Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 51–64
www.elsevier.com/locate/entcs

Typed Transformations of Typed Grammars: The Left Corner Transform

Arthur Baars[1]

Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Valencia, Spain

S. Doaitse Swierstra[2]

Department of Computer Science
Utrecht University, Utrecht, The Netherlands

Marcos Viera[3]

Instituto de Computación
Universidad de la República, Montevideo, Uruguay

Abstract

One of the questions which comes up when using embedded domain specific languages is to what extent we can analyze and transform embedded programs, as normally done in more conventional compilers. Special problems arise when the host language is strongly typed, and this host type system is used to type the embedded language. In this paper we describe how we can use a library, which was designed for constructing transformations of typed abstract syntax, in the removal of left recursion from a typed grammar description. The algorithm we describe is the Left-Corner Transform, which is small enough to be fully explained, involved enough to be interesting, and complete enough to serve as a tutorial on how to proceed in similar cases. The described transformation has been successfully used in constructing a compositional and efficient alternative to the standard Haskell *read* function.

Keywords: GADT, Left-Corner Transform, Meta Programming, Type Systems, Typed Abstract Syntax, Typed Transformations

1 Introduction

alised (i.e. written) and deserialised (i.e. read or parsed). Since data types can be passed as

[1] Email: abaars@iti.upv.es

[2] Email: doaitse@cs.uu.nl

[3] Email: mviera@fing.edu.uy

doi:10.1016/j.entcs.2010.08.031

parameter to data type constructors, and definitions can be spread over several modules, the question arises how to dynamically combine separately generated pieces of "reading code" into a function *read* for a composite data type. The standard solution in Haskell, based on a straightforward combination of top-down parsers, has turned out to exhibit exponential reading times. Furthermore, in order to avoid the dynamic construction of left recursive top-down parsers at run-time the input is required to contain many more parentheses than one would expect.

In a recent paper [12] we have presented a solution to this problem; instead of generating code which reads a value of some data type a, the compiler constructs a value of type $Grammar\ a$ which represents the piece of grammar that describes external representations of values of that data type. The striking feature of this grammar type is that it reflects the type of values represented. This is necessary, since from such a value eventually a *read* function of type $String \rightarrow a$ has to be constructed by Haskell library code.

Our purpose is to split the parsing process in two parts, when dealing with possibly left-recursive grammars at run-time. Instead of using left-corner parsers, having to analyze the grammar every time we parse, the grammar is transformed once to remove left-recursion and then conventional efficient parsing techniques can be used. The solution builds upon three, more or less independent, layers (from top to bottom):

(i) A template Haskell library which generates the values of type $Grammar\ a$ and library code which combines such values at run-time to form a complete grammar. Out of this combined value the desired read function for the composed data type is constructed, again by library Haskell code. This whole process is described in the aforementioned paper [12].

(ii) This code calls a library function which removes potential left-recursion from the composed grammar. For this we use the Left-Corner Transform (LCT) [7]. This code, which produces a function of type $Grammar\ a \rightarrow Grammar\ a$, is a fine example of how to express transformations of typed abstract syntax containing references; in the $Grammar\ a$ case these stem from occurrences of non-terminal symbols in the right hand sides of the productions.

(iii) The LCT and the left-factoring code make use of an intricate Haskell library, which exploits every corner of the Haskell type system and its extensions, such as Generalised Algebraic Data Types, existential and polymorphic types, and lazy evaluation at the type level. The design alternatives and the final design of the library, as it has been made available to the Haskell world, deserved a paper of its own [1].

In this paper we focus on the middle of the above three layers; we start out by presenting an elegant formulation of the LCT in combination with an untyped Haskell implementation, next we introduce the API as implemented by the bottom layer, and we finish by reformulating the untyped version into a typed one using this API.

The LCT [6] is more involved than the direct left recursion removal given in

[2], but is also more efficient ($O(n^2)$, where n is the number of terminals and non-terminals in the grammar). Here we will start from an improved version formulated by Robert C. Moore [7], which we present in a more intuitive form. Both his tests, using several large grammars for natural language processing, and our tests [12], using several very large data type descriptions, show that the algorithm performs very well in practice.

