# Typed Transformations of Typed Abstract Syntax

*Arthur Baars*

*Doaitse Swierstra*

Technical Report UU-CS-2008-021
July 2008

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

ISSN: 0924-3275

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

# Typed Transformations of Typed Abstract Syntax

Baars, Arthur and Swierstra S.Doaitse

August 8, 2008

**Abstract**

Advantages of embedded domain-specific languages (EDSLs) are that one does not have to implement a separate type system nor an abstraction mechanism, since these are directly borrowed from the host language. Straightforward implementations of embedded domain-specific languages map the semantics of the embedded language onto a function in the host language. The semantic mappings are usually compositional, i.e. they directly follow the syntax of the embedded language.

One of the questions which arises is whether conventional compilation techniques, such as global analysis and resulting transformations, can be applied in the context of EDSLs. The approach we take is that, instead of mapping the embedded language directly onto a function, we first build a representation of the abstract syntax tree of the embedded program fragment. This syntax tree is subsequently analyzed and transformed, and finally mapped onto a function representing its denotational semantics. In this way we achieve an online compilation of the embedded language.

We show how to use typed abstract syntax to represent fragments of embedded programs containing variables and binding structures while preserving the idea that the type system of the host language is used to emulate the type system of the embedded language.

The tricky issue is how to keep mutually recursive structures well-typed while being transformed. For this we develop an arrow-like library which assists in implementing such analyses and transformations and show its usefulness in describing left-recursion removal of an embedded grammar expressed with parser combinators using the Left-Corner Transform.

## Contents

# 1 Introduction

The use of combinator libraries for embedding a domain-specific language is common practice in Haskell programming. With such a library a programmer can use a domain-specific notation, and at the same time benefit from all the features, such as the type system and abstraction mechanisms, of the general purpose host language. The tight integration of the domain-specific language with the host language has great benefits: there is no need to extend the compiler or use ad hoc external tools which map the specific notation onto the host language.

One of the best known examples of embedded domain-specific languages is combinator parsing (Wadler 1985; Fokker 1995; Hutton and Meijer 1998; Swierstra and Duponcheel 1996; Leijen 2001; Swierstra 2001). The combinators provide an elegant way to define parsers in a notation that closely resembles BNF. For frequently occurring patterns (such as the EBNF extensions of BNF), a programmer can easily create more complex combinators out of simpler ones. The fact that parsers are just values makes it possible to compute new parsers during program execution (Baars et al. 2004), whereas lazy evaluation allows us to only construct that part of a very large grammar that is needed for the input at hand, something which is impossible to achieve with off-line generation techniques.

One of the essential aspects of combinator-based domain-specific embedded languages is that they share their type system with the host language. As a consequence, parsers are *typed values*, usually of some type *Parser a*, in which the *Parser* describes the basic parsing structure, and the parameter *a* is the type of the value that is returned by a parser as a witness of a successful parse.

Although such a combinator-based approach for implementing domain-specific languages looks very convincing at first sight, many problems show up when one starts to use this technique in practice. Just as conventional compilers may analyze programs extensively and may transform them based on the results of such

analyses, one also wants to use such techniques for combinator-based embedded languages. Why not generate a bottom-up parser on the fly or optimize a database query?

Unfortunately, such analyses and transformations are usually not possible, because the value being constructed by the combinators is a (possibly higher order) function (as in denotational semantics), and hence the internal structure can be neither inspected nor transformed. The approach we take is to have the combinators build a data structure which corresponds to its *typed abstract syntax tree following from the grammar of the embedded language*. This representation may be analyzed, transformed, and only later be mapped onto its semantics. The main complication we now face is that we have to do this *in a typed setting*. In contrast, conventional parser generators basically work on an untyped representation of a grammar, containing pieces of text which describe functions to be called when a reduction is made; whether such calls are type correct only becomes clear when we compile the generated program. We have to make sure that the code which assembles the parser from its various partial descriptions, exported from different modules, is type correct. Once the program performing the transformation is running no type checking can take place.

The problem becomes even harder when the embedded language not only borrows the type system and the abstractions provided by the host language, but also its declarative structure. Now we have to deal with several abstract syntax trees containing references to each other and with binding structures. We thus have to represent graph like structures, instead of trees.

In their work on typed meta-programming (Pasalic and Linger 2004), Pasilic and Linger, using the the encoding of equality types in Haskell (Baars and Swierstra 2002), show how to represent typed abstract syntax trees containing typed references to values. A group of mutually recursive bindings is represented by a nested cartesian product of terms, in which the variables are represented by typed pointers into this enviroment. A similar approach was developed slightly earlier by Xi and Chen, using a dependently typed version of ML, in their work on typeful program transformations (Chen and Xi 2003). The key ingredient for both is the encoding of environments and references pointing into these environments in host language terms. This encoding ensures that references always refer to *existing values* with the right types. In all approaches, references are basically indices into an environment, for the sake of demonstration encoded as Peano numbers.

In an earlier paper (Baars and Swierstra 2004), we extended Pasalic and Linger's encoding of typed abstract syntax to develop a typed implementation of on-line left-recursion removal from grammars described by parser combinators, using the textbook algorithm which unfortunately causes an exponential worst-case increase of grammar size. A grammar is represented as an environment containing productions for each nonterminal. These productions in turn contain references (nonterminals) to the production rules in the grammar. We also presented a way to alleviate the burden of encoding nonterminals (i.e. references) as Peano numbers, so one can use a notation similar to that of parsing combinators when writing the grammars. In this algorithm, a *fixed* number of

new nonterminals were introduced for "repetitions of zero or more occurrences" for the tails of the left-recursive alternatives. Hence we managed to avoid the dynamic introduction of new nonterminals by adding a special *Many*-constructor for repetitions to the term data type representing parsers, thus representing the possibly needed nonterminals beforehand. Thus we avoided the problem of dynamically extending the grammar with new nonterminals as need arises.

The main contribution of this paper is the development of an arrow-style library that handles introductions of new bindings, and thus can be used to implement a variety of typed program transformations. The use of this library is *not limited to grammar transformations* and can be used for many other typeful transformations in which new definitions are introduced. Such as common-subexpression elimination, and query optimization.

As a "real-world" use of the library we develop a left-recursion removal algorithm based on the Left-Corner (LC) transform. The type of the resulting transformation is:

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

where $a$ is the type associated with the start symbol of the grammar. This type is obvious and simple. Internally, however, we make heavy use of universally and existentially quantified types in combination with Generalized Algebraic Data Types (GADT). This paper therefore can also be seen as a non-trivial exercise using these type system extensions. Throughout the paper, we use GHC (GHC) syntax for these extensions to Haskell. The code in this paper can be found at: `http://www.cs.uu.nl/wiki/Center/TTTAS`, and was produced by running `lhs2TeX` on the source of this paper. The code is accepted by the Glasgow Haskell Compiler (GHC).

In (Viera et al. 2008) it is shown how the LC-transform itself can be used to derive an efficient version for the Haskell function *read*, by converting an exponential time algorithm into a linear one. The paper also shows how to use the library which is developed here for left-factoring, another efficiency-improving grammar transformation. The use of the techniques developed here are essential since it is only at runtime that all the information comes together. Work traditionally done by an off-line parser generator has been moved to program execution time, and thus grammars have become first-class typed objects, which are analysed, constructed and transformed at runtime.

