Dependently Typed Grammars

Kasper Brink[1], Stefan Holdermans[2], and Andres Löh[1]

[1] Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
{kjbrink,andres}@cs.uu.nl
[2] Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
stefan@vectorfabrics.com

Abstract. Parser combinators are a popular tool for designing parsers in functional programming languages. If such combinators generate an abstract representation of the grammar as an intermediate step, it becomes easier to perform analyses and transformations that can improve the behaviour of the resulting parser. Grammar transformations must satisfy a number of invariants. In particular, they have to preserve the semantics associated with the grammar. Using conventional type systems, these constraints cannot be expressed satisfactorily, but as we show in this article, dependent types are a natural fit. We present a framework for grammars and grammar transformations using Agda. We implement the left-corner transformation for left-recursion removal and prove a language-inclusion property as use cases.

Keywords: context-free grammars, grammar transformation, dependently typed programming.

1 Introduction

Parser combinators are a popular tool for designing parsers in functional programming languages. Classic combinator libraries [1–4] directly encode the semantics of the parsing process. The user of such a library builds a function that – when run – attempts to parse some input and produces a result on a successful parse.

Many of today's parser combinator libraries, however, try to compute much more than merely the result. The reasons are manifold, but most prominently efficiency and error reporting. The technique is to choose an abstract representation for the parser that enables computing additional information, such as lists of errors and their positions or lookahead tables, or to perform optimizations such as left-factoring automatically.

If one takes this approach to the extreme, one ends up with a combinator library that builds an abstract representation of the entire grammar first, coupled with the desired semantics. This representation still contains the complete information the user has specified, and is therefore most suitable for performing

C. Bolduc, J. Desharnais, and B. Ktari (Eds.): MPC 2010, LNCS 6120, pp. 58–79, 2010.

analyses and transformations. After the grammar has been transformed to satisfaction, the library can then interpret the grammar as a parser, again with a choice on which parsing algorithm to employ.

However, operating on the abstract representation is somewhat tricky. There are several invariants that such operations must satisfy. In particular, they have to preserve the semantics associated with the grammar. Using conventional type systems, writing transformations in even a type-correct way can therefore be difficult, and even if it succeeds, the underlying constraints can often not be expressed satisfactorily.

In this article, we present a framework for grammars and grammar transformations using the dependently typed language Agda [5, 6]. We show that dependent types are a natural fit for the kinds of constraints and invariants one wants to express when dealing with grammars. In our framework, grammars are explicitly parameterized over the sets of terminals and nonterminals. We can talk about the left- and right-hand sides of productions in the types of values, and express properties such as that a certain production does not have an empty right-hand side. Each production carries its semantics, and the shape of the production determines the type of the associated semantic function.

As a case study, we present the left-corner transform for removal of left recursion from a grammar. Contrary to most other presentations of this grammar transformation, we also show how the semantic functions are transformed. Furthermore, we explain how properties about the transformation can be proved if desired, and give a language-inclusion property as an example.

The paper concludes with a discussion of what we have achieved and what we envision for the future, and a treatment of related work.

The complete Agda code on which this paper is based is available for download [7].

2 Grammar Framework

An important aspect of our approach is that we do not encode a grammar merely as a set of production rules that can be applied to sequences of symbols. Instead, each grammar has an associated semantics, and the grammar and semantics are encoded together. For every production of the grammar there is a corresponding semantic function, which is applied during parsing when that production is recognized in order to compute a parse result. Naturally, the types of the semantic functions must be consistent with the way in which they are applied. Below, we describe how this is enforced in our framework.

2.1 Representing Grammars

We begin by describing how grammars and the associated semantic functions are represented. A number of parameters (such as the type of nonterminals and terminals) are fixed for the whole development. We therefore assume that all subsequent definitions in this section are part of an Agda parameterized module:

```
module Grammar (Terminal : Set) (Nonterminal : Set)
               (_≟t_ : Decidable {Terminal} _≡_)
               (_≟n_ : Decidable {Nonterminal} _≡_)
               ([[_]] : Nonterminal → Set) where
```

We require that both terminals and nonterminals come with decision procedures for equality.

We define a *Symbol* type, which is the union of the *Terminal* and *Nonterminal* types:

```
data Symbol : Set where
  st : Terminal → Symbol
  sn : Nonterminal → Symbol
```

In the following, we will often need lists of symbols, terminals and nonterminals, so we define

```
Symbols      = List Symbol
Terminals    = List Terminal
Nonterminals = List Nonterminal
```

as abbreviations.

To ensure that the semantic functions of the grammar are consistently typed with respect to the productions, the module signature introduces the mapping $[\![\_]\!]$, which assigns a *semantic type* to each nonterminal. This is the type of values that are produced when parsing that nonterminal (i. e., parsing A results in values of type $[\![\ A\ ]\!]$).

The types of the semantic functions are determined by the nonterminals in the corresponding productions, and the semantic mapping $[\![\_]\!]$. A semantic function for a production computes a value for the left-hand side (LHS) from the parse results of the right-hand side (RHS) nonterminals. Thus, its argument types are the RHS nonterminal types and its result type is the LHS nonterminal type. For example, the production $A \to a\,B\,b\,C\,c$ has a semantic function of type $[\![\ B\ ]\!] \to [\![\ C\ ]\!] \to [\![\ A\ ]\!]$. The semantic type of a *production* is the type of its semantic function; we shall write the semantic type of a production $A \to \beta$ as $[\![\ \beta \parallel A\ ]\!]$. This can be defined in Agda as follows:

```
[[_‖_]] : Symbols → Nonterminal → Set
[[ []          ‖ A ]] = [[ A ]]
[[ st _ :: β ‖ A ]] = [[ β ‖ A ]]
[[ sn B :: β ‖ A ]] = [[ B ]] → [[ β ‖ A ]]
```

Each nonterminal B in the RHS adds an argument of type $[\![\ B\ ]\!]$ to the semantic type, whereas terminal symbols in the RHS are ignored.[1]

[1] It would be possible to associate semantics also with terminals (for example, if all identifiers of a language are represented by a common terminal), but in order to keep the presentation simple, we do not consider this variation in this paper.

$$
\begin{aligned}
E &\rightarrow E\ B\ N \mid N \\
B &\rightarrow + \mid - \\
N &\rightarrow 0 \mid 1
\end{aligned}
$$

Fig. 1. Example grammar

We can now define a datatype to represent a production:

data *Production* : *Set* **where**
$\quad prod : (A : Nonterminal) \rightarrow (\beta : Symbols) \rightarrow [\![\ \beta \parallel A\ ]\!] \rightarrow Production$

A production consists of an LHS nonterminal A, the RHS symbols β, and an associated semantic function of type $[\![\ \beta \parallel A\ ]\!]$. Agda's dependent type systems enables us to concisely specify how the type of the semantic function depends on the shape of the production.

