Certified Normalization of Context-Free Grammars

Denis Firsov  Tarmo Uustalu
Institute of Cybernetics at TUT
{denis,tarmo}@cs.ioc.ee

Abstract
Every context-free grammar can be transformed into an equivalent one in the Chomsky normal form by a sequence of four transformations. In this work on formalization of language theory, we prove formally in the Agda dependently typed programming language that each of these transformations is correct in the sense of making progress toward normality and preserving the language of the given grammar. Also, we show that the right sequence of these transformations leads to a grammar in the Chomsky normal form (since each next transformation preserves the normality properties established by the previous ones) that accepts the same language as the given grammar. As we work in a constructive setting, soundness and completeness proofs are functions converting between parse trees in the normalized and original grammars.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—correctness proofs; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems

Keywords certified programs; context-free grammars; Chomsky normal form; normalization; dependently typed programming; Agda

1. Introduction
In formal language theory, a context-free grammar (CFG) is said to be in the Chomsky normal form (CNF), if all of its production rules are of the form: \( A \rightarrow BC \), \( A \rightarrow a \), or \( S \rightarrow \varepsilon \), where \( A, B \) and \( C \) are nonterminals, \( a \) is a terminal, \( S \) is the start nonterminal. Also, neither \( B \) nor \( C \) may be the start nonterminal.

Context-free grammars in the Chomsky normal form are very convenient to work with. It is often assumed that either CFGs are given in CNF from the beginning or there is an intermediate step of normalization. For example, Minamide [8] has implemented and proved correct three sophisticated decision procedures for context-free languages specified by CNF grammars:

- inclusion between a context-free language and a regular language;
- inclusion between a context-free language and an ordinary language.

Having a certified implementation of normalization for CFGs enables us to lift these decision procedures to context-free languages defined by CFGs in general form without losing the guarantees of correctness.

Another example is our previous work [3], where we reported on a certified implementation of the Cocke–Younger–Kasami (CYK) parsing algorithm in the Agda dependently typed programming language [9]. The CYK algorithm works only with grammars in the Chomsky normal form. Now, with a certified implementation of the CFG normalization algorithm we extend the reach of this work. Namely, to parse a string \( a \) for some general CFG \( G \) we could proceed as follows:

1. normalize \( G \) into a CNF \( G' \);
2. parse \( a \) by using the certified implementation of the CYK algorithm and get a parse tree \( t \) for the grammar \( G' \);
3. finally, convert the parse tree \( t \) for the grammar \( G' \) to a parse tree \( t' \) for the grammar \( G \) with the constructive soundness proof of normalization of \( G \) (which is a function from parse trees to parse trees).

Both examples demonstrate how certified normalization enables us to adopt certified development from CNF grammars to general CFGs retaining the correctness guarantees.

The full normalization transformation for a CFG is the composition of the following constituent transformations [1]:

1. elimination of all \( \varepsilon \)-rules (i.e., rules of the form \( A \rightarrow \varepsilon \)) (Section 3);
2. elimination all unit rules (i.e., rules of the form \( A \rightarrow B \)) (Section 4);
3. replacing all rules \( A \rightarrow X_1X_2 \ldots X_k \) where \( k \geq 3 \) with rules \( A \rightarrow X_1A_1 \), \( A_1 \rightarrow X_2A_2 \), \( A_{k-2} \rightarrow X_{k-1}A_k \) where \( A_i \) are “fresh” nonterminals (Section 5.1);
4. for each terminal \( a \), adding a new rule \( A \rightarrow a \) where \( A \) is a fresh nonterminal and replacing \( a \) in the right-hand sides of all rules with length at least two with \( A \) (Section 5.2).

The algorithms for the first, third and fourth transformations are functional versions of the classical imperative algorithms described, e.g., in [11]. The approach to eliminating unit rules is a little different and is designed to support certified development (uses a recursion that is easily presented as wellfounded).

We prove the correctness of this normalization transformation by showing that a given CFG and the corresponding CNF grammar accept the same language. Because we work in a constructive framework, the proof consists of total functions converting parse trees of the normalized grammar to the given grammar (soundness) and in the converse direction (completeness) (Section 6).
We used Agda 2.4.2 and Agda Standard Library 0.8.1 for this development. The full Agda code of this paper can be found at [http://cs.ioc.ee/~denis/cert-norm/](http://cs.ioc.ee/~denis/cert-norm/).