This paper is organized as follows. In Section 2, summarizing earlier work (Pasalic and Linger 2004; Baars and Swierstra 2002, 2004), we discuss the encoding of typed abstract syntax trees, references and environments. These are the objects for our transformations. In Section 3, we develop the library that maintains a changing typed environment. It ensures that all typed references (references that "know" the type of the values they refer to) remain consistent whenever a new definition is added to the environment. in Section 4.1.1, we show a large case study: the implementation of the LC-transform. Finally, in Section 5, we present our conclusions.

## 2 Typed References and Environments

We start by shortly repeating the ideas behind typed meta-programming, in which we represent programs explicitly, so they can be manipulated. We want to do this in such a way that we keep the nice properties that typed programming languages have. Specifically, the fact that the presentation is type correct can be seen as a proof that the represented expression is type correct.

As an example consider the abstract syntax (formulated as a GADT) of a simple expression language:

```
data Exp a where
  IntVal   :: Int                          -> Exp Int
  BoolVal  :: Bool                         -> Exp Bool
  Add      :: Exp Int  -> Exp Int          -> Exp Int
  LessThan :: Exp Int  -> Exp Int          -> Exp Bool
  If       :: Exp Bool -> Exp a -> Exp a   -> Exp a
```

where the expression

**if** $3 + 1 < 4$ **then** $5$ **else** $1 + 2$

is represented by:

```
expr :: Exp Int
expr = If (LessThan (Add (IntVal 3) (IntVal 1)) (IntVal 4))
          (IntVal 5)
          (Add (IntVal 1) (IntVal 2))
```

The value of the represented expression has type *Int*, which is reflected in the type of *expr* :: *Expr Int*. Note that the the ill-typed expression $3 <$ *True* cannot be represented!

The question which arises now is how to represent variables and binding structures. We extend our simple expression language with a constructor *Var*, where a variable is represented by a reference of type *Ref a env*, an index pointing to a value of type *a* in an environment of type *env*. In the next section, we will go into detail on the type *Ref*.

The type *Expr* takes an extra type parameter *env*, which stands for the type of the environment in which the expression is to be evaluated, and to elements of which thus the variables encoded in the term may refer:

```
data Expr a env where
  Var      :: Ref a env                          -> Expr a env
  IntVal   :: Int                                -> Expr Int env
  BoolVal  :: Bool                               -> Expr Bool env
  Add      :: Expr Int env  -> Expr Int env      -> Expr Int env
  LessThan :: Expr Int env  -> Expr Int env      -> Expr Bool env
  If       :: Expr Bool env -> Expr a env
                            -> Expr a env        -> Expr a env
```

An evaluator *eval* for our simple expression language takes as arguments the abstract syntax tree of an expression, and an environment which provides values for the variables in the expression, and returns the value of the expression. The function *lookup* will be discussed later, and here only its type matters.

$$lookup :: Ref\ a\ env \to env \to a$$

$$\begin{aligned}
&eval :: Expr\ a\ env \to env \to a \\
&eval\ (Var\ r) \quad\ e = lookup\ r\ e \\
&eval\ (IntVal\ i) \quad _ = i \\
&eval\ (BoolVal\ b)\ _ = b \\
&eval\ (Add\ x\ y) \quad e = eval\ x\ e + eval\ y\ e \\
&\ldots
\end{aligned}$$

## 2.1 Type equality

Pasalic and Linger (Pasalic and Linger 2004) introduce an encoding of typed references that can be used for meta-programming. This encoding relies on the equality type (Baars and Swierstra 2002; Weirich 2000; Cheney and Hinze 2003). A (non-diverging) value of type $Equal\ a\ b$ is a witness of the proof that the types $a$ and $b$ are equal. This witness takes the form of a conversion function, which turns out to always be the identity function.

The addition of GADTs (Peyton Jones et al. 2006) to GHC makes programming with the *Equal* data type a lot easier, because all the fiddling with proofs is implicitly done by the compiler. Furthermore, the performance increases, since the construction of the proofs is no longer done at run-time. The compiler "knows" that all proofs of type equality are witnessed by values like $id$, $id\ id$, $id\ (id\ id)$, $id\ id\ id$ etc, and can thus omit them safely from the generated code: they have no other observable effect than taking time to execute.

In this section, we introduce previous work as part of our library (Baars and Swierstra 2004)). The only difference is that here we use GADTs to encode the type *Equal*. Furthermore, we show that by distinguishing between the types used in an environment and the types defined in an environment, we can manipulate both sets in relative isolation. Only when we are finished with our manipulation do we require them to be the same. The encoding of type equality is trivial when using GADTs:

$$\textbf{data}\ Equal :: * \to * \to * \ \textbf{where}\ Eq :: Equal\ a\ a$$

The type *Equal* has just one constructor $Eq :: Equal\ a\ a$. If a pattern match on a value of type $Equal\ a\ b$ succeeds (i.e., a non-$\perp$ value $Eq$ is available), then the type checker is thus informed that the types $a$ and $b$ were known to be the same at the place the $Eq$ was produced. The equality relation is reflexive, symmetric and transitive, and these properties are easily encoded:

$$\begin{aligned}
&reflex \quad\ :: Equal\ a\ a \\
&reflex \quad\ = Eq
\end{aligned}$$

```
symm        :: Equal a b → Equal b a
symm Eq     = Eq
trans       :: Equal a b → Equal b c → Equal a c
trans Eq Eq = Eq
```

If we know that two types $a$ and $b$ are equal, then we can safely cast an $a$ value into a $b$ value:

```
cast :: Equal a b → a → b
cast Eq = id
```

## 2.2 Typed References

In their paper on typed meta-programming, Pasalic and Linger introduced the following GADT for representing *typed* indices which are labelled with the type of value to which they refer and the type of environment (a nested cartesian product) in which this value lives.

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

In the case of a *Suc* we are not interested in the first element, so this constructor is polymorphic in the type $x$.

Two references can be compared for equality using the function *match*. If they refer to the same element in the environment this function returns the value *Just Eq*, thus expressing the fact that the types of the referred values are the same too:

```
match :: Ref a env → Ref b env → Maybe (Equal a b)
match Zero     Zero     = Just Eq
match (Suc x) (Suc y) = match x y
match _        _        = Nothing
```

The *lookup* function, the type of which we have seen before, uses its reference parameter as an index in the environment parameter. Whenever we decrease the index, we take the *snd* part of the tuple, until the index reaches *Zero*. The types guarantee that the lookup succeeds:

```
lookup :: Ref a env → env → a
lookup Zero     (a, _) = a
lookup (Suc r) (_, e) = lookup r e
```

The function *update* takes an additional function as argument, which is used to update the value the reference addresses. The other values in the environment are left unchanged:

$$
\begin{aligned}
&update :: (a \to a) \to Ref\ a\ env \to env \to env \\
&update\ f\ Zero\ \ \ \ (a, e) = (f\ a, e) \\
&update\ f\ (Suc\ r)\ (x, e) = (x, update\ f\ r\ e)
\end{aligned}
$$

For the sake of presentation, we have taken a simple type to represent environments, but more complicated structures like binary trees are also possible in case efficiency becomes a problem. Here, it would only add complexity to the presentation.