As an example of the representation of grammars in our framework we consider the grammar shown in Figure 1. It is a small grammar fragment that derives (from the start symbol E) a language of arithmetic expressions involving the numbers 0 and 1, and left-associative binary operators + and −. We shall refer to this example grammar throughout the article.

For this grammar, we define a semantics that evaluates an arithmetic expression to the number it represents. The nonterminals E and N each evaluate to a natural number, and B evaluates to a binary operator on naturals. Given a suitable definition for the type *Nonterminal*, we can define the semantic mapping as:

$$
\begin{aligned}
&[\![\_]\!] : Nonterminal \rightarrow Set \\
&[\![\ E\ ]\!] = \mathbb{N} \\
&[\![\ N\ ]\!] = \mathbb{N} \\
&[\![\ B\ ]\!] = \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
\end{aligned}
$$

By assigning these types to the nonterminals, we also fix the semantic types of the productions. Below, we show the semantic types for four productions from the example grammar:

$$
\begin{array}{lll}
E \rightarrow E\ B\ N & [\![\ E\ B\ N \parallel E\ ]\!] & = \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
E \rightarrow N & [\![\ N \parallel E\ ]\!] & = \mathbb{N} \rightarrow \mathbb{N} \\
B \rightarrow + & [\![\ + \parallel B\ ]\!] & = \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
N \rightarrow 1 & [\![\ 1\ \parallel N\ ]\!] & = \mathbb{N}
\end{array}
$$

We can encode these productions in Agda using the *Production* datatype (with terminal symbols represented by the built-in character type). This encoding includes the desired semantic functions, which match the types shown above.

```
p1 = prod E (sn E :: sn B :: sn N :: []) (λ x f y → f x y)
p2 = prod E [sn N]                       id
p3 = prod B [st '+']                     _+_
p4 = prod N [st '1']                     1
```

2.2 Constraints on Productions

Except for the consistency of the semantic function with respect to the LHS and RHS nonterminals, the *Production* datatype imposes no constraints on the form of a production. In some cases we wish to specify at the type level that a production has a specific left-hand side. We shall need this information when constructing a parser from a list of productions in Section 2.5, in order to show that the parse results of the constructed parser are consistently typed. To constrain the LHS of a production, we define an indexed datatype *ProductionLHS*:

```
projlhs : Production → Nonterminal
projlhs (prod A _ _) = A

data ProductionLHS : Nonterminal → Set where
  prodlhs : (p : Production) → ProductionLHS (projlhs p)
```

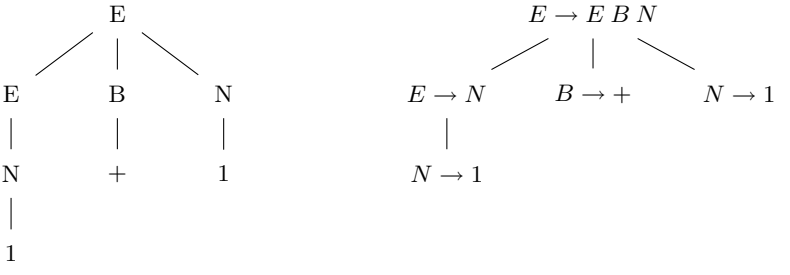
In Agda, a constructor of a datatype can restrict the index of the datatype: in this case, the wrapper constructor *prodlhs* wraps around an "ordinary" production p and uses the LHS of p as the index. In this way we expose information about the value of the production at the type level. We define a synonym for a list of productions with a specific LHS nonterminal:

```
ProductionsLHS : Nonterminal → Set
ProductionsLHS A = List (ProductionLHS A)
```

Such a list can be obtained by filtering an arbitrary list of productions with the function *filterLHS*:

```
filterLHS : (A : Nonterminal) → Productions → ProductionsLHS A
filterLHS _ [] = []
filterLHS A (prod B β sem :: ps) with A ≟n B
filterLHS A (prod .A β sem :: ps) | yes refl = prodlhs (prod A β sem) ::
                                               filterLHS A ps
filterLHS A (prod B β sem :: ps) | no _      = filterLHS A ps
```

For a non-empty input list, the LHS of the first production, B, is compared to the specified nonterminal, A, in the **with**-clause. If the equality test $\_ \stackrel{?}{=} n \_$ is successful, it returns a proof *refl* which is the single constructor of the equality type $\_ \equiv \_$. Pattern matching on *refl* exposes this equality to the type system. In the *yes*-branch, we can therefore assume that B is equal to A (as expressed in Agda using the dot-pattern $.A$) and add the production to a list of type *ProductionLHS A*. In the *no*-branch, the production is discarded, and we continue with the rest of the list. The result type of *filterLHS* holds evidence of the filtering step that has been performed.



(a) Symbols in nodes (b) Productions in nodes

Fig. 2. Representations of parse trees

2.3 Parse Trees

To enable us to state and prove properties about grammars and their transformations we must define a representation for parse trees. In the conventional depiction of parse trees, internal nodes are labelled with nonterminals, and leaves with terminals. For a parse tree to be consistent with a grammar, an internal node A and its direct descendants $X_1, \ldots, X_n$ must form a production $A \to X_1 \cdots X_n$ of the grammar. In Figure 2a, we show the conventional representation of the parse tree for the sentence "$1 + 1$" in the example grammar.

In our case, it is more convenient to label the nodes of the parse tree with *productions*. This makes it easier to express constraints on the productions (e. g., that they belong to a certain grammar), and enables us to store the semantic functions in the parse tree, so that a parse result can be computed from it. The *root* of a parse tree shall refer to the LHS nonterminal of the production in the root node. Figure 2b shows how we represent the parse tree for the sentence "$1 + 1$" in our framework.

Our representation of parse trees introduces redundant information between a node and its children. To ensure that a parse tree is well-formed, we require that the nonterminals in the RHS of the production in each node correspond to the roots of its subtrees. This correspondence is encoded with the help of the relation $\_\sim\_$:

$$\_\sim\_ : \mathit{Symbols} \to \mathit{Nonterminals} \to \mathit{Set}$$
$$\beta \sim ns \;=\; \mathit{filterN}\ \beta \equiv ns$$

where $\mathit{filterN} : \mathit{Symbols} \to \mathit{Nonterminals}$ filters the nonterminals out of a list of symbols.

We wish to enforce that the productions in a parse tree belong to a certain grammar. In some cases, the productions are subject to additional constraints, such as the requirement that the RHS is nonempty. The parse-tree datatype defined below is therefore parametrized over a predicate $Q : \mathit{Production} \to \mathit{Set}$, which represents the combined constraints that are satisfied by each production. The datatype is implemented with two mutually recursive definitions:

```
mutual
  data ParseTree (Q : Production → Set) : Set where
    node : (p : Production) → Q p → (cs : List (ParseTree Q)) →
           (projrhs p ∼ map projroot cs) → ParseTree Q
  projroot : ∀ { Q } → ParseTree Q → Nonterminal
  projroot (node (prod A _ _) _ _ _) = A
```

A *node* of a parse tree contains a *Production* (which includes a semantic function), a proof that it satisfies the predicate Q, a list of children, and a proof that the nonterminals on the RHS of the production match the roots of the child parse trees. The leaves of the tree contain productions with a RHS that consists entirely of terminals (or is empty).[2] In the function *projroot*, the type argument Q remains implicit, as indicated by the curly brackets. Agda will try to infer the argument whenever the function is called.