What makes this transformation interesting from the typed abstract syntax point of view is that a grammar consists of a collection of grammar rules (one for each non-terminal) *containing references to other definitions*; we are thus not transforming a tree but a complete binding structure. During this transformation we introduce many new definitions. In the right hand side of these definitions we again use references to such newly introduced symbols. In our setting a transformation must be type preserving and we thus have to ensure that the types of the environment and the references remain consistent, *while being modified*. Previous work on typeful program transformations [3,8,2] cannot handle such introductions of new definitions and binders.

We present the algorithm in terms of Haskell code, and thus require Haskell knowledge from the reader. Please keep in mind however that Haskell currently is one of the few general purpose languages in which the problem we describe can be solved at all.

2 Left-Corner Transform

In this section we introduce the LCT [6] as a set of construction rules and subsequently give an untyped implementation in Haskell98. Note that, despite being called a transformation, the process is actually constructing a new grammar while inspecting the input grammar. We assume that only the start symbol may derive ϵ.

We say that a symbol X is a *direct left-corner* of a non-terminal A, if there exists a production for A which has the symbol X as its left-most symbol in the right-hand side of that production. We define the *left-corner* relation as the transitive closure of the direct left-corner relation. Note that a non-terminal being left-recursive is equivalent to being a left-corner of itself.

The LCT is defined as the application of three surprisingly simple rules. We use lower-case letters to denote terminal symbols, low-order upper-case letters (A, B, etc.) to denote non-terminals from the grammar and high-order upper-case letters (X, Y, Z) to denote symbols that can either be terminals or non-terminals. Greek symbols denote sequences of terminals and non-terminals.

For a non-terminal A of the original grammar the algorithm constructs new productions for A, and a set of new definitions for non-terminals of the form A_X. A new non-terminal A_X represents that part of A which is still to be recognised after having seen an X. The following rules are applied for each non-terminal until no further results are obtained:

Rule 1 For each production $A \rightarrow X\ \beta$ of the original grammar add $A\_X \rightarrow \beta$ to the transformed grammar, and add X to the left-corners of A.

Rule 2 For each newly found left-corner X of A:

a If X is a terminal symbol add $A \to X\ A\_X$ to the new grammar.

b If X is a non-terminal then for each original production $X \to X'\ \beta$ add the production $A\_X' \to \beta\ A\_X$ to the new grammar and add X' to the left-corners of A.

As an example consider the grammar:

$$
\begin{array}{l}
A \to a\ A \mid B \\
B \to A\ b \mid c
\end{array}
$$

Applying rule 1 for the productions of A results in two new productions and two newly encountered left-corners:

$$
\begin{array}{lll}
 & A\_a \to A & \\
 & A\_B \to \epsilon & \mathit{leftcorners} = [a, B] \\
\text{rule 2a with } X \text{ bound to the left-corner } a \Rightarrow & & \\
 & A \to a\ A\_a & \mathit{leftcorners} = [\cancel{a}, B] \\
\text{rule 2b with } X \text{ bound to the left-corner } B \Rightarrow & & \\
 & A\_A \to b\ A\_B & \\
 & A\_c \to A\_B & \mathit{leftcorners} = [\cancel{a}, \cancel{B}, A, c] \\
\text{rule 2b with } X \text{ bound to the left-corner } A \Rightarrow & & \\
 & A\_a \to A\ A\_A & \\
 & A\_B \to A\_A & \mathit{leftcorners} = [\cancel{a}, \cancel{B}, \cancel{A}, c] \\
\text{rule 2a with } X \text{ bound to the left-corner } c \Rightarrow & & \\
 & A \to c\ A\_c & \mathit{leftcorners} = [\cancel{a}, \cancel{B}, \cancel{A}, \cancel{c}]
\end{array}
$$

Since now all left-corners of A have been processed we are done with A. For the non-terminal B the process yields the following new productions:

$$
\begin{array}{ll}
B\_A \to b & \text{-- rule 1} \\
B\_c \to \epsilon & \text{-- rule 1} \\
B\_a \to A\ B\_A & \text{-- rule 2b, } A \\
B\_B \to B\_A & \text{-- rule 2b, } A \\
B \to c\ B\_c & \text{-- rule 2a, } c \\
B \to a\ B\_a & \text{-- rule 2a, } a \\
B\_A \to b\ B\_B & \text{-- rule 2b, } B \\
B\_c \to B\_B & \text{-- rule 2b, } B
\end{array}
$$

Note that by construction this new grammar is not left-recursive.

