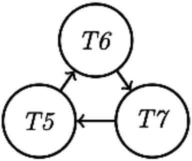


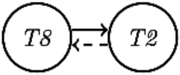
Figure 5. Mutually recursive types with graph representation

we get an unbounded number of copies of each grammar when trying to inline them. This happens because the generation of the grammars is mutually recursive too.

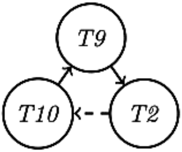
Mutual recursion occurs if there is a cycle of data types mentioned explicitly. When trying to define the representation of a type it can be detected, by constructing a directed graph with the explicit calls to other types. If the type belongs to a strongly connected component there is a cyclic type dependency with the other components.

We have solved the problem of cyclic dependencies using the idea of binding groups (Peyton Jones 2003). When a strongly connected component is found, the definitions of all the components types are tupled together into a single environment. Remember that our environments (*Env*) have no problem in describing mutually recursive definitions. So, in the case of *T3* and *T4*, we build the environment:

```
data T8 = T8 (T2 T8)
        | C8
```




```
data T9  = T9 (T2 T10)
         | C9
data T10 = T10 T9
```




```
data T11 = T11 (T2 T11)
               (T2 T12)
data T12 = T12 T11
```

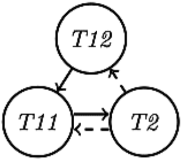


Figure 6. Mutually recursive components with *weak* edges

```
envT3T4 :: Env DGram (T3, (T4, ())) (T3, (T4, ()))
envT3T4 = consD (nonts3 _0 _1) $
            consD (nonts4 _1 _0) Empty
  where
    nonts3 _T3 _T4 = DLNontDefs
```

```
    [(DRef (_T3, 10)
     , DPS [dTerm "T3" .#. dNont (_T4, 0) .#.
             dEnd consT3
           , dTerm "C3" .#. dEnd (const C3)
           , dTerm "(" .#. dNont (_T3, 0) .#.
             dTerm ")" .#. dEnd parenT
           ]
     )
    ]
nonts4 _T4 _T3 = DLNontDefs
    [(DRef (_T4, 10)
     , DPS [dTerm "T4" .#. dNont (_T3, 0) .#.
             dEnd consT4
           , dTerm "C4" .#. dEnd (const C4)
           , dTerm "(" .#. dNont (_T4, 0) .#.
             dTerm ")" .#. dEnd parenT
           ]
     )
    ]
consT3 a = const (T3 a)
consT4 a = const (T4 a)
```

Note that when defining *T3* we pass the location of *T4* in the environment, and vice versa. For both types the instances can now be created using the same environment, only using different references for the root symbols.

```
instance Gram T3 where
  grammar = DGrammar _0 envT3T4
instance Gram T4 where
  grammar = DGrammar _1 envT3T4
```

As we can see in Figure 6, there are some cases where a type is a member of a strongly connected component, but it does not contain explicit references to the other members of its component. This happens when we have a parametrised type that is instantiated with a member of the component. This relation is expressed in the figure

as a dashed edge in the graph. We call such edges *weak* edges, and the types pointing from a such an edge a *weak member*.

These types, in the examples *T2*, generate the cyclic type dependencies but they do not form part of it: the grammar for *T2* is generated without referring to *T8*, *T9*, *T10* or *T11*. But, for example, to generate the grammar of *T9* (or *T10*) the definition of (*T2 T10*) has to be made part of the environment. So in order to define the environment for the instances of *T9* and *T10*:

```
instance Gram T9 where
  grammar = DGrammar _0 envT9T10
instance Gram T10 where
  grammar = DGrammar _1 envT9T10
```