Given a parse tree, it is straight-forward to compute the sentence it represents:

```
merge : Symbols → List Terminals → Terminals
merge []          []       = []
merge (st b  :: β) us      = b :: merge β us
merge (sn B :: β) (u :: us) = u ++ merge β us
merge _           _        = []
sentence : ∀ { Q } → ParseTree Q → Terminals
sentence (node (prod _ β _) _ cs _) = merge β (map sentence cs)
```

The function *sentence* traverses the children of a node recursively. The helper function *merge* then concatenates the recursively computed subsequences with the terminals stored in the production of the node.

Another interesting computation over a parse tree is the semantic value that is represented by such a tree:

```
semantics : ∀ { Q } → (pt : ParseTree Q) → ⟦ projroot pt ⟧
```

The implementation performs a fold on the parse tree, building up the result starting from the leaves by applying all the semantic functions stored in the nodes. We omit the code for brevity.

2.4 Parser Combinators

To construct a parser for a grammar encoded in our representation, we make use of a parser combinator library. Parser combinators form a domain-specific embedded language for the implementation of parsers in functional languages such as Agda. The interface consists of several elementary parsers, and combinators that allow us to construct complex parsers out of simpler ones. This formalism

[2] The definition of *ParseTree* does not pass Agda's positivity checker. This is mainly a problem of the current checker, not a fundamental problem. We can circumvent the problem by rewriting our code slightly, but here, we opt for readability.

offers a convenient way to implement parsers using a notation that stays close to the underlying grammar. Here, we review one possible interface for parser combinators; their implementation is described elsewhere [2, 8].

The basic type *Parser A* denotes a parser that returns values of type A. We define the following elementary parsers:

$$\begin{aligned} &\mathit{symbol} : \mathit{Terminal} \to \mathit{Parser}\ \mathit{Terminal} \\ &\mathit{succeed} : \forall\, \{\, A \,\} \to A \to \mathit{Parser}\ A \\ &\mathit{fail} \quad\ : \forall\, \{\, A \,\} \to \mathit{Parser}\ A \end{aligned}$$

The parser *symbol* recognizes a single terminal symbol and returns that symbol as a witness of a successful parse. The parser *succeed* recognizes the empty string (which always succeeds) and returns the supplied value of type A as the parse result. The parser *fail* always fails, so it never has to produce a value of the result type A.

The library contains the following elementary combinators:

$$\begin{aligned} &\_<|>\_ : \forall\, \{\, A \,\} \to \mathit{Parser}\ A \to \mathit{Parser}\ A \to \mathit{Parser}\ A \\ &\_<*>\_ : \forall\, \{\, A\ B \,\} \to \mathit{Parser}\ (A \to B) \to \mathit{Parser}\ A \to \mathit{Parser}\ B \end{aligned}$$

The combinator $\_<|>\_$ implements a choice between two parsers: the resulting parser recognizes either a sentence of the left parser or of the right parser. The result types of the parsers must be the same. The combinator $\_<*>\_$ implements sequential composition: the resulting parser recognises a sentence of the left parser followed by a sentence of the right parser. The parse results of the two parsers are combined by function application: the left parser produces a function, the argument type of which must match the result type of the right parser. We also define the derived combinator $\_<*\,\_$, which recognizes its arguments in sequence just like $\_<*>\_$, but which only returns the parse result of the left parser, discarding the result of the right.

2.5 Generating Parsers

The function *generateParser* constructs a parser for a grammar by mapping its productions onto the parser combinator interface, and it has type:

$$\mathit{generateParser} : \mathit{Productions} \to (S : \mathit{Nonterminal}) \to \mathit{Parser}\ [\![\ S\ ]\!]$$

It takes a list of productions of the grammar and a start nonterminal S, and constructs a parser for S that returns values of type $[\![\ S\ ]\!]$. The implementation of the function makes use of three mutually recursive subfunctions:

$$\begin{aligned} &\mathit{generateParser}\ \mathit{gram}\ =\ \mathit{gen}\ \textbf{where} \\ &\quad \textbf{mutual} \\ &\qquad \mathit{gen} : (A : \mathit{Nonterminal}) \to \mathit{Parser}\ [\![\ A\ ]\!] \\ &\qquad \mathit{gen}\ A\ =\ (\mathit{foldr}\ \_<|>\_\ \mathit{fail} \circ \mathit{map}\ \mathit{genAlt} \circ \mathit{filterLHS}\ A)\ \mathit{gram} \\ &\qquad \mathit{genAlt} : \forall\, \{\, A \,\} \to \mathit{ProductionLHS}\ A \to \mathit{Parser}\ [\![\ A\ ]\!] \end{aligned}$$

$$
\begin{aligned}
&\mathit{genAlt}\ (\mathit{prodlhs}\ (\mathit{prod}\ A\ \beta\ \mathit{sem})) \;=\; \mathit{buildParser}\ \beta\ (\mathit{succeed}\ \mathit{sem})\\
&\mathit{buildParser} : \forall\ \{\, A \,\}\ \beta \to \mathit{Parser}\ [\![\ \beta \parallel A\ ]\!] \to \mathit{Parser}\ [\![\ A\ ]\!]\\
&\mathit{buildParser}\ [\,]\qquad\quad p \;=\; p\\
&\mathit{buildParser}\ (\mathit{st}\ \ b\ ::\beta)\ p \;=\; \mathit{buildParser}\ \beta\ (p\ <\!\ast\ \mathit{symbol}\ b)\\
&\mathit{buildParser}\ (\mathit{sn}\ B ::\beta)\ p \;=\; \mathit{buildParser}\ \beta\ (p\ <\!\ast\!>\ \mathit{gen}\ B)
\end{aligned}
$$

The function *gen* takes a nonterminal A and generates a parser for A that returns values of type $[\![\ A\ ]\!]$. It first selects all productions with LHS A from the grammar using *filterLHS*. To each of these productions it applies *genAlt*, which generates a parser that corresponds to that particular alternative for A. The alternatives are combined into a single parser by folding with the parallel composition combinator $\_<|>\_$ (which has *fail* as a unit).

The function *genAlt* generates a parser for a single alternative; that is, the derivation always starts with the specified production. Note that the connection between the left-hand side nonterminal of the production and the result type of the parser is made in the type signature of *genAlt*. The semantic function *sem* is lifted to the trivial parser *succeed sem* with type $\mathit{Parser}\ [\![\ \beta \parallel A\ ]\!]$. The actual parser construction is performed by *buildParser*.