2.1 The Untyped Left-Corner Transform

Before presenting our typed LCT, we present an untyped implementation. Grammars are represented by the types:

```
type Grammar = Map NT [Prod]
type NT      = String
type Prod    = [Symbol]
type Symbol  = String
isNonterminal = isUpper . head
isTerminal    = isLower . head
```

Thus a *Grammar* is a mapping which associates each non-terminal name with its set of productions. Each production (*Prod*) consists of a sequence of symbols (*Symbol*). So our example grammar can be encoded as:

```
grammar = Map.fromList [("A", [["a", "A"], ["B"]])
                       ,("B", [["A", "b"], ["c"]])]
```

In the transformation process we use the *Control.Monad.State*-monad to store the thus far constructed new grammar. For each non-terminal we traverse the transitive left-corner relation as induced by the productions in depth-first order, while caching the set of thus far encountered left-corner symbols in a list:

```
type LeftCorner = Symbol
type Step_State = (Grammar, [LeftCorner])
type Trafo a = State Step_State a
```

The function *leftcorner* takes a grammar and returns a transformed grammar by running the transformation *rules1*, which yields a value of the monadic type *Trafo*. The state is initialized with an empty grammar and an empty list of encountered left-corner symbols. The final state contains the newly constructed grammar:

```
leftcorner :: Grammar → Grammar
leftcorner g = fst . snd . runState (rules1 g g) $ (Map.empty, [])
```

For each (*mapM_*) non-terminal (A) the function *rules1* visits each (*mapM*) of its productions; each visit results in new productions using *rule2a* and *rule2b*. They are added to the transformed grammar by the function *insert*. The productions resulting from *rule2a* are returned (*ps*), and together (*concat*) from the new productions for the original non-terminal A. The left-corners cache is reset when starting with the next non-terminal:

```
rules1 :: Grammar → Grammar → Trafo ()
rules1 gram nts = mapM_ nt (Map.toList nts)
  where nt (a, prods) =
                  do ps ← mapM (rule1 gram a) prods
                     modify  (λ(g, _) → (Map.insert a (concat ps) g, []))
```

For each of the rules given we define a function: *rule2b* generates new productions for non-terminals of the original grammar, and *rule1* and *rule2b* generate productions for non-terminals of the form A_X:

```
rule1 :: Grammar → NT → Prod → Trafo [Prod]
rule1 grammar a (x : beta) = insert grammar a x beta

rule2a :: NT → Symbol → Prod
rule2a a_b b = [b, a_b]

rule2b :: Grammar → NT → NT → Prod → Trafo [Prod]
rule2b grammar a a_b (y : beta) = insert grammar a y (beta ++ [a_b])
```

The function *insert* adds a new production for a non-terminal A_X to the grammar: if we have met A_X before, the already existing entry is extended and otherwise a new entry for A_X is added. In the latter case we apply *rule2* in order to find further left-corner symbols:

```
insert :: Grammar → NT → Symbol → Prod → Trafo [Prod]
insert grammar a x p =
  do let a_x = a ++ "_" ++ x
     (gram, lcs) ← get
     if x ∈ lcs then do put (Map.adjust (p:) a_x gram, lcs)
                        return []
                else  do put (Map.insert a_x [p] gram, x : lcs)
                        rule2 grammar a x
```

In *rule2* new productions resulting from applications of *rule2b* are directly inserted into the transformed grammar, whereas the productions resulting from *rule2a* are collected and returned as the result of the *Trafo*-monad. When the newly found left-corner symbol is a terminal *rule2a* is applied, and the resulting new production rule is simply returned. If it is a non-terminal, its corresponding productions are located in the original grammar and *rule2b* is applied to each of them:

```
rule2 :: Grammar → NT → Symbol → Trafo [Prod]
rule2 grammar a b
  | isTerminal b = return [rule2a a_b b]
```

```
  | otherwise     = do let Just prods = Map.lookup b grammar
                          rs ← mapM (rule2b grammar a a_b) prods
                          return (concat rs)
  where a_b = a ++ "_" ++ b
```

Note that the functions *rule2* and *insert* are mutually recursive. They apply the rules 2a and 2b until no new left-corner symbols are found. The structure of the typed implementation we present in section 4 closely resembles the untyped solution above.