As an example, consider the *example* environment:

$$
\begin{aligned}
&\textbf{type}\ ExampleEnv = (Int, (Char, (String, ()))) \\
&example :: ExampleEnv \\
&example = (1, (\texttt{'a'}, (\texttt{"b"}, ()))) \\
&ref_a\ \ \ :: Ref\ Char\ ExampleEnv \\
&ref_a\ \ \ = Suc\ Zero \\
&ref_{one}\ :: Ref\ Int\ ExampleEnv \\
&ref_{one}\ = Zero
\end{aligned}
$$

Using our extended data type we can also encode expressions which contain variables such as **if** $b$ **then** 3 **else** $a$, using an environment that starts with an *Int* and a *Bool*:

$$
\begin{aligned}
&ref_a\ \ \ \ = Var\ Zero \\
&ref_b\ \ \ \ = Var\ (Suc\ Zero) \\
&testexpr :: Expr\ Int\ (Int, (Bool, env)) \\
&testexpr = If\ ref_b\ (IntVal\ 3)\ ref_a \\
&testenv\ \ :: (Int, (Bool, ())) \\
&testenv\ \ = (11, (False, ())) \\
&test\ \ \ \ :: Int \\
&test\ \ \ \ = eval\ testexpr\ testenv
\end{aligned}
$$

The expression *lookup* $ref_a$ *example* yields the character 'a', and *lookup* $ref_{one}$ *example* yields the integer 1. Notice that the type of the reference determines the type of the result! Application of *update* $(+5)$ $ref_{one}$ to the example environment updates it to $(6, (\texttt{'a'}, (\texttt{"b"}, ())))$. This clearly shows that the $ref_a$ and $ref_b$ address values of different types in the same environment.

Some may complain that this Peano representation is extremely cumbersome and error prone. In (Baars and Swierstra 2004), we have shown how, by using some extra combinators, this problem can be overcome. Furthermore the type system also helps us avoid accidental mistakes. Also, note that building the internal representation is the work of the combinator library and not so much of the user of the embedded language.

## 2.3 Mutually recursive terms

The question which arises now is how to represent a collection of possibly mutually recursive definitions, each consisting of an identifier being defined and a right-hand side expression containing such identifiers.

The idea is to store the right-hand side expressions in a heterogeneous list, and represent the identifiers by indices in this list. This is very similar to the environments described above, with the main difference that the actual environment now contains typed terms instead of typed values. The type $a$ in a reference of type *Ref a...* is the same as the type parameter $a$ in the type of the term *Expr a env* it addresses. But what to choose now for the *env*-parameter in the references occurring in the terms? Because these references point to other terms in the collection of definitions, the type *env* should be the type of the nested tuple itself, giving rise to an infinite type for the nested tuple, which is not allowed in Haskell.

Our solution is found in splitting the second type parameter in two parameters: one indicating the environment addressed by the references occurring in the terms and one environment which is being constructed by the sequence of terms. The type *Env t use def* represents a sequence of instantiations of type $\forall a\ .\ t\ a\ use$, where all the instances of $a$ are stored in the type parameter *def*; thus the type *def* contains the type parameters $a$ of the terms of type $t\ a\ use$ occurring in the *Env t use def*. The type *use* on the other hand contains the types that may be referenced from within terms of type $t\ a\ use$.

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

When the types *def* and *use* coincide we can be sure that the references in the terms do not point to values outside the environment and to terms representing the right type. Splitting this single type into two type parameters, which we only require in the end to be equal, makes it possible to both add new terms to the environment which are not yet taken into account by the existing refs and to use references which refer to terms which still have to be added. Only after we are done with manipulating and extending the environment do we require them to be the same! The fact that a sequence of mutually recursive terms is closed and well-typed is thus encoded in the type system of the host language. So the mutually recursive declarations:

$$(y, x) = (5 + x, y)$$

can be encoded as:

```
type FinalEnv t usedef = Env t usedef usedef
x    :: Expr Int (Int, (Int, ()))
x    = Var Zero
y    :: Expr Int (Int, (Int, ()))
y    = Var (Suc Zero)
rhss :: FinalEnv Expr (Int, (Int, ()))
rhss = Cons (Add (IntVal 5) x) (Cons y Empty)
```

where we note that the $x$ and $y$ here are Haskell values referring to the right-hand side terms of their definitions in the *Env*.

The lookup and update operations are defined in a similar way as before:

$$
\begin{array}{l}
lookupEnv :: Ref\ a\ env \to Env\ t\ s\ env \to t\ a\ s \\
lookupEnv\ Zero\ \ \ (Cons\ p\ \_)\ \ = p \\
lookupEnv\ (Suc\ r)\ (Cons\ \_\ ps) = lookupEnv\ r\ ps \\
updateEnv\ :: (t\ a\ s \to t\ a\ s) \ \ \to Ref\ a\ env \to Env\ t\ s\ env \to Env\ t\ s\ env \\
updateEnv\ f\ Zero\ \ \ (Cons\ \ \ \ ta\ rs) = Cons\ (f\ ta)\ rs \\
updateEnv\ f\ (Suc\ r)\ (Cons\ \ \ \ x\ \ rs) = Cons\ x\ \ \ \ (updateEnv\ f\ r\ rs)
\end{array}
$$

The chosen representation now has an efficiency problem, to be fixed in the next section: whenever we extend the environment with a new *Cons* all existing references occurring in terms already stored in the environment have to be incremented by applying an extra *Suc* constructor to them, since the values to which they refer have an index that is one higher in the new environment.

# 3 Transformation Library

In this section we develop a type *Trafo* representing typed transformation steps on a heterogeneous collection and an *Arrow*-like library of combinators for composing such transformations. Each *Trafo* takes input and produces output and can be composed in the same way as *Arrow*s. Additionally, meta-data about the transformation process can be maintained too. Such meta-data could for example be a symbol table, debugging information, reference counts, etc.

In developing the type *Trafo* we use a Haskell-like type synonym syntax augmented with the symbols $\forall$ and $\exists$ to denote universally and existentially quantified types. We first develop the type *Trafo* to tackle the problem of maintaining a heterogeneous collection of definitions, and subsequently extend it with *Arrow*-style inputs, outputs, and meta-data. Finally we encode the the *Trafo* using data types, as accepted by GHC.

We model a collection of embedded-language definitions as a value of type *Env*, and make these definitions the subject of transformations that may induce new definitions, as in the case of common-subexpression removal, where a subexpression gets named. At the end of the transformation process each reference in this *Env* must be a reference into the final set of definitions. In this case, we call the environment *closed*. One way to ensure that our environment is always closed is to adjust all the references in all the terms whenever a new definition is added to the environment. This is cumbersome and inefficient, and is better done once, i.e., when we know how many *Suc*s to add to each reference to make it address the right element in the final structure.

We only require an environment to be closed after all transformations have been applied and all new definitions have been added. The final type must be of the form $Env\ t\ s\ s$ (or $FinalEnv\ t\ s$) for some type $s$. References into this environment $s$ are coined *final references*. If all the transformation steps only add terms of type $(\exists\, a\ .\ t\ a\ s)$ to the environment, then they contain only final references, and we do not need to adjust the references after each

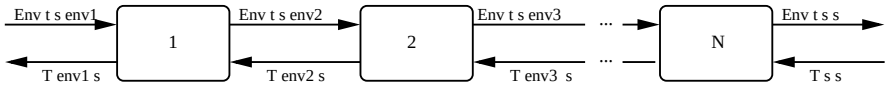


Figure 1:

transformation step. However, this seems to be impossible. How can we make the transformation steps half way through the transformation process construct terms of the type $(\exists\, a\ .\ t\ \ a\ \ s)$? Remember that $s$ is the type of the final environment and is only known after all transformation steps have completed? For creating such final references we need to know how many new definitions will be added to the front of the environment by all future transformation steps.