2. Setup

We assume that \( N \) and \( T \) are some fixed types for nonterminals and terminals respectively. We only require \( N \) and \( T \) to have decidable equality. Symbols are terminals and nonterminals. A rule is defined as a pair of a nonterminal and a list of symbols. We also define some handy abbreviations:

\[
\text{data Symbol} : \text{Set} \ 
\text{where} \\
\text{nt} : N \rightarrow \text{Symbol} \\
\text{tm} : T \rightarrow \text{Symbol} \\
\]

\( \text{RHS} = \text{List Symbol} \)

\[
\text{data Rule} : \text{Set} \ 
\text{where} \\
\_\rightarrow \_ : N \rightarrow \text{RHS} \rightarrow \text{Rule} \\
\]

\( \text{Rules} = \text{List Rule} \)

\[
\text{Ts} : \text{Rules} \rightarrow \text{List T} \\
\text{Ts Rs} = \{ a | A \rightarrow rhs \in \text{Rs}, \text{tm} a \in \text{rhs} \} \\
\]

\( \text{NTs} : \text{Rules} \rightarrow \text{List N} \)

\[
\text{NTs Rs} = \{ A | A \rightarrow rhs \in \text{Rs} \} \cup \\
\{} B | A \rightarrow rhs \in \text{Rs}, \text{nt} B \in \text{rhs} \} \\
\]

\( \text{String} = \text{List T} \)

(To avoid notational clutter, in the paper we employ an easy-to-read unofficial list comprehension syntax.)

For now and for most of the paper, we assume that a grammar is just a list of rules, we do not assume a fixed start nonterminal. In Section 6.2, we define a grammar as a list of rules together with a designated start nonterminal.

The datatype of the parse trees (abstract syntax trees) is parametrized by a grammar \( \text{Rs} \) and is defined inductively as follows:

mutual

\[
\text{data Tree} (\text{Rs} : \text{Rules}) : N \rightarrow \text{String} \rightarrow \text{Set} \ 
\text{where} \\
\text{node} : \{ A : N \} \{ \text{rhs} : \text{RHS} \} \{ s : \text{String} \} \\
\rightarrow A \rightarrow rhs \in \text{Rs} \\
\rightarrow \text{Forest} \text{Rs} \text{rhs} s \rightarrow \text{Tree} \text{Rs} A s \\
\]

\[
\text{data Forest} (\text{Rs} : \text{Rules}) : \ 
\text{RHS} \rightarrow \text{String} \rightarrow \text{Set} \ 
\text{where} \\
\text{empty} : \text{Forest} \text{Rs} [ ] [ ] \\
\_\_ : \{ \text{rhs} : \text{RHS} \} \{ s : \text{String} \} \\
\rightarrow (t : T) \rightarrow \text{Forest} \text{Rs} \text{rhs} s \\
\rightarrow \text{Forest} \text{Rs} (\text{tm} t :: \text{rhs})(t :: s) \\
\_\_ : \{ \text{rhs} : \text{RHS} \} \{ s_1, s_2 : \text{String} \} \{ A : N \} \\
\rightarrow \text{Tree} \text{Rs} A s_1 \rightarrow \text{Forest} \text{Rs} \text{rhs} s_2 \\
\rightarrow \text{Forest} \text{Rs} (\text{nt} A :: \text{rhs})(s_1 :: s_2) \\
\]

(In Agda, an argument enclosed in curly braces is implicit. The Agda type checker will try to figure it out. If an argument cannot be inferred, it must be provided explicitly.)

In general, the type \( \text{Tree} \text{Rs} A \ a \) collects all parse trees for a string \( a \) for a grammar \( \text{Rs} \) and a nonterminal \( A \) at the root. The auxiliary type \( \text{Forest} \text{Rs} \text{rhs} \ a \) collects all parse forests for a string \( a \) whose constituent individual parse trees are rooted at the symbols in \( \text{rhs} \).

Let us look at the following example. Consider the following grammar \( \text{Rs} \) with two rules. Their proofs of membership in the grammar serve as names for these rules.

\[
\text{Rs} : \text{Rules} \\
\text{Rs} = [ S \rightarrow [ nt S , \text{tm} '+' , nt S ], \\
S \rightarrow [ \text{tm} '1' ] ] \\
\]

\( \text{fr} : S \rightarrow [ nt S , \text{tm} '+' , nt S ] \in \text{Rs} \\
\text{ar} : S \rightarrow [ \text{tm} '1' ] \in \text{Rs} \\
\]

The strings "1" and "1+1" have the following unique derivations:

\( 1 \text{T} : \text{Tree} \text{Rs} S "1" \)