3 Typed Transformations

The typed version of the LC transform is implemented by using a library (TTTAS [4]) we described in a companion paper [1] to perform typed transformations of typed abstract syntax (in our case typed grammars). In the following subsections we introduce the basic constructs for representing typed abstract syntax and the library interface for manipulating it.

3.1 Typed References and Environments

Pasalic and Linger [8] introduced an encoding *Ref* of typed references pointing into an environment containing values of different type. A *Ref* is actually an index labeled with both the type of the referenced value and the type of the environment (a nested Cartesian product, growing to the right) the value lives in:

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

The type *Ref* is a generalized algebraic data type [10]. 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 b), and remembers a position in the rest of the environment.

We 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 into 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. 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.

```
data Env term 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 all the type labels a of the terms of type $t\ a\ use$ occurring in the environment. When a term is added to the environment using *Ext*, its type label is included as the first component of *def*. The type *use* describes the

[4] http://hackage.haskell.org/cgi-bin/hackage-scripts/package/TTTAS.

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 into which the reference points. The occurrence of the two *env*'s in the type of *lookupEnv* guarantees that the lookup will succeed, and that the value found is indeed labeled with the type with which the *Ref* argument was labeled, which is encoded by the two occurrences of *a*:

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero    (Ext _ t)  = t
lookupEnv (Suc r) (Ext ts _) = lookupEnv r ts
```

3.2 Transformation Library

The library is based on the type *Trafo*, which represents typed transformation steps. Each transformation step (possibly) extends an implicitly maintained environment *Env*.

```
data Trafo m t s a b = Trafo (∀env1 . m env1 → TrafoE m t s env1 a b)
```

The argument *m* stands for the type of the observable state of the transformation. A *Trafo* takes such a state value, which depends on the environment constructed thus far, as input and yields a new state corresponding to the (possibly extended) environment. The type *t* is the type of the terms stored in the environment. The type variable *s* represents the type of the final result, which is passed as the *use* argument in the embedded references. We compose transformations in an arrow style. The arguments *a* and *b* are the *Arrow*'s input and output, respectively. The *Arrow* library [5] contains a set of functions for constructing and combining values that are instance of the *Arrow* class. Furthermore there is a convenient notation [9] for programming with *Arrow*s. This notation is inspired by the **do**-notation for *Monad*s. The class *ArrowLoop* is instantiated to be able to construct feedback loops. The TTTAS library includes a combinator, analogous to the *sequence* combinator for *Monad*s, which combines a sequence of transformations into one single large transformation:

```
sequenceA :: [Trafo m t s a b] → Trafo m t s a [b]
```

Each individual transformation maps the input *a* onto a value *b*. The combined results *b* resulting from applying the individual transformations in sequence, are returned as a list [*b*].

The constructor *Trafo* contains a function which maps a state in the current environment to the actual transformation, represented by the type *TrafoE*. Because the internal details of the type *TrafoE* are of no relevance here, we do not give its definition; we only present its constructors:

```
extEnv     :: m (e, a)          → TrafoE m t s e (t a s) (Ref a s)
castSRef   :: m e → Ref a e → TrafoE m t s e i       (Ref a s)
updateSRef :: m e → Ref a e → (i → t a s → t a s)
                                → TrafoE m t s e i       (Ref a s)
```

The function *extEnv* builds a *TrafoE* which takes a typed term (of type *t a s*) as input, adds it to the environment and yields a reference pointing to this value in the final environment (*s*). The argument of *extEnv* is a state that depends on the extended environment (e, a). Thus, for example, a transformation that extends the environment without keeping any internal state can be implemented:

```
data Unit env = Unit
newSRef :: Trafo Unit t s (t a s) (Ref a s)
newSRef = Trafo (λ_ → extEnv Unit)
```

The function *castSRef* builds a *TrafoE* that returns the reference passed as parameter (in the current environment e) casted to the final environment. The function *updateSRef* builds a *TrafoE* that updates the value pointed by the passed reference. Note that the update function (of type $i \to t\ a\ s \to t\ a\ s$) can use the input of the *Arrow*. The type $(TrafoE\ m\ t\ s\ e\ a)$ is an instance of the class *Functor*, so the function

```
fmap :: (b → c) → TrafoE m t s e a b → TrafoE m t s e a c
```

lifts a function with type $(b \to c)$ and applies it to the output of the *Arrow*.