Lazy evaluation comes to the rescue. We solve this problem by passing knowledge about the "future" backwards through the computation. In our case information about the number of definitions added by future steps is encoded in a *Ref*-transformer, that prepends as many *Suc*-nodes to a reference as there are new definitions to come. The type of such *Ref*-transformers is[1]:

$$\textbf{newtype}\ T\ e\ s = T\{\, unT :: \forall x\ .\ Ref\ x\ e \rightarrow Ref\ x\ s\,\}$$

Figure 1 depicts the idea described above. The environment is constructed from left to right. Each step takes as input the environment constructed thus far and yields an updated environment as result. On the left, the computation starts with an environment of some type *Env t s env1*. Each step may extend the environment with some new definitions, making the type of the environment change at every step. The final result of all steps has to be an environment of type *FinalEnv t s*. *Ref*-transformers are passed on and modified from right to left. This transformer effectively tells every step how deep down the environment constructed thus far is located in the final environment.

The environment yielded by the last step is the place where the *use* and the *def* types have to coincide. Therefore the identity transformer is used as the initial value for the "pass-back" chain. Every step updates the transformer according to the number of definition it adds to the environment, before passing it on to preceding transformation steps.

## 3.1 The *Trafo* data type

We now develop the type *Trafo* to implement the idea described above. Every step has two incoming and two outgoing arrows, one of each type at each side. This means our *Trafo*-type is a function taking two arguments and returning two results. We want our *Trafo*-type to be polymorphic in the type of the terms $(t)$ stored in the environment. As a first attempt, we take:

$$\textbf{type}\ Trafo\ t = T\ env2\ s \rightarrow Env\ t\ s\ env1 \rightarrow (T\ env1\ s, Env\ t\ s\ env2)$$

[1]Note that the keyword **forall** is presented by the logical symbol $\forall$

the role of the different elements is as follows:

*Env t s env1* is the environment which has been constructed up to where the current transformation starts, and corresponds to the incoming arrow at the top left. The *env1* parameter describes which elements have thus far been added to the environment.

*T env2 s* is the incoming arrow at the bottom right. It maps references into an enviroment labelled with *env2* into references into the final environment *s*.

*Env t s env2* is the newly constructed environment, in which *env2* will usually be an extension of *env1*.

*T env1 s* corresponding to the bottom left arrow coming out of a *Trafo*, is the updated *T env2 s*. It can be constructed by this transformation since it knows how many elements were added to the environment.

This type definition is incomplete, the variables *env1*, *env2*, and *s* are still unbound. We do not want these variables to appear on the left-hand side of the type definition, as this would expose the internal complexity of the library to the user, so we have to add universal or existential quantification.

The type *env1* is the type of the environment constructed thus-far. A step should not make any assumptions about this environment, hence the type variable *env1* is universally quantified. The type *env2* is the type of the result of a transformation step. This type depends on the number of new definitions introduced by the step. As this can be an arbitrary number, the type *env2* is fully determined by the transformation and the incoming *env1*. Because *env2* depends on *env1* by extending it, this quantifier has to be within the scope of *env1*: once *env1* is fixed the transformation fixes *env2*. Finally, the type variable *s* represents the type of the final result. This type is only known when we are done with all transformation steps, and hence computations in a *Trafo* cannot make any assumptions about it, and thus it must be universally quantified. All this leads to the following definition for the type *Trafo*:

$$\begin{aligned}&\textbf{type } \mathit{Trafo}\ t = \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\ \forall s\ .\\ &\qquad\qquad T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \to (T\ \mathit{env1}\ s, \mathit{Env}\ t\ s\ \mathit{env2})\end{aligned}$$

In the next step, we extend the type *Trafo* with *Arrow*-style input and output. This allows us to pass values from one transformation *Trafo* to the next. The types of the input and output are also labeled with the type of the final environment, because we want to be able to communicate typed references and terms between transformation steps. So we add two arguments ($a$ and $b$) to the type *Trafo*, which stand for the types of the input and output:

$$\begin{aligned}&\textbf{type } \mathit{Trafo}\ t\ a\ b =\\ &\quad \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\ \forall s\ .\ a\ s \to T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1}\\ &\qquad\qquad\qquad \to (b\ s, T\ \mathit{env1}\ s, \mathit{Env}\ t\ s\ \mathit{env2})\end{aligned}$$

Finally we may want to maintain meta-information about the environment, such as which elements have been added already. This information may be used to determine what kind of new elements have to be added to the environment and hence has to live outside the type which is existentially quantified by *env2*.

Thus, we introduce an extra argument $m$ that stands for the type of the meta-data. A *Trafo* takes the meta-data on the current environment *env1* as input and yields meta-data for the (possibly extended) environment *env2*.

```
type Trafo m t a b =
  ∀env1 . m env1 → ∃env2 .
    (m env2, ∀s .  a s → T env2 s → Env t s env1 →
                   (b s, T env1 s, Env t s env2)
    )
```

Since the meta-information should not depend on the type of the final environment $s$ the value of type *m env2* is kept outside the scope of the $s$. Furthermore the incoming meta-data of type *m env1* is outside the scope of the existential quantifier for *env2*. This allows us to use the meta-data to decide for example whether a new definition must be added to the environment or not. Placing this meta-information inside the existential quantifier makes this impossible.

We now have come to a problematic point: the type above is not Haskell, nor is it accepted by the GHC due to the use of existential quantifiers in type definitions. Instead, an existential type can only be introduced by using the keyword ∀ on the left side of a constructor in a data-declaration. Thus, we have to resort to an encoding of the above type as:

```
data Trafo    m t a b = Trafo (∀env1 . m env1 → TrafoE m t a b env1)
data TrafoE m t a b env1 =
      ∀env2 . TrafoE (m env2)
                     (∀s . a s → T env2 s → Env t s env1 →
                           (b s, T env1 s       , Env t s env2)
                     )
```

Now that we have developed the final version of our *Trafo* data type, we can define the combinators to construct and compose transformation steps.

## 3.2 *newSRef* and *runTrafo*

The most important operation is creating new references. This is implemented by *newSRef*, which takes a typed term as input, stores it in the environment and yields a reference pointing to this value. The type of *newSRef* is:

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

No meta-information on the environment is recorded by *newSRef*; therefore we use the type *Unit* for the meta-data. If meta-information is required, one

must define an application-specific version of *newSRef*, which we will do in our example application. The type variable $t$ stands for the type of the terms. We want the input to *newSRef* to be of type $t\ a\ s$, where $s$ stands for the as-yet-unknown type of the final environment. The result of a *newSRef* is a reference of type $Ref\ a\ s$, which points to the newly inserted, $a$ labelled, term in the final environment. It may be worth reminding the reader at this point that $Trafo\ Unit\ t\ (t\ a)\ (Ref\ a)$ resembles an arrow from $t\ a$ to $Ref\ a$. Note that we cannot use normal *Arrow*s because, here, type arguments have kind $(\ *\ \rightarrow\ *\ )$.

$$
\begin{array}{l}
newSRef = Trafo \\
\quad (\lambda\_ \rightarrow TrafoE\ Unit \\
\qquad\qquad (\lambda ta\ (T\ tr)\ env \rightarrow (tr\ Zero, T\ (tr\ .\ Suc) \\
\qquad\qquad\qquad\qquad\qquad\qquad , Cons\ ta\ env \\
\qquad\qquad\qquad\qquad\qquad\qquad ) \\
\qquad\qquad ) \\
\quad )
\end{array}
$$

The incoming meta-information is ignored and *Unit* is returned as meta-data. The function in *TrafoE* is more interesting: it takes as arguments the term to be inserted $ta :: t\ a\ s$, the current environment $env :: Env\ t\ s\ e$, and a reference transformer $(T\ tr) :: T\ e\ s$, which transforms references into the current environment into references into the final environment. The result is a tuple containing the new environment, which has type $Env\ t\ s\ (a, e)$, and a reference of type $Ref\ s\ a$. The term $(ta)$ becomes the first element of the new environment, hence the reference pointing to this term is *Zero*. However, more terms may be added in the future, therefore the reference transformer $(tr)$ is applied, which basically prepends to *Zero* as many *Suc*-nodes as there are future additions to the environment under construction. Finally, we record the fact that one new element was added to the environment by adding an extra *Suc*-node to the reference transformer $tr$, which we pass on to the our predecessors.