We include a copy of the definition of the non-terminals of *T2 a* instantiated with *T10*:

```
envT9T10 :: Env DGram (T9, (T10, (T2 T10, ())))
                      (T9, (T10, (T2 T10, ())))
envT9T10 = consD (nonts9  _0 _2) $
           consD (nonts10 _1 _0) $
           consD (nonts2  _2 _1) Empty
  where
    nonts9 _T9 _T2 = DLNontDefs
      [(DRef (_T9, 10)
       , DPS  [dTerm "T9" .#. dNont (_T2, 0) .#.
```

```
                dEnd consT9
               , dTerm "C9" .#. dEnd (const C9)
               , dTerm "(" .#. dNont (_T9, 0) .#.
                 dTerm ")" .#. dEnd parenT
               ]
    )
  ]
nonts10 _T10 _T9 = DLNontDefs
  [(DRef (_T10, 10)
   , DPS  [dTerm "T10" .#. dNont (_T9, 0) .#.
           dEnd consT10
          , dTerm "(" .#. dNont (_T10, 0) .#.
            dTerm ")" .#. dEnd parenT
          ]
   )
  ]
nonts2 _T2 _T10 = DLNontDefs
  [(DRef (_T2, 5)
   , DPS  [dNont (_T2, 6)  .#. dTerm ":+:" .#.
           dNont (_T2, 6)  .#. dEnd infixP]
   )
  , (DRef (_T2, 6)
   , DPS  [dNont (_T10, 7) .#. dTerm ":*:" .#.
           dNont (_T2, 7)  .#. dEnd infixT]
   )
  , (DRef (_T2, 10)
   , DPS  [dTerm "C2"  .#. dEnd (const C2)
          , dTerm "("  .#. dNont (_T2, 0) .#.
            dTerm ")"  .#. dEnd parenT]
   )
```

```
]
consT9  a = const (T9  a)
consT10 a = const (T10 a)
infixP e1 _ e2 = e2 :+: e1
infixT e1 _ e2 = e2 :*: e1
```

Note that the instance of *Gram T2* does not occur in this environment; the instance of *Gram T2* is the one defined in Section 3.3.

We have to include all the instances of *weak* edges into a binding group. In the case of *T11* there are two *weak* edges from *T2*. Hence both (*T2 T11*) and (*T2 T12*) are included.

```
envT11T12 :: Env DGram
  (T11, (T12, (T2 T11, (T2 T12, ()))))
  (T11, (T12, (T2 T11, (T2 T12, ()))))
envT11T12 = consD (nonts11 _0 _2 _3) $
            consD (nonts12 _1 _0) $
            consD (nonts2  _2 _0) $
            consD (nonts2  _3 _1) Empty
```

3.5 Non Representable Data Types

There are some cases in which we cannot define a representation of the grammar. In the presence of non uniform data types, we cannot avoid the generation of infinite grammars. Consider the data type:

```
data T13 a = T13 (T13 (a, a)) | C13 a
```

To generate the grammar of *T13 a*, we need the grammar of *T13* (a, a), that needs the grammar of *T13* $((a, a), (a, a))$, and so on. Note that all grammars are of different type, so we cannot use the approach defined before.

Another type that cannot be represented with our approach, because is also a kind of non uniform type, is the fix-point type:

```
data Fix f = In (f (Fix f))
```

In these cases we have to resort to the current way the read function works.

3.6 Deriving Data Type Grammars

To automatically derive the data type grammars, we use Template Haskell. While you can do most of the introspection needed also with *Data.Generics* (Lämmel and Peyton Jones 2003, 2004), we specifically need the fixity information of infix constructors for our grammar, which is not available from *Data.Generics*.

We first need to find out if the type is part of a mutually recursive group. Then we generate code for all types in the group, but only construct an instance for the type *deriveGrammar* was called on.

3.6.1 Calculating binding groups

The algorithm that finds the set of types that is mutually recursive is pretty straightforward: recursively getting the information of the types used in the constructors, while building a graph of types.

To make sure we do not loop, we stop when we find a type that is already in the graph. This works fine, but for types of a kind other than $*$, we need to take the type arguments into account . We bind the arguments in the environment and we do not recurse if we have done so with the same arguments before.

3.6.2 Generating *Gram* instances

Using the binding group, we generate the *DLNontDefs* for each of the types. This is straightforward: for a normal constructor we add a non-terminal at precedence level 10, using the constructor as term and it is arguments as references to non-terminals. For infix constructors we use the precedence and associativity information to add the at the right precedence. For each types we add a special non-terminal for parentheses.

When we need a reference to another grammar we use a naming scheme using the type, bindings (if applicable) and a prefix. For references to grammars that are not known at compile time we use the argument name, prefixed by the type and a general prefix.

When we have all the generated *DLNontDefs* we can chain them together using *consD*. For types that take arguments, we add a *consG grammar* for each argument. In the resulting environment, there will still be variables for references to grammars that

are not defined yet. We solve this by wrapping the definitions in two lambda expressions. The inner expression makes the mapping from the 'polymorphic' grammars to names (using explicit polymorphic signatures in the patterns). The outer lambda is used to create the mappings for the parametrised grammars.