\( 1 \text{T} = \text{node} \text{fr} ( '1' :: :) \text{t} \text{empt}y) \)

\( 1+1 \text{T} : \text{Tree} \text{Rs} S "1+1" \)

\( 1+1 \text{T} = \text{node} \text{fr} (1 \text{T} :: a '1' :: :) \text{t} \text{empt}y) \)

But the string "1+1+1" has two derivations:

<table>
<thead>
<tr>
<th>String</th>
<th>Parse Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+1</td>
<td><img src="image1" alt="Parse Tree 1+1" /></td>
</tr>
<tr>
<td>1+1+1</td>
<td><img src="image2" alt="Parse Tree 1+1+1" /></td>
</tr>
</tbody>
</table>

3. \( \varepsilon \)-rule elimination and its correctness

The main consequence of the presence of \( \varepsilon \)-rules in a grammar is that parse trees for the empty string can be constructed for some nonterminals. A nonterminal \( A \) is called nullable for a grammar \( \text{Rs} \), if one can construct a parse tree for the empty string with \( A \) at the root, i.e., an inhabitant of the type \( \text{Tree} \text{Rs} A [ ] \). We describe the transformation of \( \varepsilon \)-rule elimination:

1. find all nullable nonterminals;
2. for each rule with some nullable nonterminals in its right-hand side \( \text{rhs} \), add a set of new rules given by all subsequences of \( \text{rhs} \) obtained by dropping some nullable nonterminals;
3. remove every rule whose right-hand side is empty string.

For example, for the grammar

\[
S \rightarrow AbA \ | B \\
B \rightarrow b \ | c \\
A \rightarrow e \ | d \\
\]

the transformation produces the following grammar:

<table>
<thead>
<tr>
<th>String</th>
<th>Parse Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+1+1</td>
<td><img src="image3" alt="Parse Tree 1+1+1" /></td>
</tr>
</tbody>
</table>
In this section, we describe how to find all nullable nonterminals of a grammar. We use the following observation: if a nonterminal \( A \) is nullable, then there exists a rule \( A \rightarrow \text{rhs} \in \text{Rs} \) such that \( \text{rhs} \) consists only of nullable nonterminals (in particular, it is also possible that \( \text{rhs} \equiv [] \)). Therefore, to find all nullable nonterminals, we iteratively build all trees for the empty string. Here is the algorithm:

\[
\begin{align*}
nlbls : \text{Rules} & \rightarrow \text{N} \rightarrow \text{List N} \\
nlbls \text{ Rs zero} & = \text{start} \\
nlbls \text{ Rs (suc n)} & = \text{collect (nlbls Rs n)} \\
\text{where} & \\
\text{start} & = \{ A \mid A \rightarrow [] \in \text{Rs} \} \\
\text{collect ans} & = \{ A \mid A \rightarrow \text{rhs} \in \text{Rs} , \\
& \quad (B : \text{N}) \rightarrow \text{nt} B \in \text{rhs} \rightarrow B \in \text{ans} \}
\end{align*}
\]

Clearly, the algorithm is sound (by construction):

\[
\begin{align*}
nlbls \text{-snd} : (\text{Rs} : \text{Rules}) & \rightarrow (A : \text{N}) \rightarrow (n : \text{N}) \\
& \rightarrow A \in \text{nlbls Rs n} \rightarrow \text{Tree Rs A []}
\end{align*}
\]

But how many iterations do we need for the completeness? Let us look at a weak version of completeness:

\[
\begin{align*}
nlbls\text{-cmplt-weak} : (\text{Rs} : \text{Rules}) & \rightarrow (A : \text{N}) \rightarrow (n : \text{N}) \\
& \rightarrow (t : \text{Tree Rs A []}) \rightarrow A \in \text{nlbls Rs n} \rightarrow \text{Tree Rs A []}
\end{align*}
\]

\[
\text{Proof} \quad \text{If height } t \leq \text{length } \text{Rs}, \text{then the theorem is proved by } \text{nlbls-cmplt-weak.} \quad \text{If height } t > \text{length } \text{Rs}, \text{then there exists} \quad \text{at least one branch in the parse tree with} \quad \text{at least one rule used} \quad \text{twice. Suppose this rule is } r : B \rightarrow \text{rhs} \in \text{Rs}. \text{Next, let the subtrees rooted at the left-hand nonterminal B of the rule } r \quad \text{be} \quad t \text{ and } t', \quad \text{both } t \text{ and } t' \text{ have type} \text{ Tree Rs B []}. \text{Therefore, we can substitute } t' \text{ for } t \text{ and still get a parse tree of type } \text{Tree Rs A []}. \text{This procedure can be repeated} \quad \text{until height } t \leq \text{length } \text{Rs}. \text{Finally, we define an abbreviation:}
\]

\[
\begin{align*}
nullables : \text{Rules} & \rightarrow \text{List N} \\
nullables \text{ Rs} & = \text{nlbls Rs (length Rs)}
\end{align*}
\]