The function *buildParser* builds the parser by recursing over the right-hand side symbols β. The argument p is an accumulating parameter that is expanded into a parser that recognizes β. When β is empty, we return the constructed parser p, which has type $\mathit{Parser}\ [\![\ A\ ]\!]$. If β starts with a terminal b, we recognize it with *symbol* b, leaving the semantic types unchanged. If β starts with a nonterminal B, we generate a parser for B by calling *gen* recursively, and append this to p. Note that in this branch, p has type $\mathit{Parser}\ ([\![\ B\ ]\!] \to [\![\ \beta \parallel A\ ]\!])$, and *gen* B has type $\mathit{Parser}\ [\![\ B\ ]\!]$, and the sequential composition removes the leftmost argument from the parser's result type.

3 Left-Corner Transform

The function *generateParser* from the preceding section does not pass Agda's termination checker. This is not surprising, since we do not impose any restrictions on the grammar at this point. In particular, a left-recursive grammar will lead to a non-terminating parser.

In this section, we discuss the left-corner transform (LCT), a grammar transformation that removes left recursion from a grammar [9, 10].

3.1 Transformation Rules

The transformation is presented below as a set of transformation rules that are applied to the productions of a grammar. If the grammar satisfies certain preconditions, applying the rules yields a transformed grammar that derives the same language and that does not contain left-recursive nonterminals. A nonterminal A is *left-recursive* if it derives a sequence of symbols beginning with A itself (i. e., $A \overset{*}{\Rightarrow} A\,\beta$). Such nonterminals can lead to non-termination with top-down parsers.

The LCT is based on manipulation of the left corners of the grammar. A symbol X is a *direct left corner* of the nonterminal A if there is a production $A \to X\beta$ in the grammar. The *left-corner relation* is the transitive closure of the direct left-corner relation.

The transformation extends the set of nonterminals with new nonterminals of the form A–X, where A and X are a nonterminal and a symbol of the original grammar, respectively. A new nonterminal A–X represents the part of an A that follows an X. For example, if $A \overset{*}{\Rightarrow} Bcd \overset{*}{\Rightarrow} abcd$, then $A\text{–}B \overset{*}{\Rightarrow} cd$ and $A\text{–}a \overset{*}{\Rightarrow} bcd$.

The three transformation rules for the left-corner transform, as formulated by Johnson [10], are as follows:

$$\forall A \in N,\ a \in T: \qquad A \to a\, A\text{–}a \qquad \in P' \tag{1}$$

$$\forall C \in N,\ A \to X\beta \in P: \qquad C\text{–}X \to \beta\, C\text{–}A \ \in P' \tag{2}$$

$$\forall A \in N: \qquad A\text{–}A \to \epsilon \qquad \in P' \tag{3}$$

The rules are universally quantified over the terminals T, nonterminals N and productions P of the original grammar. The set P' contains the productions of the transformed grammar; the start symbol remains the same.

Some of the nonterminals and productions generated by these rules are useless: they can never occur in a complete derivation of a terminal string from the start symbol. There are other formulations of the LCT which avoid generating such useless productions; we have chosen this variant because its simplicity makes it easier to prove properties about the transformation (cf. Section 4).

3.2 Transforming Productions

Because rule (2) refers to the left corner of the input production, it is not defined for ϵ-productions. Therefore, we must encode the precondition that the transformation can only be applied to non-ϵ-productions. We begin by defining a predicate that identifies productions with a nonempty RHS:

```
isNonEpsilon : Production → Set
isNonEpsilon p = T ((not ∘ null ∘ projrhs) p)
```

The standard library function T maps boolean values to the corresponding propositions: the result is either the type $\top$ for truth with one inhabitant, or the type $\bot$ for falsity without inhabitants.

The type of non-ϵ-productions is a dependent pair of a production and a proof that its RHS is nonempty:

```
data NonEpsilonProduction : Set where
  nϵ : (p : Production) → {_ : isNonEpsilon p} →
       NonEpsilonProduction
```

We make the proof implicit,[3] since we typically only want to refer to it when dismissing the "impossible case" (empty RHS) in function definitions with a *NonEpsilonProduction* argument.

[3] In Agda, implicit arguments must always be named, hence the underscore in the type signature.

In the LCT, the transformed grammar uses a different set of nonterminals than the original grammar. To encode this in our implementation we require two separate instantiations of the *Grammar* module. We define modules O and T, and inside these modules we instantiate the *Grammar* module with the appropriate parameters. This enables us to refer to entities from either grammar with the prefixes "O" and "T".[4]

The set of nonterminals of the transformed grammar is derived from that of the original grammar by adding nonterminals of the form A–X. This is represented by the datatype *TNonterminal*:

```
data TNonterminal : Set where
  n     : ONonterminal → TNonterminal
  n_-_ : ONonterminal → OSymbol → TNonterminal
```

With this datatype, the original nonterminal A can be encoded in the transformed grammar as $n\ A$, and the new nonterminal A–B as $n\ A - O.sn\ B$.

The semantic types of the transformed nonterminals are a function of the original semantic types:

```
T⟦_⟧ : TNonterminal → Set
T⟦ n A ⟧          = ⟦ A ⟧
T⟦ n A - O.st b ⟧ = ⟦ A ⟧
T⟦ n A - O.sn B ⟧ = ⟦ B ⟧ → ⟦ A ⟧
```

The semantic types of the original nonterminals are preserved in the transformed grammar. To explain the semantic type for a nonterminal A–X, consider the situation where we have recognized the left corner X, and continue by recognizing the remainder of A. If X was a terminal b, we must simply produce a value of type $[\![A]\!]$, but if X was a nonterminal B, we have already got a value of type $[\![B]\!]$, so to produce a result of type $[\![A]\!]$ we need a function $[\![B]\!] \to [\![A]\!]$.

With the representation of the two grammars in place, we now turn to the transformation itself. At the top level, we implement the universal quantification over symbols and productions in the transformation rules with a combination of *concat* and *map*.

```
lct : ONonEpsilonProductions → TProductions
lct ps = concatMap (λ A → map (rule1 A) ts) ns ++
         concatMap (λ C → map (rule2 C) ps) ns ++
         map rule3 ns
           where ts = terminals    ps
                 ns = nonterminals ps
```

The type of *lct* specifies the precondition that the productions of the original grammar must have a nonempty RHS. In the interface of the *Grammar* module, we do not require an operation to enumerate all symbols in the sets *Terminal*

[4] For readability, we also define synonyms such as $O[\![\_]\!][\![\_]\!] = O.[\![\_]\!][\![\_]\!]$.

and *Nonterminal*. Instead we use the functions *terminals* and *nonterminals* here, which traverse the list of productions, collect all terminal or nonterminal symbols encountered, and remove duplicates. This means that we only quantify over symbols that are actually used in the grammar.