As an example, when calling $\$(deriveGrammar\ “T8)$ the generated code looks like Figure 7.

```
instance Gram T8
  where grammar = DGrammar Zero
    ((λ_t_T8 _t_T2'T8 →
      (λ(_nonts_T8 :: ∀ env . Ref T8 env
                         → Ref (T2 T8) env
                         → DLNontDefs T8 env)
        (_nonts_T2 :: ∀ env a_0 . Ref (T2 a_0) env
                         → Ref a_0 env
                         → DLNontDefs (T2 a_0) env)
        →
        consD (_nonts_T8 _t_T8 _t_T2'T8)
          (consD (_nonts_T2 _t_T2'T8 _t_T8) Empty))
      (λ_r_T8 _r_T2'T8 → DLNontDefs
          [(DRef (_r_T8, 10), DPS [
              ((.#.) $ dTerm "T8")
              ((.#.) (dNont (_r_T2'T8, 0))
              (dEnd (λarg1 _ → T8 arg1)))
             , ((.#.) $ dTerm "C8") (dEnd (λ_ → C8))
             , dTerm "("
               .#. (dNont (_r_T8, 0)
               .#. (dTerm ")" .#. dEnd parenT))
              ])])
      (λ_r_T2 _r_T2_a → DLNontDefs [...]))
    Zero (Suc Zero)
    :: Env DGram ((T8, (T2 T8, ())))
                 ((T8, (T2 T8, ())))
```

Figure 7. Generated grammar of type *T8*

4. Typed Transformations

In this section we present the approach used in implementing the transformations:

$$
\begin{aligned}
&\mathit{group} &&:: \mathit{DGrammar}\ a \rightarrow \mathit{Grammar}\ a\\
&\mathit{leftcorner} &&:: \mathit{Grammar}\ a \quad \rightarrow \mathit{Grammar}\ a\\
&\mathit{leftfactoring} &&:: \mathit{Grammar}\ a \quad \rightarrow \mathit{Grammar}\ a
\end{aligned}
$$

All these functions are implemented by using the typed transformation library constructed by Baars and Swierstra (Baars and Swierstra 2008). In the following subsections we introduce the library and describe the implementation of the function *group*. The function *leftcorner* has been presented in the mentioned paper, and *leftfactoring* has a quite similar structure. [1]

4.1 Transformation Library

The library is based on the type *Trafo*, which represents typed transformation steps which modify an *Env*. Each type parameter of *Trafo* is lifted with respect to the final environment, except for the meta data in the first parameter, which depends on the type of the maintained environment at the start of the transformation:

[1] The code of the library and the transformation functions can be found at `http://www.cs.uu.nl/wiki/bin/view/Center/TTTAS`.

```
Trafo :: (* → *)               -- meta-data
      → (* → * → *)            -- type of the terms
      → (* → *)                -- input
      → (* → *)                -- output
      → *
```

The second argument describes the type of the terms in the maintained environment, and the next two arguments provide an arrow like interface, but of higher kind.

When we run a transformation we start with an empty environment and an initial value. Since this argument type is labeled with the final environment, which we do not know yet, is has to be a polymorphic value.

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

The *Result* contains the meta data, the output type and the final environment. Since in general we do not know how many new non-terminals and of which types are introduced by the transformation the result is existential in the final environment s. Despite this existentiality, we can enforce the environment to be closed.

$$\mathbf{data}\ Result\ m\ t\ b = \forall\ s\ .\ Result\ (m\ s)\ (b\ s)\ (Env\ t\ s\ s)$$

During the transformation we create references to types using *newSRef*, which takes as input a typed term, adds this to the environment, and returns the reference to the value.

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

We compose transformations in an arrow-like style. Unfortunately a *Trafo* is not really an *Arrow*, because the type arguments are of kind $(* \to *)$ instead of $*$. We provide a short overview of the interface.
The *arr* combinator lifts a function.

$$arr :: (\forall\ s\ .\ a\ s \to b\ s) \to Trafo\ m\ t\ a\ b$$

The >>> combinator composes two *Trafo*s, connecting the output of the first to the input of the second one.

$$(\ggg) :: Trafo\ m\ t\ a\ b \to Trafo\ m\ t\ b\ c$$

$$\rightarrow \mathit{Trafo}\ m\ t\ a\ c$$

The functions *first* and *second* apply part of the input (first and second component, respectively) to the argument *Trafo*, copying the rest unchanged to the output. The type *Tuple* is used to tuple types that are polymorphic in the final environment, having again something polymorphic in this environment.