3.2 Subsequences

In this section, we describe how to compute certain subsequences of a list. More precisely, given some list \( x :: \text{List X} \) and some predicate \( P : \text{X} \rightarrow \text{Bool} \), we would like to compute all subsequences of \( x \) obtainable by dropping some elements satisfying \( P \).

3.3 \( e \)-rule elimination

Finally, to eliminate \( e \)-rules, we combine the \text{allSubSeq} and nullables:

\[
\text{norm-e} : \text{Rules} \rightarrow \text{Rules}
\]

\[
\begin{align*}
\text{norm-e} \text{ Rs} & = \{ A \rightarrow \text{rhs} \mid A \rightarrow \text{rhs} \in \text{Rs} , \\
& \quad \text{rhs} \in \text{nullables Rs} \text{ rhs} , \\
& \quad \text{rhs} \neq [] \}
\end{align*}
\]

First, we find all nullable nonterminals in the grammar. Then, for each rule \( A \rightarrow \text{rhs} \in \text{Rs} \), we compute all subsequences \( \text{rhs}' \) of \( \text{rhs} \) obtainable by dropping some nullable nonterminals in \( \text{rhs} \). Finally, for all nonempty \( \text{rhs}'s \), the rule \( A \rightarrow \text{rhs}' \) is added to the resulting grammar.

3.4 Correctness

\textbf{Progress} \quad \text{Observe that the function } \text{norm-e} \text{ explicitly excludes rules with empty right-hand sides. Therefore, it is simple to show that, for any grammar } \text{Ra}, \text{the normalized grammar } \text{norm-e} \text{ RRs has no } e\text{-rules:}

\[
\begin{align*}
\text{ne-progress} : (\text{Rs} : \text{Rules}) & \rightarrow (A : \text{N}) \\
& \rightarrow A \rightarrow [] \notin \text{norm-e} \text{ Rs}
\end{align*}
\]

\textbf{Soundness} \quad \text{Next, let us show soundness. Namely, given some tree in the normalized grammar } \text{Tree (norm-e Rs A s)} \text{ A s, we would like to construct a parse tree of s in the original grammar } \text{Rs:}

\[
\begin{align*}
\text{ne-snd} : (\text{Rs} : \text{Rules}) & \rightarrow (A : \text{N}) \rightarrow (s : \text{String}) \\
& \rightarrow \text{Tree (norm-e Rs A s} \rightarrow \text{Tree Rs A s}
\end{align*}
\]

\textbf{Proof} \quad \text{The proof is by induction on the height of } t : \text{Tree (norm-e Rs A s}. \text{ Pattern matching yields f : Forest (norm-e Rs) rhs s and r : A \rightarrow \text{rhs} \in \text{norm-e Rs} \text{ such that t \equiv node r f}. \text{ For each tree t' of type Tree (norm-e Rs) B s' such that t' \in f, by the induction hypothesis, we construct a tree Tree Rs B s'. Hence, we can construct f' : Forest Rs rhs s. Next, analyze the rule A \rightarrow \text{rhs}. If r' : A \rightarrow \text{rhs} \in \text{Rs}, then the proof is completed by the witness node r' f'. If A \rightarrow \text{rhs} \notin \text{Rs}, then by the definition of norm-e there exists some rule r' : A \rightarrow \text{rhs}' \in \text{Rs} such that r' \in \text{allSubSeq (nullables Rs) rhs}'. By soundness of allSubSeq, the list rhs is a subsequence of rhs'. Moreover, the nonterminals of all removed positions in rhs' are contained in nullables Rs. Therefore, the proof can be completed by the witness node r' f'' : Tree Rs A s, where f'' is constructed.
from $f'$ by putting trees for the empty string (produced by using soundness of nullables) at the dropped positions of $\mathrm{rhs}'$.