The functions *rule1* and *rule3* directly encode the corresponding transformation rules (1) and (3) from page 67:

$$
\begin{aligned}
&\mathit{rule1} : \mathit{ONonterminal} \to \mathit{Terminal} \to \mathit{TProduction}\\
&\mathit{rule1}\ A\ a = T.\mathit{prod}\ (n\ A)\ (T.\mathit{st}\ a :: [\,T.\mathit{sn}\ (n\ A - O.\mathit{st}\ a)\,])\ \mathit{id}\\
&\mathit{rule3} : \mathit{ONonterminal} \to \mathit{TProduction}\\
&\mathit{rule3}\ A = T.\mathit{prod}\ (n\ A - O.\mathit{sn}\ A)\ [\,]\ \mathit{id}
\end{aligned}
$$

In both cases, the semantics of the constructed production has type $[\![A]\!] \to [\![A]\!]$, so the corresponding semantic function is the identity.

The function *rule2* is more interesting, since it is the only rule that actually transforms productions of the original grammar.

$$
\begin{aligned}
&\mathit{rule2} : \mathit{ONonterminal} \to \mathit{ONonEpsilonProduction} \to \mathit{TProduction}\\
&\mathit{rule2}\ C\ (O.n\epsilon\ (O.\mathit{prod}\ A\ (X :: \beta)\ \mathit{sem})) =\\
&\qquad T.\mathit{prod}\ (n\ C - X)\ (\mathit{liftSymbols}\ \beta +\!\!+ [\,T.\mathit{sn}\ (n\ C - O.\mathit{sn}\ A)\,])\\
&\qquad\qquad (\mathit{semtrans}\ C\ A\ X\ \beta\ \mathit{sem})\\
&\mathit{rule2}\ \_\ (O.n\epsilon\ (O.\mathit{prod}\ \_\ [\,]\ \_)\ \{\,\})
\end{aligned}
$$

Although the notation is slightly cluttered by the various symbol constructors, it is clear that this function performs a straightforward rearrangement of the input symbols. The function $\mathit{liftSymbols} : \mathit{OSymbols} \to \mathit{TSymbols}$ maps symbols of the original grammar to the same symbols in the transformed grammar (e. g., $O.\mathit{sn}\ A$ to $T.\mathit{sn}\ (n\ A)$). We will explain *semtrans* below. The second case is required to get the function through Agda's totality checker. We consider the case that the RHS of the production is empty, but can refute it due to the implicit proof of non-emptiness contained in the constructor $O.n\epsilon$, using Agda's notation for an absurd pattern $\{\,\}$.

3.3 Transforming Semantics

One problem still remains to be solved: when transforming a production with *rule2*, how should the associated semantic function be transformed? This task is performed by the function *semtrans*. We compute the semantic transformation incrementally, by folding over the symbols in the RHS of the production, and the type of the transformation depends on the symbols we fold over. For this, we use a dependently typed fold for a list of symbols, which is defined as part of the grammar framework:

$$
\begin{aligned}
&\mathit{foldSymbols} : \{\,P : \mathit{Symbols} \to \mathit{Set}\,\} \to\\
&\qquad (\forall\ b\ \{\beta\} \to P\ \beta \to P\ (\mathit{st}\ b\ \ :: \beta)) \to\\
&\qquad (\forall\ B\ \{\beta\} \to P\ \beta \to P\ (\mathit{sn}\ B :: \beta)) \to\\
&\qquad P\ [\,] \to
\end{aligned}
$$

$$
\begin{aligned}
&(\beta : \mathit{Symbols}) \to P\ \beta \\
&\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ (\mathit{st}\ b\ \ :: \beta) &&= \mathit{ft}\ b\ \ (\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ \beta) \\
&\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ (\mathit{sn}\ B :: \beta) &&= \mathit{fn}\ B\ (\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ \beta) \\
&\mathit{foldSymbols}\ \_\ \_\ \ \mathit{fe}\ [\,] &&= \mathit{fe}
\end{aligned}
$$

This is a dependently typed generalization of the ordinary *foldr* for lists; in addition, we also make a distinction between terminal and nonterminal symbols at the head of the list. The type P is the result type of the fold, which depends on the symbols folded over.

To define the transformation of the semantic functions, we use the semantic *types* as a guide. A production $A \to B\beta$ is transformed as follows by rule (2):

$$A \to B\beta \quad \longrightarrow \quad C\text{–}B \to \beta\, C\text{–}A$$

The types of the semantic functions must be transformed accordingly:

$$[\![B\beta \| A]\!] \quad \longrightarrow \quad [\![\beta C\text{–}A \| C\text{–}B]\!]$$

This can be viewed as a function type mapping the original semantic function to the transformed semantic function; in other words, it is the type of the semantic transformation. We encode this type in Agda as:

$$
\begin{aligned}
&\mathit{semtransN} : \forall\ C\ A\ B\ \beta \to \\
&\qquad O[\![\ O.\mathit{sn}\ B :: \beta \parallel A\ ]\!] \to \\
&\qquad T[\![\ \mathit{liftSymbols}\ \beta +\!\!+ [\,T.\mathit{sn}\ (n\ C - O.\mathit{sn}\ A)] \parallel n\ C - O.\mathit{sn}\ B\ ]\!]
\end{aligned}
$$

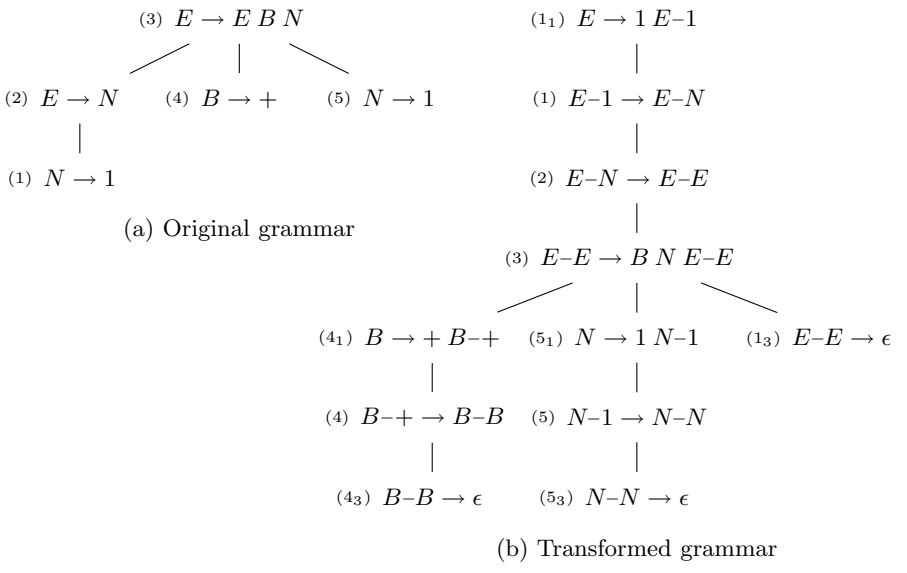
The implementation of *semtransN* is obtained by constructing a function that satisfies the given type:

$$
\begin{aligned}
\mathit{semtransN}\ \_\ \_\ \_ = O.\mathit{foldSymbols}\ &(\lambda\ c\ \ f \to f) \\
&(\lambda\ C\ f \to \lambda\ g \to f \circ \mathit{flip}\ g) \\
&(\lambda\ f\ \ g \to g \circ f)
\end{aligned}
$$

The suffix N to *semtransN* signifies that this transformation applies to a production with a *nonterminal* left corner; the function *semtransT* for a terminal left corner is analogous, with slightly different arguments to the fold. The function *semtrans* that is used in the definition of *rule2* makes the choice between the two based on the left corner of the input production.