Of course we want to "run" our *Trafo*-computations. This is the task of the function *runTrafo*:

$$runTrafo :: Trafo\ m\ t\ a\ b \rightarrow m\ () \rightarrow (\forall s\ .\ a\ s) \rightarrow Result\ m\ t\ b$$

The function *runTrafo* takes as arguments the *Trafo*s we want to run, meta-information for the empty environment, and an input value. The last argument needs to be polymorphic in $s$, since we cannot make any assumptions about the type of the final environment. The result of *runTrafo* is the output value $(b\ s)$, together with the final environment $(Env\ (t\ s)\ s)$ and meta-data $(m\ s)$. Because $s$ could be anything we have to hide it using existential quantification, and thus introduce the data definition *Result*.

$$\textbf{data}\ Result\ m\ t\ b = \forall s\ .\ Result\ (m\ s)\ (b\ s)\ (FinalEnv\ t\ s)$$

The function *runTrafo* now reads:

```
runTrafo (Trafo st) m a
  = case st m of TrafoE m' f → let (b, _, e) = f a (T id) Empty
                               in Result m' b e
```

Consider the type of the function $f$ which is the second component of the *TrafoE*-constructor:

```
f :: ∀s . a s → T env s → Env t s () →
           (b s,    T ()   s,    Env t s env)
```

where $a$, $t$, and $b$ are the input, term, and output types from the type signature of *runTrafo*, and *env* stands for the existential type of the resulting environment. The resulting environment is the final environment; therefore, the reference transformer that converts between references for the result environment and the final environment, is chosen to be the identity transformer ($T$ *id*), hence instantiating the polymorphic $s$ to *env* for the call to $f$!

### 3.3 Arrow-style combinators

Unfortunately the data type *Trafo* is not really an *Arrow*, because the type variables $a$ and $b$ are of kind $* \rightarrow *$ instead of $*$ ; keep in mind that they are parameterised by the final environment $s$. To implement the *Arrow*-like interface one needs to implement at least *arr*, >>>, and *first*. The types of the real *Arrow*-combinators are almost identical to the ones we define here. The only difference is that all input and output parameters are lifted.

The *arr* combinator lifts a function.

```
arr :: (∀s . a s → b s) → Trafo m t a b
arr f = Trafo (λm → TrafoE m (λa t e → (f a, t, e)))
```

The >>> combinator composes two *Trafo*s. It is actually a straightforward transcription of the composition depicted in Figure 1. In that figure box 1 refers to the incoming environment, box 2 to the intermediate and box 3 to the outgoing.

```
(>>>) :: Trafo m t a b → Trafo m t b c → Trafo m t a c
(>>>) (Trafo sa) (Trafo sb) =
  Trafo (λm1 → case sa m1 of { TrafoE m2 f1 →
                case sb m2 of { TrafoE m3 f2 →
                TrafoE m3 (λa t3s e1 → let (b, t1s, e2) = f1 a t2s e1
                                           (c, t2s, e3) = f2 b t3s e2
                                       in  (c, t1s, e3))}})
```

The function *first* from the real *Arrow*-class has the following type:

```
first :: arrow a b → arrow (a, c) (b, c)
```

Our version of *first* works on lifted tuples

```
newtype Tuple a b s = TP (a s, b s)
```

```
first :: Trafo m t a b → Trafo m t (Tuple a c) (Tuple b c)
first (Trafo s)
   = Trafo (λm1 → case s m1 of
                     TrafoE m2 f →
                       TrafoE m2
                              (λ(TP (a, c)) t2s e1
                                    → let (b, t12, e2) = f a t2s e1
                                      in (TP (b, c), t12, e2)
                              )
           )
```

For the sake of completeness, we also show the other functions of the *Arrow*-interface. The code is just the default definition found in the *Arrow*-class. The only difference is in the types.

```
second :: Trafo m t b c → Trafo m t (Tuple d b) (Tuple d c)
second f = arr swap >>> first f >>> arr swap
  where   swap∼(TP (x, y)) = TP (y, x)


(***) ::   Trafo m t b c → Trafo m t b' c'
       →   Trafo m t (Tuple b b') (Tuple c c')
f *** g = first f >>> second g
(&&&) ::   Trafo m t b c → Trafo m t b c' → Trafo m t b (Tuple c c')
f &&& g = arr (λb → TP (b, b)) >>> (f *** g)
```

The function *loop* is used to construct feedback loops. It takes a *Trafo* that has an input of type *Tuple a x* and output of type *Tuple b x*. The component of type $x$ is fed back resulting in a *Trafo* with input $a$ and output $b$.

```
loop :: Trafo m t (Tuple a x) (Tuple b x) → Trafo m t a b
loop (Trafo st) =
    Trafo
    (λm → case st m of
              TrafoE m1 f1 →
                TrafoE m1
                       (λa t e →
                         let (TP (b, x), t1, e1) = f1 (TP (a, x)) t e
                         in (b, t1, e1)
                         )
    )
```

The combinator *sequenceA* sequentially composes a list of *Trafo*s into a *Trafo* that yields a list of outputs. Its use is analogous to the *sequence* combinator for *Monad*s.

```
newtype List a s = List [a s]
sequenceA :: [Trafo m t a b] → Trafo m t a (List b)
```

```
sequenceA [] = arr (const (List []))
sequenceA (x : xs) = (x &&& sequenceA xs) >>>
                     arr (λ(TP (a, List as)) → List (a : as))
```

Although the combinators above do not define a real *Arrow*, programming with them is the same as programming with *Arrow*s. Unfortunately, one cannot use the special *Arrow* syntax (Paterson 2001).

# 4 Left-Corner transform

## 4.1 Motivation

We now come to second part of this paper in which we show how to use the library in transforming a grammar with the Left-Corner Transform (LCT) which removes left-recursion (Johnson 1998). LTC has complexity $O(n^2)$ where $n$ is the number of symbols (terminals as well as nonterminals).

Here we will use the version developed by Robert C. Moore (Moore 2000). His tests, using several large grammars for natural language processing, show that the algorithm performs very well in practice. Transformations like the LCT usually *introduce many new nonterminals to the grammar*. In our setting, a transformation must be type preserving and thus ensure that the types of the environment and the references remain consistent *while being modified*. Previous work on typeful program transformations (Chen and Xi 2003; Pasalic and Linger 2004; Baars and Swierstra 2004) does not deal with such an introduction of new definitions and binders, which complicates matters considerably.

We want to stress once more that our transformation is done at run-time and not off-line. It is to be used in situations where, while running the program, several parts of a grammar are coming together, and an efficient parser has to be constructed on the fly.