**Completeness** Conversely, given a parse tree for some non-empty string (recall that $\mathrm{norm-e}$ makes all nonterminals non-nulleable) in the original grammar, we can convert it into a parse tree in the normalized grammar:

\[
\begin{align*}
\mathrm{ne-cmplt} : (\mathrm{Rs} : \mathrm{Rules}) & \to (A : N) \to (s : \mathrm{String}) \\
& \to \mathrm{Tree} \, \mathrm{Rs} \, A \, s \to s \not\equiv \[] \to \mathrm{Tree} \, (\mathrm{norm-e} \, \mathrm{Rs}) \, A \, s
\end{align*}
\]

**Proof** The proof is by induction on the height of the parse tree $t : \mathrm{Tree} \, \mathrm{Rs} \, A \, s$. By pattern matching, we have $t \equiv \mathrm{node} \, r \, f$ where $f : \mathrm{Forest} \, \mathrm{Rs} \, \mathrm{rhs} \, s$ and $r : A \to \mathrm{rhs} \in \mathrm{Rs}$. For each tree $t'$ : $\mathrm{Tree} \, \mathrm{Rs} \, B \, s'$ such that $t' \in f$, let us analyze the possible cases:

* If $s' \not\equiv \[]$, then by the induction hypothesis, we can construct $\mathrm{Tree} \, (\mathrm{norm-e} \, \mathrm{Rs}) \, A \, s'$.
* If $s' \equiv \[]$, then by $\mathrm{nlbls-cmplt}$, we have that $B \in \mathrm{nullables} \, \mathrm{Rs}$.

Therefore, $f' : \mathrm{Forest} \, (\mathrm{norm-e} \, \mathrm{Rs}) \, \mathrm{rhs}' \, s$ can be constructed where $\mathrm{rhs}'$ is a subsequence of $\mathrm{rhs}$ (the positions at which $f : \mathrm{Forest} \, \mathrm{Rs} \, \mathrm{rhs} \, s$ contains trees for the empty string are skipped). If $\mathrm{rhs}' \not\equiv \[]$, then by completeness of nullables and allSubSeq, we get that $r' : A \to \mathrm{rhs}' \in \mathrm{norm-e} \, \mathrm{Rs}$ and the proof is completed by the witness node $r' \, f'$. If $\mathrm{rhs}' \equiv \[]$, then $s$ should be empty (all positions are nulled), but this contradicts the assumption that $s \not\equiv \[]$.

### 3.5 Example

Consider the following grammar $\mathrm{Rs}$:

\[
\begin{align*}
A & \to BCD \mid B \\
B & \to \epsilon \mid A \\
C & \to c \\
D & \to \epsilon
\end{align*}
\]

Since $B$ and $D$ are the only nullable nonterminals, the grammar $\mathrm{norm-e} \, \mathrm{Rs}$ has the following rules:

\[
\begin{align*}
A & \to BCD \mid B \mid CD \mid BC \mid C \\
B & \to A \\
C & \to c
\end{align*}
\]

The nonterminal $D$ in the grammar $\mathrm{norm-e} \, \mathrm{Rs}$ is nonproductive (i.e., $(s : \mathrm{String}) \to \mathrm{Tree} \, (\mathrm{norm-e} \, \mathrm{Rs}) \, D \, s \to \bot$). Let us look at the example of a tree $t$ for the original grammar $\mathrm{Rs}$ and its counterpart for the $\epsilon$-normalized grammar $\mathrm{norm-e} \, \mathrm{Rs}$:

\[
\begin{align*}
\text{A} & \quad \text{C} \\
\text{B} & \quad \text{D} \\
\text{A} & \\
\text{B} & \\
\epsilon & \text{c} & \epsilon
\end{align*}
\]

\[
\begin{align*}
t & : \mathrm{Tree} \, \mathrm{Rs} \, A \, "c"
\end{align*}
\]

### 4. Unit rule elimination and its correctness

#### 4.1 Implementation

We describe in list comprehension notation how unit rules with a particular right-hand nonterminal are eliminated:

\[
\begin{align*}
\mathrm{nu-step} : \mathrm{Rules} & \to N \to \mathrm{Rules} \\
\mathrm{nu-step} \, \mathrm{Rs} \, A & = \{ \mathrm{rule}' | \mathrm{rule} \in \mathrm{Rs}, \mathrm{rule}' \in \mathrm{step-f} \, \mathrm{Rs} \, A \, \mathrm{rule} \} \\
\mathrm{step-f} : \mathrm{Rules} & \to N \to \mathrm{Rule} \to \mathrm{Rules} \\
\mathrm{step-f} \, \mathrm{Rs} \, A \, (B \to \mathrm{rhs}) & = \\
& \quad \text{if} \, \mathrm{rhs} \equiv \[ \, \mathrm{nt} \, A \, \] \, \text{then} \\
& \quad \{ \, B \to \mathrm{rhs}' \mid A \to \mathrm{rhs}' \in \mathrm{norm-e} \, \mathrm{Rs}, \mathrm{rhs}' \not\equiv \[ \, \mathrm{nt} \, A \, \] \} \\
& \quad \text{else} \, \{ \, B \to \mathrm{rhs} \, \}
\end{align*}
\]