4 Proof of a Language-Inclusion Property

Dependent types can be used to develop correctness proofs for our programs, without resorting to external proof tools. We illustrate this by proving a language-inclusion property for the LCT. This property forms part of a correctness proof for our implementation of the transformation; a full proof of correctness would also establish the converse property (thereby proving language preservation for the LCT), and the absence of left-recursion in the transformed grammar.



(a) Original grammar

(b) Transformed grammar

Fig. 3. Original and transformed parse trees

4.1 Language Inclusion

When applying grammar transformations, we usually require that they preserve the language that is generated by the grammar. In this section, we show how to prove that our implementation of the left-corner transform satisfies a *language-inclusion* property,

$$L(G) \subseteq L(G') \,, \tag{4}$$

which states that the language generated by the transformed grammar G' includes at least the language of the original grammar G.

The left-corner transform operates on the productions of a grammar by application of the transformation rules (1)–(3). The transformation of the productions leads to a corresponding transformation of the parse trees of the grammar. A parse tree is essentially a proof that the derived sentence is in the language of the grammar. To prove property (4), we must show that for every sentence w in the language of G (as evidenced by a parse tree using the productions of G), we can construct a parse tree using the productions of G'. By implementing the parse-tree transformation function we give a constructive proof of the language-inclusion property.

4.2 Relating Parse-Tree Transformation to Grammar Transformation

To illustrate the parse-tree transformation, Figure 3a shows the parse tree for the sentence "1 + 1" in the example grammar, and Figure 3b shows the parse tree for the same sentence in the transformed grammar. Johnson [10] notes that

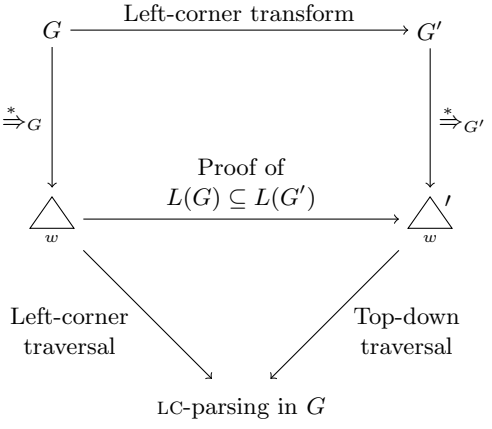


Fig. 4. Relationship between grammar transformation and parse-tree transformation

the LCT emulates a particular parsing strategy called *left-corner* (LC) parsing, in the sense that a top-down parser using the transformed grammar behaves identically to an LC-parser with the original grammar. Left-corner parsing contains aspects of both top-down and bottom-up parsing. We can characterize the parsing strategies as follows: in top-down parsing, productions are recognized before their children and their right siblings; in bottom-up parsing, productions are recognized after their children and their left siblings; and in left-corner parsing, productions are recognized after their left corner, but before their other children, and before their right siblings. Thus, the parse-tree transformation induced by the LCT satisfies the following property: for two parse trees related by the parse-tree transformation, an LC-traversal of the original tree corresponds to a top-down traversal of the transformed tree.

In Figure 3, the nodes of the original tree have been labelled in LC-order, and those of the transformed tree in top-down order. Each node of the transformed tree that is labelled with a plain number (without a subscript) is derived from the node in the original tree with the same label, by application of the LCT transformation rule (2). Note that the *left-hand side nonterminal* of an original node is reflected in the *right corner* of a transformed node. The transformed tree also contains nodes that are generated by LCT transformation rules (1) and (3), indicated by the subscripts on the labels. These nodes occur at the root and the lower right corner of all subtrees that do not correspond to a left corner in the original tree.

The relationships between grammars, parse trees and traversals are depicted schematically in Figure 4. On the top row, we see the original and transformed grammars, related by the left-corner transform. The middle row shows parse trees for the sentence w in the original and transformed grammar. These trees are related by the parse-tree transformation function, which is also the proof of the language-inclusion property (4). On the bottom row, we see that an LC-traversal

of the original parse tree, which corresponds to LC-parsing in G, recognizes productions in the same order as a top-down traversal of the transformed parse tree.

4.3 Parse-Tree Transformation: Specification

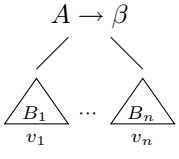
The parse-tree transformation function performs an LC-traversal of the original parse tree, transforming each original production with rule (2), and adding productions to the transformed tree in top-down order. Each subtree of the original tree that does not correspond to a left-corner is a new *goal* for the transformation. At the root and the lower right corner of these subtrees, productions are added that are generated by rules (1) and (3), respectively. Finally, we must show that the transformation of a subtree preserves the derived sentence.

We now give a precise definition of the parse-tree transformation. This consists of two mutually recursive functions $\mathcal{G}$ and $\mathcal{T}$. In the description of these functions, we use the following notation to concisely represent parse trees with root A, deriving the sentence w:

(original parse tree)

(transformed parse tree)

Note that we use this notation both to represent the *set* of parse trees with root A and sentence w (in the types of $\mathcal{T}$ and $\mathcal{G}$), and an *inhabitant* of that set (in the definitions of $\mathcal{T}$ and $\mathcal{G}$). A parse tree with a specific production $A \to \beta$ in the root node is written as:



(where $B_1, \ldots, B_n$ are the nonterminals of β)

The transformation functions are defined in Figure 5. Free variables in the type signatures, such as A and w, are universally quantified. The function $\mathcal{G}$ is the top-level transformation function, which is applied to each new goal. The type of this function specifies that the sentence of the original tree is preserved by the transformation.

The recursive transformation of the subtrees is performed by $\mathcal{T}$, which takes as arguments the current goal nonterminal C, the subtree to be transformed, and an accumulating parameter, which holds the lower right corner of the tree that is being constructed. This function satisfies the invariant that the tail v of the original sentence, concatenated with the sentence w of the tree being constructed, is the sentence of the result.

As can be seen from Figure 5, the transformation functions satisfy certain invariants related to roots of parse trees and the sentence derived by them. This is expressed in the types of $\mathcal{T}$ and $\mathcal{G}$ by referring not to arbitrary sets of parse trees, but to sets of parse trees that depend on a particular nonterminal for the root of the tree and a particular string of terminals for the sentence of the tree. Thus, the transformation functions are naturally dependently typed.