$$\mathbf{newtype}\ \mathit{Tuple}\ a\ b\ s = \mathit{TP}\ (a\ s, b\ s)$$

$$\begin{aligned}
\mathit{first} &:: \mathit{Trafo}\ f\ t\ a\ b \rightarrow \mathit{Trafo}\ f\ t\ (\mathit{Tuple}\ a\ c)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Tuple}\ b\ c)\\
\mathit{second} &:: \mathit{Trafo}\ m\ t\ b\ c \rightarrow \mathit{Trafo}\ m\ t\ (\mathit{Tuple}\ d\ b)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Tuple}\ d\ c)
\end{aligned}$$

The combinators *** and &&& compose two *Trafo*s in a "parallel" way. The first one takes the input as a *Tuple*, splitting it into two inputs, while the combinator &&& uses the same input for the two *Trafo*s. The outputs of the combined *Trafo*s are tupled into a single output in both cases.

$$\begin{aligned}
(\texttt{***}) &:: \mathit{Trafo}\ m\ t\ b\ c \rightarrow \mathit{Trafo}\ m\ t\ b'\ c'\\
&\quad \rightarrow \mathit{Trafo}\ m\ t\ (\mathit{Tuple}\ b\ b')\ (\mathit{Tuple}\ c\ c')\\
(\texttt{\&\&\&}) &:: \mathit{Trafo}\ m\ t\ b\ c \rightarrow \mathit{Trafo}\ m\ t\ b\ c'\\
&\quad \rightarrow \mathit{Trafo}\ m\ t\ b\ (\mathit{Tuple}\ c\ c')
\end{aligned}$$

The function *loop* takes as argument a *Trafo* with input of type $\mathit{Tuple}\ a\ x$ and output of type $\mathit{Tuple}\ b\ x$. The second component

is fed-back (the output is passed as input). The function results in a *Trafo* with input of type a and output b.

$$\begin{aligned}
\mathit{loop} &:: \mathit{Trafo}\ m\ t\ (\mathit{Tuple}\ a\ x)\ (\mathit{Tuple}\ b\ x)\\
&\quad \rightarrow \mathit{Trafo}\ m\ t\ a\ b
\end{aligned}$$

The combinator *sequenceA* composes a list of *Trafo*s with input a and output b, as a *Trafo* with input a and output a list of outputs generated sequentially by each *Trafo* of the composed list.

$$\mathbf{newtype}\ List\ a\ s = List\ [a\ s]$$
$$sequenceA :: [Trafo\ m\ t\ a\ b] \rightarrow Trafo\ m\ t\ a\ (List\ b)$$

4.2 Implementation of Grouping

The function *group* splits the grammar into parts, depending on the precedence, while changing the representation of the grammar to the one used in the implementation of the left-corner transform:

$$group :: DGrammar\ a \rightarrow Grammar\ a$$

4.2.1 References Mapping

The transformation has to map references in a *DGrammar* with explicitly indicated precedences to a representation where all elements represent normal non-terminals. So, we have to transform the *DRef*s references into the old representation to *Ref*s into the new environment. We introduce a *DRef*-transformer for this conversion, where *env1* describes the types of the old non-terminals and *env2* those of the new non-terminals:

$$\mathbf{newtype}\ DT\ env1\ env2$$
$$= DT\{unDT :: \forall\ a\ .\ DRef\ a\ env1 \rightarrow Ref\ a\ env2\}$$

With this transformer we map each production into its new representation using references into new environment. This is done by applying *unDT* to each non-terminal reference in the production:

$$mapDP2Prod\ ::\ DT\ env1\ env2 \rightarrow DProd\ a\ env1$$
$$\rightarrow Prod\ a\ env2$$

$$
\begin{aligned}
&mapDP2Prod\ t\ (DEnd\ x) = End\ x\\
&mapDP2Prod\ t\ (DSeq\ (DNont\ x)\ r)\\
&\qquad\qquad = Seq\ (Nont\ (unDT\ t\ x))\\
&\qquad\qquad\qquad (mapDP2Prod\ t\ r)\\
&mapDP2Prod\ t\ (DSeq\ (DTerm\ x)\ r)\\
&\qquad\qquad = Seq\ (Term\ x)\\
&\qquad\qquad\qquad (mapDP2Prod\ t\ r)
\end{aligned}
$$