Compared to the grammar $\mathrm{Rs}$, in the grammar $\mathrm{nu-step} \, \mathrm{Rs} \, A$ every rule of the form $B \to [ \, \mathrm{nt} \, A \, ]$ is replaced with all rules of the form $B \to \mathrm{rhs}'$, where $\mathrm{rhs}'$ stands for a right-hand side such that $A \to \mathrm{rhs}' \in \mathrm{Rs}$ and $\mathrm{rhs}' \not\equiv \[ \, \mathrm{nt} \, A \, \]$. Now, full unit rule elimination is achieved by applying this procedure to all nonterminals:

\[
\begin{align*}
\mathrm{norm-u} : \mathrm{Rules} & \to \mathrm{Rules} \\
\mathrm{norm-u} \, \mathrm{Rs} & = \mathrm{foldl} \, \mathrm{nu-step} \, \mathrm{Rs} \, (\mathrm{NTs} \, \mathrm{Rs})
\end{align*}
\]

Recall that $\mathrm{NTs} \, \mathrm{Rs}$ is an enumeration of all nonterminals appearing in the grammar $\mathrm{Rs}$.

#### 4.2 Correctness

**Progress** First, we show that $\mathrm{nu-step}$ gains some progress:

\[
\begin{align*}
\mathrm{nu-step-progress} : (\mathrm{Rs} : \mathrm{Rules}) & \to (A \mid B : N) \\
& \to A \to [ \, \mathrm{nt} \, B \, ] \not\in \mathrm{nu-step} \, \mathrm{Rs} \, B
\end{align*}
\]

This lemma states that there is no rule with the right-hand side $[ \, \mathrm{nt} \, B \, ]$ in the grammar $\mathrm{nu-step} \, \mathrm{Rs} \, B$. The progress lemma for the $\mathrm{norm-u}$ is a trivial consequence:

\[
\begin{align*}
\mathrm{nu-progress} : (\mathrm{Rs} : \mathrm{Rules}) & \to (A \mid B : N) \\
& \to A \to [ \, \mathrm{nt} \, B \, ] \not\in \mathrm{norm-u} \, \mathrm{Rs}
\end{align*}
\]

\[
\begin{align*}
\text{A} & \quad \text{C} \\
\text{B} & \quad \text{D} \\
\text{A} & \\
\text{B} & \\
\epsilon & \text{c} & \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{ne-cmplt} \, t & : \mathrm{Tree} \, (\mathrm{norm-e} \, \mathrm{Rs}) \, A \, "c"
\end{align*}
\]
Soundness  We start by proving a lemma about possible shapes of rules in the original grammar:

\[
\text{nu-sound-main} : (Rs : \text{Rules}) \rightarrow (A \; B : N) \\
\rightarrow (\text{rhs} : \text{RHS}) \rightarrow A \rightarrow \text{rhs} \in \text{nu-step Rs B} \\
\rightarrow A \rightarrow \text{rhs} \in \text{Rs} \\
\vee (A \rightarrow [nt \; B] \in \text{Rs} \times B \rightarrow \text{rhs} \in \text{Rs})
\]

This lemma shows that, if a rule \(A \rightarrow \text{rhs}\) belongs to a normalized grammar nu-step Rs B, then either the rule \(A \rightarrow \text{rhs}\) belongs to Rs or the rules \(A \rightarrow [nt \; B]\) and \(B \rightarrow \text{rhs}\) do.

Now, we show how soundness follows from \text{nu-sound-main}:

\[
\text{nu-step-sound} : (Rs : \text{Rules}) \rightarrow (A \; B : N) \\
\rightarrow (s : \text{String}) \\
\rightarrow \text{Tree (nu-step Rs B) A s} \rightarrow \text{Tree Rs A s}
\]

Proof  The proof is by induction on the height of the tree \(t : \text{Tree (nu-step Rs B) A s}\). Pattern matching on \(t\) yields some \(f\) of type Forest (nu-step Rs B) rhs s and \(p \rightarrow \text{rhs} \in (\text{nu-step Rs B})\) such that \(t \equiv \text{node p f}\). Next, for all trees \(t' : \text{Tree (nu-step Rs B) C s'}\) such that \(t' \in f\), by the induction hypothesis, we turn \(t'\) into \(t'' : \text{Tree Rs C s'}\). Therefore, by induction on the length of \(f\), a forest \(f' : \text{Forest Rs rhs s}\) can be constructed. Finally, by \text{nu-sound-main} we have two cases:

\* \(p' : A \rightarrow \text{rhs} \in \text{Rs}\). Then the proof is completed by constructing the witness node \(p' f' : \text{Tree Rs A s}\).