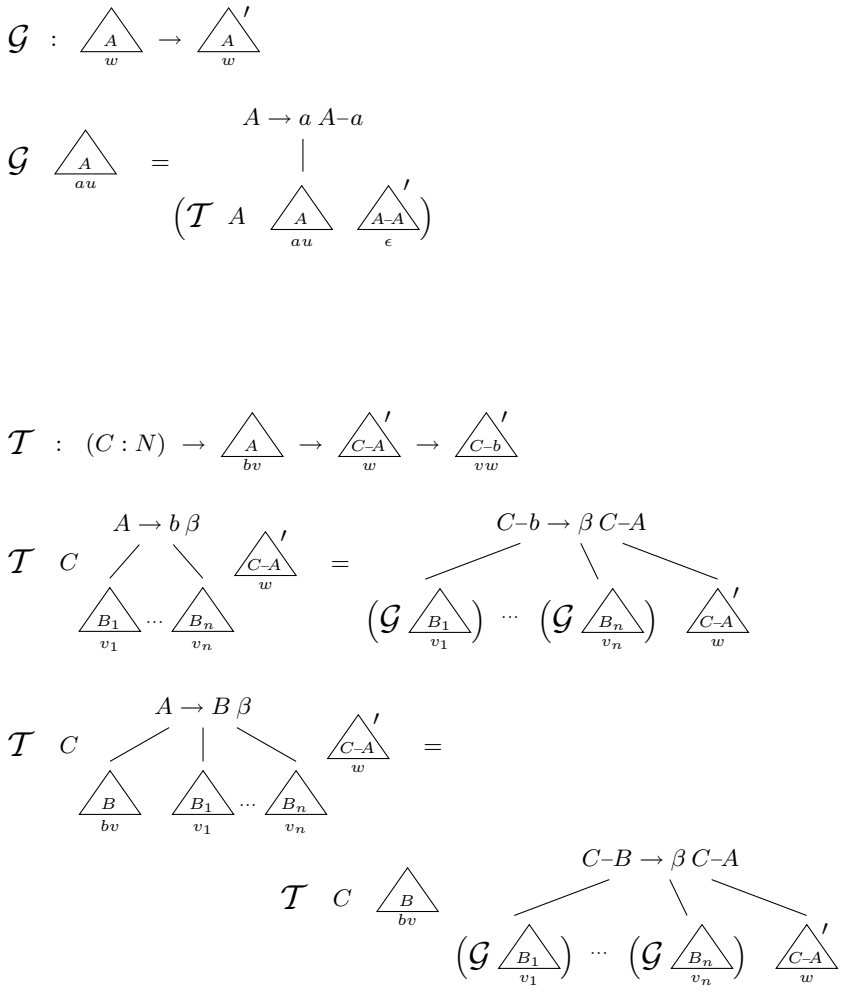


Fig. 5. Parse-tree transformation functions

4.4 Parse-Tree Transformation: Agda Implementation

We now turn to the Agda implementation of the transformation functions that we defined in pseudocode in Figure 5. Our first task is to create a representation of the parse-tree types used in the specification. We begin by defining the synonym *OGrammar*:

$$OGrammar \;=\; ONonEpsilonProductions$$

The original grammar is given as a list of productions, which are guaranteed to be non-ϵ. We now define the general types of parse trees of the original and the transformed grammar, that is, types that do not specify the root and sentence

of their inhabitants. To use the parse-tree type of Section 2.3, we must supply it with a predicate that describes the constraints that apply to each production in the parse tree. For original parse trees, we require that the production is a non-ϵ production, and that it is contained in the original grammar:

$$\begin{aligned}&\mathit{OParseTree} : \mathit{OGrammar} \to \mathit{Set}\\&\mathit{OParseTree}\ G\ =\ O.\mathit{ParseTree}\ (\lambda\ p \to \Sigma\ (O.\mathit{isNonEpsilon}\ p)\\&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ p n\epsilon \to O.n\epsilon\ p\ \{p n\epsilon\} \in G))\end{aligned}$$

Note that the parse tree itself contains "plain" productions; to express the requirement that they are contained in the grammar, we must combine them with their non-ϵ proofs to construct values of type *NonEpsilonProduction*.

For transformed parse trees, we only require that the productions are contained in the left-corner transform of the original grammar.

$$\begin{aligned}&\mathit{TParseTree} : \mathit{OGrammar} \to \mathit{Set}\\&\mathit{TParseTree}\ G\ =\ T.\mathit{ParseTree}\ (\lambda\ p \to p \in \mathit{lct}\ G)\end{aligned}$$

From the general parse-tree types *OParseTree* and *TParseTree* we can create the specific parse-tree types that are used in the types of the transformation functions. This is done by taking the dependent pair of a parse tree with a pair of proofs about its root and sentence. For original parse trees we define:

$$\begin{aligned}&\mathit{OPT} : \mathit{OGrammar} \to \mathit{ONonterminal} \to \mathit{Terminals} \to \mathit{Set}\\&\mathit{OPT}\ G\ A\ w\ =\ \Sigma\ (\mathit{OParseTree}\ G)\ (\lambda\ \mathit{opt} \to O.\mathit{projroot}\ \ \mathit{opt} \equiv A\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \times\ O.\mathit{sentence}\ \mathit{opt} \equiv w)\end{aligned}$$

The type $\mathit{OPT}\ G\ A\ w$ is the Agda representation of the type $\overset{A}{\underset{w}{\triangle}}$. The type TPT is defined in the same way.

Using the types OPT and TPT, it is straightforward to translate the types of the transformation functions into Agda. For $\mathcal{G}$ we get:

$$\mathit{transG} : \forall\ \{\, G\ A\ w\,\} \to \mathit{OPT}\ G\ A\ w \to \mathit{TPT}\ G\ (n\ A)\ w$$

And the type of $\mathcal{T}$ becomes:

$$\begin{aligned}&\mathit{transT} : \forall\ \{\, G\ A\ b\ v\ w\,\} \to\\&\qquad (C : \mathit{ONonterminal}) \to\\&\qquad C \in \mathit{nonterminals}\ G \to\\&\qquad \mathit{OPT}\ G\ A\ (b :: v) \to\\&\qquad \mathit{TPT}\ G\ (n\ C - O.\mathit{sn}\ A)\ w \to\\&\qquad \mathit{TPT}\ G\ (n\ C - O.\mathit{st}\ \ b)\ (v +\!\!+ w)\end{aligned}$$

In the translation to Agda, we have added an additional agument: the condition $C \in \mathit{nonterminals}\ G$, which states that the current goal nonterminal C is actually used in one of the productions of the grammar. In Figure 5, this was left implicit; in Agda, we need this condition to prove that the productions of the transformed tree really exist in the transformed grammar.

The implementations of *transG* and *transT* are also straightforward translations of the pseudocode of Figure 5. The resulting code does, however, require many small helper proofs in order to prove its type correctness. This clutters the structure of the transformation somewhat, compared to the pseudocode.