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.

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

The *Result* contains the final state $(m\ s)$, the output value $(b\ s)$ and the final environment $(Env\ t\ s\ s)$. Since in general we do not know how many new definitions and of which types are introduced by the transformation the result is existential in the final environment s. Despite this existentially, we can enforce the environment to be closed:

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

4 The Typed Left-Corner Transform

For a typed version of the LCT we need a typed representation of grammars. A grammar consists of a start symbol, represented as a reference labeled with the type that serves as the witness value of a successful parse, and an *Env*, containing for each non-terminal its list of productions. The actual type *env*, describing the types associated with the non-terminals, is hidden using existential quantification:

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

Since in our LCT we want to have easy access to the first symbol of a production we have chosen a representation which facilitates this. Hence the types of the elements in a sequential composition have been chosen a bit different from the usual one [11], such that *Seq* can be chosen to be right associative. The types have been chosen in such a way that if we close the right hand side sequence of symbols with an *End f* element, then this *f* is a function that accepts the results of the earlier elements (parsing results of the right hand side) as arguments, and builds the parsing result for the left-hand side non-terminal. In our case a production is a sequence of symbols, and a symbol is either a terminal with a *String* as its witness or a non-terminal (reference):

```
data Symbol :: ∗ → ∗ → ∗ where
  Nont :: Ref a env → Symbol a      env
  Term :: String →    Symbol String env
```

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

In order to make our grammars resemble normal grammars we introduce some extra operators:

```
infixr 5 `cons`, .*.

cons prods g = Ext g (PS prods)
(.*.) = Seq
```

We now have the machinery at hand to encode our example grammar:

```
_A = Nont        Zero
_B = Nont (Suc Zero)
_a  = Term "a"
_b  = Term "b"
_c  = Term "c"
```

Assume we want the witness type for non-terminal A to be a *String* and for non-terminal B an *Int*:

```
grammar :: Grammar String
grammar = Grammar Zero productions

type Types_nts = (((), Int), String)

productions :: Env Productions Types_nts Types_nts
productions = [_a .*. _A .*. End (++)
              , _B          .*. End show      ] `cons`
              [_A .*. _b .*. End (λy x → length x + length y)
              , _c          .*. End (const 1)] `cons` Empty
```

Before delving into the LCT itself we introduce some grammar related functions we will need:

```
append :: (a → b → c) → Prod a env → Symbol b env → Prod c env
matchSym :: Symbol a env → Symbol b env → Maybe (Equal a b)
mapProd :: T env1 env2 → Prod a env1 → Prod a env2
```

The function *append* is used in the LCT to build productions of the form $\beta\ X\_A$. Basically it corresponds to the *snoc* operation on lists; we only have to make sure that all the types match. The function *matchSym* compares two symbols and, if they are equal, returns a witness (*Equal*) of the proof that the types a and b are equal. The function *mapProd* systematically changes all the references to non-terminals occurring in a production. It takes a *Ref*-transformer ($T\ env1\ env2$) to transform references in the environment *env1* to references in the environment *env2*.

```
newtype T env1 env2 = T{unT :: ∀x . Ref x env1 → Ref x env2}
```

4.1 The Typed Transformation

The LCT is applied in turn to each non-terminal (A) of the original grammar. The algorithm performs a depth first search for left-corner symbols. For each left-corner X a new non-terminal A_X is introduced. Additionally a new definition for A itself is added to the transformed grammar.