\* \(p' : A \rightarrow [nt \; B] \in \text{Rs}\) and \(p'' : B \rightarrow \text{rhs} \in \text{Rs}\). Then the proof is completed by the giving the witness node \((\text{node p'' f'')} : ; \text{empty}\) which has type Tree Rs A s.

Soundness of norm-u follows trivially from \text{nu-step-sound}:

\[
\text{nu-snd} : (Rs : \text{Rules}) \rightarrow (A : N) \rightarrow (s : \text{String}) \\
\rightarrow \text{Tree (norm-u Rs) A s} \rightarrow \text{Tree Rs A s}
\]

Completeness  We start again by observing special properties:

\[
\text{nu-cmlt'} : (Rs : \text{Rules}) \rightarrow (A \; B : N) \\
\rightarrow (\text{rhs} : \text{RHS}) \rightarrow A \rightarrow [nt \; B] \in \text{Rs} \\
\rightarrow B \rightarrow \text{rhs} \in \text{Rs} \rightarrow \text{rhs} \equiv [nt \; B] \\
\rightarrow A \rightarrow \text{rhs} \in \text{nu-step Rs B}
\]

\[
\text{nu-cmlt''} : (Rs : \text{Rules}) \rightarrow (A \; B : N) \\
\rightarrow (\text{rhs} : \text{RHS}) \rightarrow A \rightarrow \text{rhs} \in \text{Rs} \\
\rightarrow \text{rhs} \equiv [nt \; B] \rightarrow A \rightarrow \text{rhs} \in \text{nu-step Rs B}
\]

The \text{nu-cmlt'} lemma states that, if rules \(A \rightarrow [nt \; B]\) and \(B \rightarrow \text{rhs}\) belong to Rs and \(\text{rhs} \equiv [nt \; B]\), then the rule \(A \rightarrow \text{rhs}\) belongs to the normalized grammar nu-step Rs B. At the same time the lemma \text{nu-cmlt''} establishes that rules \(A \rightarrow \text{rhs}\) where \(\text{rhs} \equiv [nt \; B]\) will stay in the normalized grammar.

Using this property, completeness is proved by induction on a given parse tree and inspection of rules at two consecutive levels.

\[
\text{nu-step-complete} : (Rs : \text{Rules}) \\
\rightarrow (A \; B : N) \rightarrow (s : \text{String}) \\
\rightarrow \text{Tree Rs A s} \rightarrow \text{Tree (nu-step Rs B) A s}
\]

Proof  The claim is proved by induction on the height of the tree \(t\) of type Tree Rs A s. Pattern matching on \(t\) yields some \(f : \text{Forest Rs rhs s}\) and \(p : A \rightarrow \text{rhs} \in \text{Rs}\) such that \(t \equiv \text{node p f}\). Next, for all trees \(t' : \text{Tree Rs C s'}\) such that \(t' \in f\), by the induction hypothesis, we construct \(t'' : \text{Tree (nu-step Rs B) C s'}\). So, by induction on the length of \(f\), we get \(f' : \text{Forest (nu-step Rs B) rhs s}\). Finally, let us analyze two cases:

\* If \(\text{rhs} \neq [nt \; B]\), then by \text{nu-cmlt''} we have \(p' : A \rightarrow \text{rhs} \in \text{nu-step Rs B}\) and proof is finished by the witness node \(p' f' : \text{Tree (nu-step Rs B) A s}\).

\* If \(\text{rhs} \equiv [nt \; B]\), then the previously constructed \(f'\) satisfies \(f' \equiv (\text{node q f''}) :: ;;\) empty where \(f''\) is of type Forest (nu-step Rs B) rhs s and \(q\) is of type B \(\rightarrow \text{rhs'}\) in nu-step Rs B. By nu-step-progress we know that \(\text{rhs'} \neq [nt \; B]\), therefore, by \text{nu-cmlt'} we get \(p' : A \rightarrow \text{rhs'} \in \text{nu-step Rs B}\). Finally, the witness node \(p' f'\) of type Tree (nu-step Rs B) A s concludes the proof.