The function *dp2prod* lifts *mapDP2Prod* using the combinator *arr*. Thus, it takes a *DProd* and returns a transformation that has as output a *Prod*, which is a production in the new environment.

$$
\begin{aligned}
&dp2prod :: DProd\ a\ env\\
&\qquad \rightarrow Trafo\ Unit\ Productions\ (DT\ env)\ (Prod\ a)\\
&dp2prod\ p = arr\ (\lambda env2s \rightarrow mapDP2Prod\ env2s\ p)
\end{aligned}
$$

The type of the resulting *Trafo* indicates that the transformation creates an environment of *Productions* (a *Grammar*).

Each precedence level definition is converted to a non-terminal in the new grammar, using the function *ld2nt*. This function takes a pair of type $(DRef\ a\ env, DProductions\ a\ env)$, that defines a level of precedence, and creates the new non-terminal, returning a reference to it. The transformation made by *dp2prod* is applied to all the elements of the list of alternative productions (*DProductions*) using *sequenceA*, in order to obtain a list of alternative productions in the new grammar (*Productions*). In parallel, the function *mkNxtLev* creates a new production to add to the list, that directly refers to the next level of precedence, if the represented level is less than 10.

$$
\begin{aligned}
&ld2nt :: (DRef\ a\ env, DProductions\ a\ env)\\
&\qquad \rightarrow Trafo\ Unit\ Productions\ (DT\ env)\ (DRef\ a)\\
&ld2nt\ (DRef\ (rnt, i), DPS\ lp)
\end{aligned}
$$

```
        = (sequenceA (map dp2prod lp) &&& mkNxtLev)
          >>> arr  (λ(TP (List ps, PS nl))
                          → PS $ nl ++ ps)
              >>> newSRef >>> arr (λr → DRef (r, i))
  where
      mkNxtLev = arr $ λt → PS $
        if (i < 10)
          then [Seq (Nont $ unDT t $
                                DRef (rnt, i + 1))
                     (End id)]
          else  []
```

Then the possible new production (or an empty list otherwise) is appended to the mapped alternative productions, generating the list that is combined with the creation of a new reference. This new reference is the new non-terminal, which stores its productions. The reference and the precedence level that represents are the output of the transformation.

By applying this transformation to a list of definitions of precedence levels we obtain a list of *DRef*s:

```
newtype ListDR a s = ListDR ([DRef a s])
```

We now apply this transformation to all the defined levels of precedence in all the non-terminal definitions and recursively to all the referenced grammars. In this way we construct a mapping from the references in the original environment to references in the transformed one.

```
newtype DMapping o n = DMapping (Env ListDR n o)
```

A *DRef*-transformer can be obtained from the *DMapping* by constructing a function that takes a *DRef a env*, looks up the reference in the environment and subsequently locates the appropriate precedence level in the list:

```
dmap2trans :: DMapping env s → DT env s
```

```
dmap2trans (DMapping env)
   = DT (λ(DRef (r, i))
             → case (lookupEnv r env) of
                    ListDR rs → (plookup i rs))
```

Having an ordered list of *DRef*s, the function *plookup* returns the first reference (*Ref*) that applies to a given preference level.

```
plookup :: Int → [DRef a s] → Ref a s
plookup i []    = error "Wrong Grammar!!"
plookup i ((DRef (r, p)) : drs)
   | i ⩽ p      = r
   | otherwise = plookup i drs
```

4.2.2 Transformation

The function *group* runs a *Trafo* that generates the new environment and returns as output the reference of the starting point (precedence level 0 in the main non-terminal). We construct the new grammar by taking the output and the constructed environment from the *Result*.

```
group :: DGrammar a → Grammar a
group gram
   = let r = runTrafo
               (gGrammar gram
                 >>> arr (λ(ListDR rs) → (plookup 0 rs)))
               Unit ⊥
     in case r of Result _ r gram → Grammar r gram
```


The function *gGrammar* implements the grammar transformation. It takes a *DGrammar* and returns a transformation that constructs the "grouped" environment and has as output the list of new references of the main non-terminal.