In the untyped implementation we simply used strings to represent non-terminals. In the typed solution non-terminals are, however, represented as typed references. The first time a production for a non-terminal A_X is generated, we must create a new entry for this non-terminal and remember its position. When the next production for such an A_X is generated we must add it to the already generated productions for this A_X: hence we maintain a finite map from encountered left-corner symbols (X) to references corresponding to the non-terminals (A_X). This finite map again caches the already encountered left-corner symbols:

$$\textbf{newtype}\ \mathit{MapA\_X}\ \mathit{env}\ a\ \mathit{env2}$$
$$= \mathit{MapA\_X}\ (\forall x\ .\ \mathit{Symbol}\ x\ \mathit{env} \rightarrow \mathit{Maybe}\ (\mathit{Ref}\ (x \rightarrow a)\ \mathit{env2}))$$

The type variable *env* comes from the original grammar, and *env2* is the type of the new grammar constructed thus far. The type variable a is the type of the current non-terminal. A left-corner symbol labelled with type x is mapped to a reference to the definitions of the non-terminal A_X in the new grammar, provided it was inserted earlier. The type associated with a non-terminal of the form A_X is $(x \rightarrow a)$, i.e. a function that returns the semantics of A, when it is passed the semantics of the symbol X. The empty mapping is defined as:

$$\mathit{emptyMap} :: \mathit{MapA\_X}\ \mathit{env}\ a\ \mathit{env2}$$
$$\mathit{emptyMap} = \mathit{MapA\_X}\ (\mathit{const}\ \mathit{Nothing})$$

We introduce the type-synonym *LCTrafo*, which is the type of the transformation step of the LCT. The type of our terms is *Productions*, and the internal state is a table of type $\mathit{MapA\_X}$, containing the encountered left-corner symbols.

$$\textbf{type}\ \mathit{LCTrafo}\ \mathit{env}\ a = \mathit{Trafo}\ (\mathit{MapA\_X}\ \mathit{env}\ a)\ \mathit{Productions}$$

Next we define the function *newNontR* which is a special version of the function *newSRef*, using $\mathit{MapA\_X}$ as internal state instead of *Unit*. It takes a left-corner symbol X as argument and yields a *LCTrafo* that introduces a new non-terminal A_X. The input of the *LCTrafo* is the first production (*Productions*) for A_X, and the output is the reference to this newly added non-terminal:

$$\mathit{newNontR} :: \forall x\ \mathit{env}\ s\ a\ .\ \mathit{Symbol}\ x\ \mathit{env}$$
$$\rightarrow \mathit{LCTrafo}\ \mathit{env}\ a\ s\ (\mathit{Productions}\ (x \rightarrow a)\ s)\ (\mathit{Ref}\ (x \rightarrow a)\ s)$$
$$\mathit{newNontR}\ x = \mathit{Trafo}\ \$\ \lambda m \rightarrow \mathit{extEnv}\ (\mathit{extendMap}\ x\ m)$$

The symbol X is added to the map of encountered left-corners of A by the function *extendMap*, which records the fact that the newly founded left-corner is the first element of the environment (*Zero*) and the previously added ones have to be shifted one place (*Suc*).

$$\mathit{extendMap} :: \mathit{Symbol}\ x\ \mathit{env} \rightarrow \mathit{MapA\_X}\ \mathit{env}\ a\ \mathit{env}'$$
$$\rightarrow \mathit{MapA\_X}\ \mathit{env}\ a\ (\mathit{env}', x \rightarrow a)$$
$$\mathit{extendMap}\ x\ (\mathit{MapA\_X}\ m) = \mathit{MapA\_X}\ (\lambda s \rightarrow \textbf{case}\ \mathit{matchSym}\ s\ x\ \textbf{of}$$
$$\mathit{Just}\ \mathit{Eq} \rightarrow \mathit{Just}\ \mathit{Zero}$$
$$\mathit{Nothing} \rightarrow \mathit{fmap}\ \mathit{Suc}\ (m\ s))$$

The index at which the new definition for A is stored is usually different from the index of A in the original grammar. This is a problem as we need to copy parts (the βs in the rules) of the original grammar into the new grammar. The non-terminal references in these parts must be adjusted to the new indexes. To achieve this we first collect all the new references for the non-terminals of the original grammar into a finite map, and then use this map to compute a *Ref*-transformer that is subsequently passed around and used to convert references from the original grammar to corresponding references in the new grammar. The type of this finite map is:

$$\textbf{newtype}\ \mathit{Mapping}\ o\ n = \mathit{Mapping}\ (\mathit{Env}\ \mathit{Ref}\ n\ o)$$