We provide three situations where this may be useful. The first one comes from the definition of the class *Read* in Haskell. Suppose we have the following situation:

```
module A (A_type) where
data A_type = S
            | A_type :+: A_type
            deriving Read

module B where
import A
data B_type a = S
              | a          :+: Int
              | B_type a :*: Int
              deriving Read
type Problematic = B_type A_type
```

Here we see that the first module exports a derived instance of the parsing function *read*, which is used to parameterize the function *read* generated from the

deriving clause for the *B_type*. Unfortunately, since we are exporting functions, there is no way in which we can discover that the second and the third alternative of *B_type* have possibly long common prefixes giving rise to exponential parsing times, which we should like to factor out in the parser. Recent work by Viera et al.(Viera et al. 2008) uses the techniques developed in this paper do just that. They export the description of the type *A_type* and subsequently analyze the composite structure, then transform it using the LCT, and finish by factoring out common prefixes (a kind of common subexpression removal for grammars). Subsequently, an efficient parser is generated at run-time. There is no way we can do so with any of the existing off-line parser generators, other than by exporting the joined description into a file, calling an external program to generate code, calling the GHC on this code, and dynamically linking this code in.

All combinator-based parsers rely on an underlying top-down parsing technique (be careful, we did not say parsers generated off-line from combinator-based grammar descriptions), and hence cannot handle left-recursive grammars. Even worse is the fact that it is not detected whether the underlying grammar is left-recursive, other than by getting into an endless loop at runtime or getting an error message from the runtime system such as a stack overflow. Furthermore the underlying backtracking mechanism can make parsers very slow, especially if a nonterminal has alternatives with a common prefix. A naive user, i.e. one who thinks that he is really writing grammars instead of parsers and directly transcribes his grammar into combinator-based code, is likely to be disappointed about the behavior of the parsers thus constructed. Making effective use of parser combinators requires the user to first apply transformations like left-factorization and left-recursion removal to the grammar. Of course, one can apply the transformations by hand; unfortunately, the resulting parsers look very involved and are hard to maintain. Changing the associated semantic actions is only for the brave-hearted, and we have seen many situations where the result cannot be described as anything but a mess. So the question arises whether we can get the benefits that are usually associated with real parser generators – that base their actions on a global grammar analysis – while keeping the things we like so much about combinator parsers such as their type system and the abstraction mechanisms they provide.

### 4.1.1 Algorithm: The Leftcorner-Transform

In this subsection we introduce the Left-Corner Transform (LCT) (Johnson 1998) as a set of rewrite rules and subsequently give an untyped implementation in Haskell98. We assume that our grammar has already been transformed (again using our library) such that only the start symbol may derive $\epsilon$.

We say that a symbol $X$ is a *direct left-corner* of a nonterminal $A$ if there exists a production for $A$ with $X$ as the left-most symbol on the right-hand side. We define the *left-corner* relation as the transitive closure of the direct left-corner relation. Note that a nonterminal being left-recursive is equivalent to being a left-corner of itself.

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

For each nonterminal $A$ of the original grammar, the algorithm applies the following rules for building a set of new productions for $A$, and productions for *new nonterminals* $A\_X$ for those $X$s which show up as a left-corner of $A$:

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 $A$.
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 nonterminal $B$ 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 $A$.

Here a new nonterminal like $A\_B$ represents that part of an $A$ after a $B$ has been recognized. As an example, consider the grammar:

$C \rightarrow a\ C \mid D$
$D \rightarrow C\ b \mid c$

Applying rule 1 on the productions of $C$ gives us two new production rules and two newly discovered left-corners:

$C\_a \rightarrow C$ $\qquad\qquad$ $\mathit{leftcorners} = [a, D]$
$C\_D \rightarrow \epsilon$

Applying rule 2a on $a$ yields:

$C \rightarrow a\ C\_a$ $\qquad\qquad$ $\mathit{leftcorners} = [\cancel{a}, D]$

Applying rule 2b on $D$ yields:

$C\_C \rightarrow b\ C\_D$ $\qquad\qquad$ $\mathit{leftcorners} = [\cancel{a}, \cancel{D}, C, c]$
$C\_c \rightarrow C\_D$

Applying rule 2b on $A$ yields two new productions, but no new left-corners:

$C\_a \rightarrow C\ C\_C$ $\qquad\qquad$ $\mathit{leftcorners} = [\cancel{a}, \cancel{D}, \cancel{C}, c]$
$C\_D \rightarrow C\_C$

And applying rule 2a on $c$ gives us:

$$C \rightarrow c\ C\_c \qquad\qquad leftcorners = [\sout{a}, \sout{D}, \sout{C}, \sout{c}]$$

Now that all left-corners of $C$ have been processed, we are done for $C$. For the nonterminal $D$, the process is repeated, yielding the following new productions: Applying these rules gives:

$$\begin{array}{lll}
D_C \rightarrow b & \text{-- rule 1} \\
D_c \rightarrow \epsilon & \text{-- rule 1} \\
D_a \rightarrow C\ B_C & \text{-- rule 2b, } C \\
D_D \rightarrow D_C & \text{-- rule 2b, } C \\
D \rightarrow c\ D_c & \text{-- rule 2a, } c \\
D \rightarrow a\ D_a & \text{-- rule 2a, } a \\
D_C \rightarrow b\ D_D & \text{-- rule 2b, } D \\
D_c \rightarrow D_D & \text{-- rule 2b, } D
\end{array}$$

Note that, by construction, this new grammar is not left-recursive, but we may still want to group alternatives belonging to the same nonterminal together and perform left-factorization.

## 4.2 Untyped LC-Transformation

Before presenting our typed LC-transformation, we first develop an untyped implementation: We start with a representation for our grammars:

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

A *Grammar* is a mapping from nonterminal names to their corresponding productions. All productions for the same nonterminal are grouped together. A production (*Prod*) is a sequence of symbols (*Symbol*). Our example grammar is encoded as:

```
grammar = Map.fromList [("C", [["a", "C"], ["D"]])
                       ,("D", [["C", "b"], ["c"]])]
```

During the transformation process, we maintain (using the *State*-monad from *Control.Monad.State*) the new grammar under construction. For each nonterminal, we traverse the left-corner graph, as induced by the productions, in depth-first order and store a list of encountered left-corner symbols.

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

```
put       :: Step_State → Trafo ()
get       ::              Trafo Step_State
modify    :: (Step_State → Step_State) → Trafo ()
runState :: Trafo a → Step_State → (a, Step_State)
```

We start with presenting the function *leftcorner* which takes a grammar and returns a transformed grammar. The actual work is delegated to the function *rules1* which yields a value of type *Trafo*. The state is initialized with an empty grammar and an empty list of discovered left-corners. The final state contains the newly constructed grammar which is returned as result:

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

For each (represented by *mapM_*) nonterminal ($A$) the function *rules1* visits (represented by *mapM*) its productions, each visit resulting in new productions by rules *rule2a* and *rule2b*, which are added to the transformed grammar by the function *insert*. The productions resulting from *rule2a* are returned (*ps*) and together (*concat*) become the new productions for the original nonterminal $A$. The list of discovered left-corners is reset when starting with the next nonterminal:

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

We continue by defining three functions that correspond to the rules in the transformation. The function *rule2b* generates new productions for the nonterminals of the original grammar, and *rule1* and *rule2b* generate productions for nonterminals 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 nonterminal $A\_X$ to the grammar. If we have met $A\_X$ before its entry is extended, otherwise an entry for $A\_X$ is added and 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. In case it is a nonterminal, its corresponding productions are looked up in the original grammar and *rule2b* is applied to each of these productions.

```
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 all left-corner symbols are discovered. The structure of the typed implementation we present in section 4.4 closely resembles that of the untyped solution above. Before we can develop the typed solution we need a representation for typed grammars.

## 4.3 Typed Grammar Representations

A grammar consists of a start symbol, represented as a reference labeled with the type of the witness value of a complete successful parse, and an *Env*, containing for each nonterminal its list of alternative productions. The actual type *env* is hidden using existential quantification.

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

Since in our LC transform 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 (Swierstra and Duponcheel 1996), 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 $f$ can be a function that accepts the results of the earlier elements as

argument. 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 nonterminal (reference).

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

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

We introduce some extra convenience operators for constructing grammars:

```
infixr 5`cons`, .*.
cons prods g = Cons (PS prods) g
(.*.) = Seq
```

We now have the machinery at hand to encode our example grammar in typed abstract syntax form:

```
_C = Nont       Zero
_D = Nont (Suc Zero)
_a =  Term "a"
_b =  Term "b"
_c =  Term "c"
```

Assume we want the witness type for nonterminal $C$ to be a *String* and for nonterminal $D$ an *Int*:

```
type NT_Types = (String, (Int, ()))
```

```
grammar :: Grammar String
grammar = Grammar Zero productions
  -- C ::= aC | D
  -- D ::= Cb | c
productions :: Env Prods NT_Types NT_Types
productions
  = [_a      .*. _C .*. End (++)
    ,_D              .*. End show
    ]`cons`
    [_C      .*. _b  .*. End (λy x → length x + length y)
    ,_c              .*. End (const 1)
    ]`cons` Empty
```

Before delving into the LC transform itself, we introduce some grammar-related functions we will need. The function *append* is used in the LC transform to

build productions of the form $\beta\ X\_A$. Basically, it corresponds to the *snoc* operation on lists. We only have to take care that all the types match.

```
append :: (a → b → c) → Prod a env → Symbol b env → Prod c env
append g (End f  ) s = Seq s (End (g f))
append g (Seq t ts) s = Seq t (append (λb c d → g (b d) c) ts s)
```

The function *matchSym* compares two symbols:

```
matchSym :: Symbol a env → Symbol b env → Maybe (Equal a b)
matchSym (Nont x) (Nont y)          = match x y
matchSym (Term x) (Term y) | x ≡ y = Just Eq
matchSym _        _                 = Nothing
```

The function *mapProd* systematically changes all the references to nonterminals occurring in a production:

```
mapProd :: T env1 env2 → Prod a env1 → Prod a env2
mapProd t (End x)          = End x
mapProd t (Seq (Nont x) r) = Seq (Nont (unT t x)) (mapProd t r)
mapProd t (Seq (Term x) r) = Seq (Term x)         (mapProd t r)
```

## 4.4 Typed LC-transformation

The Left-Corner transform is applied in turn to each nonterminal ($A$) of the original grammar. The algorithm performs a depth first search for left-corner symbols. For each left-corner $X$, a new nonterminal $A\_X$ 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 nonterminals. In the typed solution, nonterminals are, however, represented as typed references. The first time a production for a nonterminal $A\_X$ is generated, we must create a new entry for this nonterminal 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 discovered left-corner symbols ($X$) to references corresponding to the nonterminals ($A\_X$). This finite map plays the same role as the list of already discovered left-corner symbols in the untyped implementation:

```
newtype MapA_X env a env2 =
  MapA_X (∀x . Symbol x env → Maybe (Ref (x → a) env2))
```

The type variable *env* is the type of the original grammar, and *env2* is the type of the grammar constructed thus far. The type variable $a$ is the type of the current nonterminal. A left-corner symbol of type $x$ is mapped to a reference in the grammar under construction if it exists. The type associated with the nonterminal $A\_X$ is $(x \to a)$, 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:

```
emptyA_X :: MapA_X env a env2
emptyA_X = MapA_X (const Nothing)
```

The function *initA_X* initialises the meta-information with an empty table of discovered left-corners.

```
initA_X :: Trafo (MapA_X env a) t c d → Trafo Unit t c d
initA_X (Trafo st) = Trafo (λ_ → case st emptyA_X of
                                   TrafoE _ f → TrafoE Unit f
                           )
```

Next, we define the function *newNont* which is a special version of the function *newSRef* and which corresponds to the else-part of the function *insert* in the untyped version. It takes a left-corner symbol $X$ as argument and yields a *Trafo* that introduces a new nonterminal $A\_X$. The input of the *Trafo* is the first production (*Prods*) for $A\_X$, and the output is the reference to this newly added nonterminal. Internally, the symbol $X$ is added to the map of discovered left-corners of $A$.

```
newNont :: ∀x env t a .
           Symbol x env  → Trafo    (MapA_X env a) Prods
                                    (Prods (x → a))  (Ref (x → a))
newNont x =
  Trafo (λ(MapA_X m :: MapA_X env a env') →
           let m2 :: MapA_X env a (x → a, env')
               m2 = MapA_X (λs → case matchSym s x of
                                   Just Eq → Just Zero
                                   Nothing → fmap Suc (m s)
                           )
           in TrafoE m2 (λtx (T t) e → (t Zero, T (t . Suc), Cons tx e))
    )
```

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 $\beta$s in the rules) of the original grammar into the new grammar. The nonterminal references in these parts must be adjusted to the new indices. To achieve this, we first collect all the new references for the nonterminals of the original grammar into a finite map, and 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:

```
newtype Mapping o n = Mapping (Env Ref n o)
```

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

```
map2trans :: Mapping env s → T env s
map2trans (Mapping env) = T (λr → (lookupEnv r env))
```

We introduce the type-synonym *LCStep* which is the type of the transformation step of the LC-transform for one nonterminal.

```
type LCStep env a inp out =
  Trafo (MapA_X env a) Prods (Tuple (T env) inp) out
```

The type variable *env* is the type of the original grammar, and *a* is the type of the current nonterminal. The type variables *inp* and *out* are the *Arrow*-input and output. The meta-data we maintain is a table of type *MapA_X* in which we keep a mapping from discovered left-corner symbols to corresponding nonterminal references. The type of our terms is *Prods*. As extra input we pass a *Ref*-transformer which is used to map references from the original grammar to corresponding references in the transformed grammar (*Tuple* (*T env*)).

Now, all that is left to do is to glue all the pieces defined above together. We start with the function *insert*:

```
insert :: ∀env a x . Env Prods env env      → Symbol x env
                    → LCStep env a (Prod (x → a)) (Prods a)
insert grammar x = Trafo (λ(MapA_X m) →
  case m x of
  Nothing → case (second (arr    (λp → PS [p]) >>> newNont x)) >>>
                 rule2 grammar x
            of Trafo step → step (MapA_X m)
  Just r  → TrafoE
            (MapA_X m)
            (λ(TP (_, p)) t e → (PS []
                                , t
                                , updateEnv (λ(PS ps) → PS (p : ps))
                                            r e
                                )
            )
)
```

This function takes the original grammar and a left-corner symbol $x$ as input. It yields a transformation that takes as input a production for the nonterminal $A\_X$ 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 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 nonterminal $A\_X$, and add the new production at this position.

If in the function *rule2* the left-corner is a terminal symbol *rule2a* is applied and the new production rule is returned as *Arrow*-output, and in case it is a

nonterminal 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 Prods env env → Symbol x env
      → LCStep env a (Ref (x → a)) (Prods a)
rule2 grammar (Term a) = arr (λ(TP (_, a_x)) → PS [rule2a a a_x])
rule2 grammar (Nont b) =
  case lookupEnv b grammar of
    PS ps → sequenceA (map (rule2b grammar) ps)
              >>> arr (      λ(List pss) → PS (concatMap unPS pss))
```