$$
\begin{aligned}
&gGrammar :: DGrammar\ a \\
&\qquad\qquad \to Trafo\ Unit\ Productions\ t\ (ListDR\ a) \\
&gGrammar\ (DGrammar\ r\ gram) \\
&\quad = loop\ \$ \\
&\qquad arr\ (\lambda(TP\ (\_, menv\_s)) \to menv\_s) \\
&\qquad >\!\!>\!\!>\ (arr\ (\lambda(DMapping\ env) \to lookupEnv\ r\ env) \\
&\qquad\qquad \&\&\&\ (arr\ (\lambda menv\_s \to dmap2trans\ menv\_s) \\
&\qquad\qquad\qquad >\!\!>\!\!>\ gDGrams\ gram))
\end{aligned}
$$

The function applies the transformation returned by *gDGrams* to the elements of the environment. This transformation takes as input a *DRef*-transformer, mapping all non-terminals from the original environment to the newly generated one. The output is a *DMapping* which remembers the new locations of the non-terminals from the original grammar. To obtain the needed *DRef*-transformer for this transformation, the function *gGrammar* uses a feed-back loop using the *DMapping* returned by the transformation itself. To obtain the list of mapped references for the main non-terminal it just looks up the reference in the *DMapping*.

The function *gDGrams* iterates (by induction) over the environment that contains the non-terminal definitions and the grammars referenced by them.

$$
\begin{aligned}
&gDGrams :: Env\ DGram\ env\ env' \\
&\qquad\qquad \to Trafo\ Unit\ Productions \\
&\qquad\qquad\qquad\qquad (DT\ env)\ (DMapping\ env') \\
&gDGrams\ env = mapTrafoEnv\ tr\ env \\
&\quad \mathbf{where}
\end{aligned}
$$

```
tr (DGG gram) = gGrammar gram
tr (DGD (DLNontDefs nonts))
                  = (sequenceA (map ld2nt nonts))
                    >>> arr (λ(List r) → ListDR r)
```

In the case of a grammar, the function *gGrammar* is invoked. The output of this transformation is the list of new references assigned to the main non-terminal of this grammar. The list is added to the *DMapping* in the place of the grammar.

In the case of a list of precedences (a non-terminal), we *map* the function *ld2nt* to the list, obtaining a list of transformations. Each transformation adds a new non-terminal to the new grammar and returns the new reference and the precedence level that represents. We execute this transformations sequentially (using *sequenceA*) and add the resulting list of references to the *DMapping*.

The iteration over the environment is performed by the function *mapTrafoEnv*.

```
mapTrafoEnv :: (∀ a . t a env
                  → Trafo Unit tf af (ListDR a))
            → Env t env env'
            → Trafo Unit tf af (DMapping env')
mapTrafoEnv _ Empty
  = arr (const (DMapping Empty))
mapTrafoEnv t (Cons x xs)
  = (t x &&& mapTrafoEnv t xs)
    >>> arr (λ(TP (r, DMapping rxs))
                → DMapping (Cons r rxs))
```

5. Efficiency

In this section we show some experimental results about the efficiency of our approach[2]. First of all compare *read* and *gread* in the presence of infix constructors. Finally we show how the presence of the left-factoring optimisation influences efficiency.

5.1 *gread* versus *read*

In Figure 8 we show the execution times resulting from the use of *read* and *gread* to parse an expression of the form `C1 :>: (C1 :>: ...)`, where n is the number of constructors `C1` the expression has.

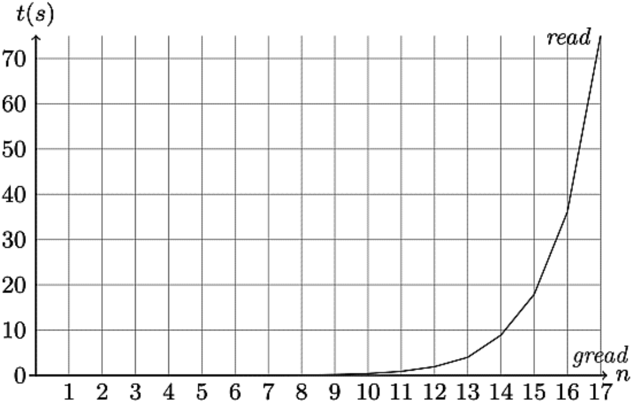


Figure 8. Execution times of reading `C1` :> (`C1 :>: ...`)

The function *read* clearly has an exponential behaviour. It takes 75 seconds to resolve the case with 17 `C1`s and does not run after 18. On the other hand, the function *gread* maintains negligible times. If we do not use parentheses we can read 50000 `C1`s within a second.