The mapping is represented as an *Env*, and contains for each non-terminal of the old grammar, the corresponding reference in the new grammar. The mapping can easily be converted into a *Ref*-transformer:

$$\mathit{map2trans} :: \mathit{Mapping}\ \mathit{env}\ s \rightarrow T\ \mathit{env}\ s$$
$$\mathit{map2trans}\ (\mathit{Mapping}\ \mathit{env}) = T\ (\lambda r \rightarrow (\mathit{lookupEnv}\ r\ \mathit{env}))$$

Now all that is left to do is to glue all the pieces defined above together. Each of the following functions corresponds to the untyped version with the same name. We start with the function *insert*:

```
insert :: ∀env s a x . Env Productions env env → Symbol x env
        → LCTrafo env a s (T env s, Prod (x → a) s) (Productions a s)
insert old_gram x =
   Trafo  (λ(MapA_X m) →
               case m x of
                  Just r    → extendA_X (MapA_X m) r
                  Nothing → insNewA_X (MapA_X m))
   where
      Trafo insNewA_X = proc (tenv_s, p) → do
         r ← newNontR x ≺ PS [p]
         rule2 old_gram x ≺ (tenv_s, r)
```

This function takes the original grammar and a left-corner symbol x as input. It yields a transformation that takes as input a *Ref*-transformer from the original to the new (transformed) grammar and a production for the non-terminal A_X, and stores this production in the transformed grammar. If the symbol x is new ($m\ x$ returns *Nothing*), the production is stored at a new index (using *newNontR*) and the function *rule2* is applied, to continue the depth-first search for left-corners. If we already know that x is a left-corner of a then we obtain an index r to the previously added to the non-terminal A_X, and add the new production at this position. The function *extendA_X* returns the *TrafoE* that performs this update into the environment:

```
extendA_X ::  m env1  → Ref (x → a) env1
                 → TrafoE m Productions s env1 (T env s, Prod (x → a) s)
                                                             (Productions a s)
extendA_X m r = fmap (const $ PS []) $ updateSRef m r addProd
                where addProd (_, p) (PS ps) = PS (p : ps)
```

If in the function *rule2* the left-corner is a terminal symbol then *rule2a* is applied, and the new production rule is returned as *Arrow*-output. In case the left-corner is a non-terminal the corresponding productions are looked up in the original grammar, and *rule2b* is applied to all of them, thus extending the grammar under construction:

```
rule2 :: Env Productions env env → Symbol x env
        → LCTrafo env a s (T env s, Ref (x → a) s) (Productions a s)
rule2 _ (Term a) = proc (_, a_x) → returnA ≺ PS [rule2a a a_x]
rule2 old_gram (Nont b) = case lookupEnv b old_gram of
   PS ps → proc (tenv_s, a_x) → do
                  pss ← sequenceA (map (rule2b old_gram) ps) ≺ (tenv_s, a_x)
                  returnA ≺ PS (concatMap unPS pss)
```

We now define the functions *rule2a*, and *rule2b* that implement the corresponding rules of the LCT. Firstly, *rule2a*, which does not introduce a new non-terminal, but simply provides new productions for the non-terminal (A) under consideration. The implementation of rule 2a is as follows:

```
rule2a :: String → Ref (String → a) s → Prod a s
rule2a a refA_a = Term a .*. Nont refA_a .*. End ($)
```

The function *rule2b* takes the original grammar and a production from the original grammar as arguments, and yields a transformation that takes as input a *Ref*-transformer and a reference for the non-terminal A_B, and constructs a new production which is subsequently inserted. Note that the *Ref*-transformer *tenv_s* is applied to the non-terminal references in *beta* to map them on the corresponding references in the new grammar.

```
rule2b :: Env Productions env env → Prod b env
       → LCTrafo env a s (T env s, Ref (b → a) s) (Productions a s)
rule2b old_gram (Seq x beta)
  = proc (tenv_s, a_b) →
    insert old_gram x ≺ (tenv_s
                        , append (flip (.)) (mapProd tenv_s beta) (Nont a_b)
                        )
```

The function *rule1* is almost identical to *rule2b*; the only difference is that it deals with direct left-corners and hence does not involve a "parent" non-terminal A_B.

```
rule1 :: Env Productions env env → Prod a env
      → LCTrafo env a s (T env s) (Productions a s)
rule1 old_gram (Seq x beta)
  = proc tenv_s →
    insert old_gram x ≺ (tenv_s, mapProd tenv_s beta)
```