By implementing *transG*, we have created a machine-checkable proof that our implementation of the LCT satisfies the language-inclusion property (4).

5 Related Work

Baars et al. [11] implement a left-corner transformation of typed grammars in Haskell. To guarantee that the types of associated semantic functions are preserved across the transformation, they make use of various extensions to Haskell's type system, such as generalized algebraic datatypes for maintaining several invariants and nonstrict evaluation at the type level for wrapping the transformation in an arrow [12]. Inspired by Pasălić and Linger [13], their implementation, which is built on top of a general-purpose library for typed transformations of typed syntax [14], uses what are essentially De Bruijn indices for representing nonterminal symbols. At the expense of some additional complexity, this representation allows for a uniform representation of nonterminals across the transformation. Our approach, at the other hand, requires a dedicated representation (*TNonterminal*) for the nonterminals used in the transformed grammar and a corresponding representation for its productions (*TProduction*). In principle, our implementation could be adapted to use a uniform representation of nonterminal symbols across the transformation as well, but doing so would make it considerably more involved to state and prove properties of our transformation. Baars et al., limited by the restrictions of Haskell's type system, do not state or prove any properties of their implementation other than the preservation of semantic types.

Danielsson and Norell present a library [15] of total parser combinators in Agda. Type-correct parsers in this library are guaranteed not to be left-recursive. It would be interesting to investigate if we could generate a parser for our LCT-transformed grammars using these combinators. A more recent version of the library by Danielsson [16] can actually deal with many left-recursive grammars, by controlling the grammar traversal using a mix of induction and co-induction.

6 Conclusions

We have presented a framework for the representation of grammars, together with their semantics, in Agda. Dependent types make it possible to specify precisely how the type of the semantic functions is determined by the shape of the productions. We can generate parsers for the grammars expressed in our framework with the help of a parser-combinator library.

As an example of the use of our framework, we have shown how to implement the left-corner transform, a transformation that removes left recursion from a

grammar. This transformation consists not only of relatively simple manipulations of grammar symbols, but also requires a corresponding adaptation of the semantic functions. Fortunately, we can use the types to guide the implementation: by treating the semantic types as a specification of the desired transformation, the problem is reduced to a search for a function of the appropriate type.

Dependent types play an important role in the development of correctness proofs for our programs. We illustrate this by proving a language-inclusion property for our implementation of the LCT, which states that the transformed grammar derives at least the language of the original grammar. From the transformation rules of the LCT, it is not immediately obvious how the parse trees of the original and transformed grammars are related. A key insight in the proof, due to Johnson [10], is the realization that the grammar transformation effectively simulates a left-corner traversal of the original parse tree. This leads us to a specification of the parse-tree transformation in pseudocode, which involves several invariants on the roots and derived sentences of the parse trees. Those invariants are expressed most naturally using dependent types.

The Agda implementation of the proof is a straightforward translation of the pseudocode specification. As we have shown, the language-inclusion property can be represented elegantly in Agda as a type. The *proof* of this property, which is a parse-tree transformation function satisfying the aforementioned type, also follows directly from the pseudocode. However, to show that this function satisfies the specified type, we have to prove many small helper properties, which clutters the main proof considerably. Although Agda's interactive mode proved helpful in getting these details of the proof right, we speculate that the availability of an extensible tactic language, much as has been available in Coq for many years [17], would even further streamline the construction of proof objects.

During the development of the proof of the language-inclusion property we were confronted with inefficiencies in the current implementation of Agda (version 2.2.4). In order for the memory footprint of the typechecker to fit within the physical memory of the machine, we were forced to factor the proof – sometimes unnaturally – into several submodules. Even with this subdivision, the standalone typechecker takes about 8 minutes to check the proof on our hardware,[5] limiting the pace of development.

The framework we have presented in this paper can represent any context-free grammar, but when encoding the grammar we are limited to using plain BNF notation. In contrast, parser combinators are far more expressive than plain BNF, offering constructs such as repetition, optional phrases, and more. One of the key advantages of parser combinators is that they allow us to capture recurring patterns in a grammar by defining custom combinators. Our main focus in extending the present work is to develop a library of "grammar combinators". We envisage a combinator library with an interface similar to that of the parser combinators, which constructs an abstract representation of a grammar in our framework. This representation can then be analyzed, transformed, and ultimately turned into a parsing function.

[5] Using a 2 GHz Intel Core 2 Duo CPU (32-bit) and 2 GB RAM.

Another area we are investigating is the development of a library of grammar transformations, such as removal of ϵ-productions, removal of unreachable productions, or left-factoring.

Currently, the generation of a parser for a grammar makes use of top-down, backtracking parser combinators, which leads to very inefficient parsers. However, from our grammar representation, we can generate many kinds of parsers with various parsing strategies. In particular, our grammar representation is well suited to the kind of global grammar analysis normally performed by standalone parser generators, so we intend to explore the possibility of generating efficient, deterministic bottom-up parsers for our grammars.

Finally, we wish to prove more properties about grammar transformations. For instance, we want to expand the proof of the language-inclusion property into a full correctness proof of our implementation of the LCT. We hope that by doing more proofs, recurring proof patterns for proofs over grammars will emerge that we can then include in our general framework.

Acknowledgements. This work was partly supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and carried out while the second author was employed at Utrecht University. The authors thank the anonymous reviewers for their helpful remarks and constructive suggestions.

References

1. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming 2, 323–343 (1992)
2. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 184–207. Springer, Heidelberg (1996)
3. Swierstra, S.D.: Combinator parsing: A short tutorial. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) ALFA. LNCS, vol. 5520, pp. 252–300. Springer, Heidelberg (2009)
4. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinator for the real world. Technical Report UU-CS-2001-035, Utrecht University (2001)
5. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
6. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
7. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars, Agda code (2010), `http://www.cs.uu.nl/~andres/DTG/`
8. Fokker, J.: Functional parsers. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 1–23. Springer, Heidelberg (1995)
9. Rosenkrantz, D.J., Lewis, P.M.: Deterministic left corner parsing. In: Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory, pp. 139–152. IEEE, Los Alamitos (1970)
10. Johnson, M.: Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In: COLING-ACL, pp. 619–623 (1998)

11. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed grammars: The left corner transform. To appear in the proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009), York, England (March 29, 2009)
12. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (2000)
13. Pasălić, E., Linger, N.: Meta-programming with typed object-language representations. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 136–167. Springer, Heidelberg (2004)
14. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed abstract syntax. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, pp. 15–26. ACM Press, New York (2009)
15. Danielsson, N.A., Norell, U.: Structurally recursive descent parsing (2008), `http://www.cs.nott.ac.uk/~nad/publications/danielsson-norell-parser-parser-combinators.html`
16. Danielsson, N.A.: Total parser combinators (2009), `http://www.cs.nott.ac.uk/~nad/publications/danielsson-parser-combinators.html`
17. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)