We obtain similar behaviour with (... `:<:` `C1`) `:<:` `C1`. Note that this is a bad case for the function *read*, due to the opening parentheses. The function *read* takes 23 seconds to resolve the case with 9 `C1`s (does not run after 10), while the function *gread* requires negligible times: more than 40000 `C1`s can be read within a second, without the extra parentheses.

Data type grammars are usually very small, but in order to test our approach in its worst case, we defined a large data type of the form:

$$\begin{aligned}
&\textbf{data}\ \mathit{TBig}\ t1\ t2\ t3\ t4\ t5\ t6\ t7\ t8\ t9\ t10\\
&\quad = \mathtt{CB}\\
&\quad \mid \mathtt{TB1}\ (\mathit{TBig}\ t1\ t2\ t3\ t4\ t5\ t6\ t7\ t8\ t9\ t10)\\
&\quad \mid \ldots\\
&\quad \mid \mathtt{TB}n\ (\mathit{TBig}\ t1\ t2\ t3\ t4\ t5\ t6\ t7\ t8\ t9\ t10)
\end{aligned}$$

where n is a number between 10 and 100. Note that the type has 10 parameters and no infix constructors. So a relatively large combination and transformation effort is needed, while the optimisations do not add anything. We tested this type with an expression `TB`n (...(`TB`n `CB`)...) with 10000 constructors.
We can see in Figure 9 that the function *gread* has linear behaviour. From this case we can conclude that the time needed to perform the transformations is almost negligible. We have performed the same tests using the expressions `TB1` (...(`TB1` `CB`)...) and `TB`$\frac{n}{2}$ (...(`TB`$\frac{n}{2}$ `CB`)...) obtaining similar results.

[2] The tests were run in a computer with 1.6 GHz Intel Core Duo processor and 1 GB RAM.

$t(s)$

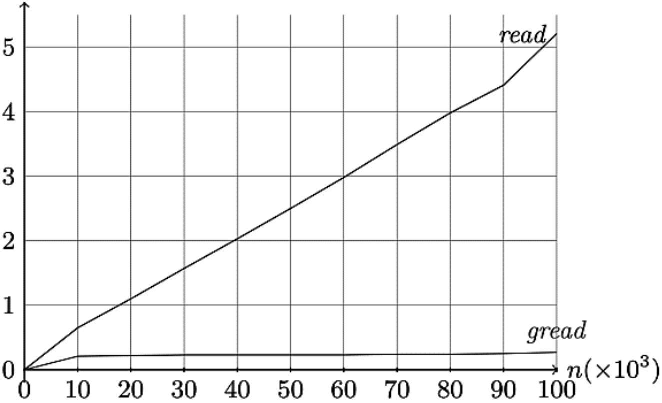


Figure 9. Execution times of reading a large data type

5.2 *gread* versus *leftcorner*

We have shown that the *gread* function has efficient behaviour in comparison with the Haskell *read*. But what happens if we do not include the left-factoring?

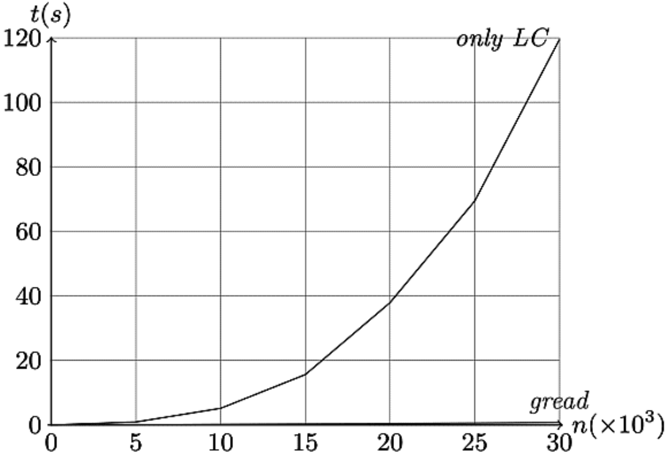


Figure 10. Execution times of reading `C1 :<: ... :<: C1` with and without left-factoring

As we can see in Figure 10 (expression `C1 :<: ... :<: C1`) the inclusion of left-factoring improves the efficiency by avoiding duplicate parsing.

We have tested both functions in situations where the left-factoring is not needed and they behave in a similar way; so the extra transformation work and the few extra non-terminals add little to the total cost of parsing. For example, in the case of Table 1 there are no common prefixes in the evaluated productions while only applying the LC-transform.