The function *rules1* is defined by induction over the original grammar (i.e. it iterates over the non-terminals) with the second parameter as the induction parameter. It is polymorphically recursive: the type variable env' changes during induction, starting with the type of the original grammar (i.e. *env*) and ending with the type of the empty grammar (). The first argument is a copy of the original grammar which is needed for looking up the productions of the original non-terminals:

```
rules1 :: Env Productions env env → Env Productions env env'
       → Trafo Unit Productions s (T env s) (Mapping env' s)
rules1 _ Empty = proc _ → returnA ≺ Mapping Empty
rules1 old_gram (Ext ps (PS prods)) = proc tenv_s → do
        p ← initMap nt ≺ tenv_s
        r ← newSRef ≺ p
        Mapping e ← rules1 old_gram ps ≺ tenv_s
        returnA ≺ Mapping (Ext e r)
  where
    nt = proc tenv_s → do
             pss ← sequenceA (map (rule1 old_gram) prods) ≺ tenv_s
             returnA ≺ PS (concatMap unPS pss)
```

The result of *rules1* is the complete transformation represented as a value of type *Trafo*. At the top-level the transformation does not use any state, hence the type *Unit*. When dealing with one non-terminal (*nt*), *rule1* is applied for each of its productions and the new productions are collected to be inserted in the new grammar. The function *initMap* initialises the state information of the transformation *nt* with an empty table of encountered left-corners.

```
initMap :: LCTrafo env a s c d → Trafo Unit Productions s c d
initMap (Trafo st) = Trafo (λ_ → case st emptyMap of
                                   TrafoE _ f → TrafoE Unit f)
```

As input the transformation returned by *rules1* needs a *Ref*-transformer to remap non-terminals of the old grammar to the new grammar. During the transformation *rules1* inserts the new definitions for non-terminals of the original grammar, and remembers the new locations for these non-terminals in a *Mapping*. This *Mapping* can be converted into the required *Ref*-transformer, which must be fed-back as the *Arrow*-input. This feed-back loop is made in the function *leftcorner* using **mdo**-notation:

```
leftcorner :: ∀a . Grammar a → Grammar a
leftcorner (Grammar start productions)
    = case runTrafo lctrafo Unit ⊥ of
```

```
      Result _ (T tt) gram → Grammar (tt start) gram
  where
    lctrafo = proc _ → mdo
      let tenv_s = map2trans menv_s
      menv_s ← (rules1 productions productions) ≺ tenv_s
      returnA ≺ tenv_s
```

The resulting transformation is run using ⊥ as input; this is perfectly safe as it does not use the input at all: the result is a new start symbol and the transformed production rules, which are combined to form the new grammar.

5 Conclusions

We have shown how complicated transformations can be done at run-time, while having been partially verified statically by the type system. Doing so we have used a wide variety of type system concepts, like GADTs and existential and polymorphic types, which cannot be found together in other general purpose languages than Haskell. This allows us to use techniques which are typical of dependently typed systems while maintaining a complete separation between types and values. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

Implementing transformations like the left-corner transform implies the introduction of new references to a collection of possibly mutually recursive definitions. Previous work on typeful transformations of embedded DSLs represented as typed abstract syntax [3,2,4] does not deal with such complexity. Thus, as far as we know, this is the first description of run-time typed transformations which modify references into an abstract syntax represented as a graph instead of a tree.

We have shown how the untyped version of a transformation can be transformed into a typed version; after studying this example the implementation of similar transformations, using the TTTAS library, should be relatively straightforward. Despite the fact that this transformation is rather systematic, it remains a subject of future research to see how such transformations can be done automatically.

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*, New York, NY, USA, 2009. ACM.

[2] 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 Press.

[3] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM'03*, 2003.

[4] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in haskell. *Electron. Notes Theor. Comput. Sci.*, 174(7):23–39, 2007.

[5] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[6] M. Johnson. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL 98, Montreal, Quebec, Canada*, pages 619–623. Association for Computational Linguistics, 1998.

[7] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[8] 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.

[9] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

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

[11] Doaitse Swierstra and Luc Duponcheel. Deterministic, error correcting combinator parsers. In *Advanced Functional Programming, Second International Spring School*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[12] Marcos Viera, S. Doaitse Swierstra, 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.