We now define the functions *rule2a*, and *rule2b* that implement the corresponding rules of the Left-Corner algorithm. Firstly, *rule2a*, which does not introduce a new nonterminal, but simply provides new productions for the nonterminal ($A$) 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 reference for the nonterminal $A\_B$, and constructs a new production which is subsequently inserted. Note that the reference transformer *env2s* is applied to the nonterminal references in *beta* to map them on the corresponding references in the new grammar.

```
rule2b :: Env Prods env env → Prod b env
       → LCStep env a (Ref (b → a)) (Prods a)
rule2b grammar (Seq x beta) =
  arr (λ(TP (env2s, a_b)) → TP (env2s
                                , append (flip (.))
                                         (mapProd env2s beta)
                                         (Nont a_b)
                                )
      )
  >>> insert grammar x
```

The function *rule1* is almost identical to *rule2*, the only difference is that it deals with direct left-corners and hence does not involve a "parent" nonterminal $A\_B$.

```
rule1  :: Env Prods env env → Prod a env
       → LCStep env a Unit (Prods a)
rule1 grammar (Seq x beta) =
  arr (λ(TP (env2s, Unit)) → TP (env2s, mapProd env2s beta))
  >>> insert grammar x
```

The function *rules1* is defined by induction over the original grammar (i.e. it iterates over the nonterminals) 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 nonterminals:

```
rules1 :: Env Prods env env → Env Prods env env'
       → Trafo Unit Prods (T env) (Mapping env')
rules1 productions (Cons nt@(PS prods) ps) =
    (  (initA_X (      arr (λtenv_s → TP (tenv_s, Unit))
          >>>          sequenceA (map (rule1 productions) prods)
          >>>          arr (λ(List pss) → PS (concatMap unPS pss))
        ) >>>          newSRef)
  &&& rules1 productions ps
   )
  >>> arr (λ(TP (r, (Mapping e))) → Mapping (Cons r e))
rules1 _ Empty = arr (const (Mapping Empty))
```

The result of *rules1* is the complete transformation represented as a value of type *Trafo*. At the top-level the transformation does not use meta-data, hence the type *Unit*. As input the transformation needs a reference transformer to remap nonterminals of the old grammar to the new grammar. During the transformation *rules1* inserts the new definitions for nonterminals of the original grammar, and remembers the new locations for these nonterminals in a *Mapping*. This *Mapping* can be converted into the required reference transformer, which must be fed-back as the *Arrow*-input.

This feed-back loop is made in the function *leftcorner* using the *loop* combinator:

```
leftcorner :: ∀a . Grammar a → Grammar a
leftcorner (Grammar start productions) =
  let result = runTrafo
               (loop
                  (  arr (λ(TP (_, menv_s)) → map2trans menv_s) >>>
                     (arr (λtenv_s → unT tenv_s start)
                        &&& rules1 productions productions
                     )
                  )
               ) Unit  {-meta-data -} ⊥ {-input -}
  in case result of Result _ r gram → Grammar r gram
```

The resulting transformation is run using $\bot$ 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. Furthermore, *Unit* is used as meta-data. The result can now easily be mapped

onto a parser (see website). Of course, we cannot use the abstract syntax representation to actually parse, so we define a function *compile* which maps a *Grammar a* onto a real *Parser a* (Swierstra and Duponcheel 1996). We compile each of the nonterminals into a parser; nonterminals occurring in the right-hand side of a production are looked up the resulting environment, which maps nonterminal references to parsers.

```
mapEnv :: (∀a . f a s → g a s) → Env f s env → Env g s env
mapEnv f Empty          = Empty
mapEnv f (Cons v rest) = Cons (f v) (mapEnv f rest)

newtype Const f a s = C{ unC :: f a }

compile :: ∀a . Grammar a → Parser a
compile (Grammar (start :: Ref a env) rules) =
  unC (lookupEnv start result) where
    result = mapEnv (λ(PS ps) → C (choice [comp p | p ← ps])) rules
    comp :: ∀a . Prod a env → Parser a
    comp (End x)            = succeed x
    comp (Seq (Term t) ss) = (flip ($)) <$> token t <*> comp ss
    comp (Seq (Nont n) ss) = (flip ($)) <$> unC (lookupEnv n result)
                                        <*> comp ss
```

# 5 Conclusion

We have shown how to use the Haskell type system and its extensions to perform a fully typed program transformation. Doing so we have used a wide variety of type system concepts: placing existentials precisely at the positions where needed, making things polymorphic where needed, using loop combinators to feed back the result of the computation into the computation inside the scope of an existential, using GADTs to type the environments we construct, scoped type variables, splitting the type labels of the environment into a *use* and a *def* part and thus temporarily decoupling the types of the occurring references and the types associated with the terms in the environment being constructed. We introduced an arrow like style for composing the transformations. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

We think that studying the algorithm and its approaches to the various subproblems is indispensable for anyone who wants to program similar transformation-based algorithms in a strongly typed setting. Some might wonder why the approach taken may be necessary at all, and why not resort to off-line techniques, and they have a point. It is often easier to work in an untyped setting, only to check the generated result afterwards for type correctness. We claim however that this is also a form of overkill. Would you really want to call the GHC

again when linking your program, as would be needed in the case of combining *read*s. Furthermore one can see the added complexity as a partial correctness proof of the transformation, and as we all know proofs of correct lemmas are superfluous.

We believe that the arrow-based library will turn out to be useful in building programs that transform typed abstract syntax, and that the pattern we have followed in this paper will be followed in many more interesting applications to come.

Finally a remark on efficiency. It may seem that the chosen representation is expensive and clumsy. In the first place we could of course take a smarter type than just Peano-numbers for encoding pointers. We think however that these costs are to be compared with other solutions in an online setting; the costs of calling external programs like a parser generator and a compiler like the GHC, and the costs of type inferencing and generating code, and the cost of dynamically linking in this code will in general be much more expensive. Furthermore they might not even be available in the context where the program is executing. We have been using our own parser combinators which analyze themselves at each run of our Haskell compiler, without any measurable overhead.

# References


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

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002. ISBN 1-58113-487-8.

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. ISBN 1-58113-850-4.

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

James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report TR2003-1901, Cornell University, 2003. `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.c%is/TR2003-1901`.

Jeroen Fokker. Functional parsers. In *Advanced Functional Programming, First International Spring School*, volume 925 of *LNCS*, pages 1–23, Båstad, Sweden, May 1995. Springer-Verlag. URL `http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps`.

GHC. Glasgow haskell compiler. URL `http://www.haskell.org/ghc/`.

G. Hutton and H.J.M. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1998.

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.

Daan Leijen. Parsec, a fast combinator parser. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Utrecht University, 2001.

R. Moore. Removing left recursion from context-free grammars, 2000. URL `citeseer.ist.psu.edu/article/moore00removing.html`.

Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, volume LNCS 3286, pages 136 – 167, October 2004.

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340.

S. D. Swierstra. Combinator parsers: From toys to tools. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001. URL `http://math.tulane.edu/~entcs/`.

S.D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell do you read me? constructing and composing efficient top-down parsers at runtime. In A. Gill, editor, *Haskell Symposium*. ACM, 2008.

Philip Wadler. How to replace failure with a list of successes. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, 1985.

Stephanie Weirich. Type-safe cast. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 58–67. ACM press, 2000. ISBN 1-58113-202-6.