6. Conclusions and Future Work

We have shown an alternative way to implement the *read* (and consequently also the *show*) functions. We read data in linear time, generate shorter output, and the overhead caused by generating the read functions at runtime does not seem to be a problem; not even for very large data types. Unfortunately we are not able to handle nested data types which have infix constructors; for these one has to write the parsing functions by hand. Note that this problem only occurs if the nested data type occurs at the left-hand side of an infix type constructor, and that in such cases also the conventional solution is problematic.

n ($\times 10^3$)	*gread* (s)	*only LC* (s)
10	0.14	0.14
20	0.33	0.32
30	0.57	0.56
40	0.82	0.82
50	1.13	1.11
60	1.47	1.47
70	1.82	1.81
80	2.23	2.22
90	2.68	2.67
100	3.18	3.15

Table 1. Execution times of reading `C1 :>: ... :>: C1` with and without left-factoring

Besides the completely dynamic implementation which we have presented in which we compose all grammars at runtime, a large part of the work could be done by the Haskell compiler at compilation time.

We consider the Template Haskell implementation to be a prototype. Further optimisations are to tuple grammars with their corresponding parser. If we know there are no problems with common prefixes or left-recursion we can resort to simpler parsing methods, and generate parsers only once by sharing them.

Straightforward extensions are the inclusion of a generator for record constructors. An open research problem is how to merge in the techniques for parsing record fields in arbitrary order, since the proposed solution (Baars et al. 2004) critically depends on the dynamic generation of parsers; we expect lazy evaluation to save us here. Finally, we need a more robust naming scheme to deal with problems due a similarly named types coming from different modules.

7. Acknowledgments

We thank Simon Peyton-Jones for bringing the problem under our attention and Koen Claessen for discussing the impossibilities of simpler approaches. We also thank the anonymous reviewers, Atze Dijkstra, Andres Löh and Thomas Van Noort who provided valuable comments on drafts of this paper.

References

Arthur Baars and Doaitse Swierstra. Typed transformations of typed abstract syntax. UU-CS 21, Utrecht University, 2008.

Arthur Baars, Andres Löh, and Doaitse Swierstra. Parsing permutation phrases. *J. Funct. Program.*, 14(6):635–646, 2004. ISSN 0956-7968.

Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017485.

Eric Bouwers, Martin Bravenboer, and Eelco Visser. Grammar engineering support for precedence rule recovery and compatibility checking. *Electron. Notes Theor. Comput. Sci.*, 203(2):85–101, 2008. ISSN 1571-0661.

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

M. Johnson. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL Õ98, Montreal, Quebec, Canada*, pages 619–623. Association for Computational Linguistics, 1998.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38

(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007. doi: 10.1017/S0956796807006326.

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.

JC Petruzza, Koen Claessen, and Simon Peyton Jones. Derived read instances for recursive datatypes with infix constructors are too inefficient. URL `http://hackage.haskell.org/trac/ghc/ticket/1544`. GHC Ticket 1544.

Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and

Geoffrey Washburn. Simple nification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002. ISBN 1-58113-605-6.

S.D. Swierstra. The Utrecht Parsing Libraries. `http://www.cs.uu.nl/wiki/bin/view/HUT/ParserCombinators`, July 2008.

A. Additional Parser Combinators

```
pChainr op x   = r
                 where r = x <??> (flip <$> op <*> r)
p <??> q = p <**> (q `opt` id)
p <**> q = flip ($) <$> p <*> q
pChainl op e   = foldl (flip ($))
                 <$> e <*> pMany (flip <$> pOp op <*> e)
pOp (tok, sem) = const sem <$> pToken tok
pParens p = (λ_ v _ → v)
            <$> pToken "(" <*> p <*> pToken ")"
```

B. Parser Generation

```
newtype Const f a s = C{unC :: f a}
compile :: Grammar a → Parser Token a
compile (Grammar (start :: Ref a env) rules)
        = unC (lookupEnv start result)
  where
    result =
      mapEnv
```

```
    (λ(PS ps) → C (foldr1 (<|>)
                          [comp p | p ← ps]))
    rules
comp :: Prod a env → Parser Token a
comp (End x) = pLow x
comp (Seq (Term t) ss)
  = (flip ($)) <$> pSym t <*> comp ss
comp (Seq (Nont n) ss)
  = (flip ($)) <$> unC (lookupEnv n result)
               <*> comp ss
```