And lifting this result to the full elimination of unit rules:

\[
\text{nu-cmlt} : (Rs : \text{Rules}) \rightarrow (A : N) \rightarrow (s : \text{String}) \\
\rightarrow \text{Tree Rs A s} \rightarrow \text{Tree (norm-u Rs) A s}
\]

4.3 Example  

Consider the grammar

\[
A \rightarrow CA \mid B \mid a \\
B \rightarrow b \mid A \\
C \rightarrow BA
\]

After the norm-u transformation we have:

\[
A \rightarrow CA \mid a \mid b \\
B \rightarrow b \mid CA \mid a \\
C \rightarrow BA
\]

Observe how an example tree for the original grammar is transformed into a tree for the normalized grammar:

Mapping this tree back from the normalized grammar to the original grammar gives a tree with the unit loop cut out:

```plaintext
nu-cmlt t : Tree (norm-u Rs) A "bab"
```
5. Final transformations

5.1 Long right-hand sides

Next, we describe how to eliminate rules \( A \rightarrow \text{rhs} \) where \( \text{length \ rhs} > 2 \)—so-called \emph{long rules}.

To do so, we first need a function that will supply fresh nonterminals,

\[
\text{newnt : Rules} \rightarrow N
\]

and a proof that \text{newnt \ Rs} does not occur anywhere in the grammar \( \text{Rs} \).

\[
\text{newnt-lem : (Rs : Rules)} \rightarrow \text{newnt \ Rs} \notin \text{NTs Rs}
\]

The above states that \text{newnt \ Rs} is a “fresh” nonterminal. Note that there are no side effects involved here, the expression \text{newnt \ Rs} always returns the same nonterminal. Hence, to get the next “fresh” nonterminal, one must first embed the current one in the grammar.

For an explanation, assume that \( N = T = N \). Then, let us define \text{newnt} and \text{Rs} as follows:

\[
\begin{align*}
\text{newnt : Rules} & \rightarrow N \\
\text{newnt Rs} & = 1 + \max(\text{NTs Rs})
\end{align*}
\]

\[
\begin{align*}
\text{Rs} & : \text{Rules} \\
\text{Rs} & = [ 1 \rightarrow [ \text{nt \ 2, tm \ 3, nt \ 4 } ] ]
\end{align*}
\]

\[
\begin{align*}
\text{Rs'} & : \text{Rules} \\
\text{Rs'} & = 1 \rightarrow [ \text{nt \ (newnt \ Rs)} ] :: \text{Rs}
\end{align*}
\]

In that case, \text{newnt \ Rs} \equiv 5. Also, if we define \text{Rs'} by adding the rule \( 1 \rightarrow [ \text{nt \ (newnt \ Rs)} ] \) to \text{Rs}, then \text{newnt \ Rs'} \equiv 6.

Next, we are ready to define a step of normalization:

\[
\begin{align*}
\text{nl-step' : Rules} & \rightarrow N \rightarrow \text{Rules} \\
\text{nl-step'} & ((A \rightarrow X :: Y :: Z :: \text{rhs}) :: \text{Rs}) F = \\
& (A \rightarrow \text{nt \ F} :: Z :: \text{rhs}) :: \\
& (F \rightarrow X :: Y :: [\text{}]) :: \text{Rs}
\end{align*}
\]

\[
\begin{align*}
\text{nl-step' & ((A \rightarrow \text{rhs}) :: \text{Rs}) F = } \\
& (A \rightarrow \text{rhs}) :: \text{nl-step' \ Rs F}
\end{align*}
\]

\[
\begin{align*}
\text{nl-step : Rules} & \rightarrow \text{Rules} \\
\text{nl-step \ Rs} & = \text{nl-step' \ Rs \ (newnt \ Rs)}
\end{align*}
\]

The function \text{nl-step} looks for the first rule of the form \( A \rightarrow X :: Y :: Z :: \text{rhs} \) and replaces it with rules \( A \rightarrow \text{nt \ F} :: Z :: \text{rhs} \) and \( F \rightarrow X :: Y :: [\text{}] \) where \( \text{F} \) is fresh.

After applying the function \text{nl-step} to the grammar \( \text{Rs} \), the sum of the lengths of the right-hand sides of all long rules decreases. This will be the measure of how many times \text{nl-step} needs to be applied to the grammar \( \text{Rs} \).

\[
\begin{align*}
\text{nl-measure : Rules} & \rightarrow N \\
\text{nl-measure \ Rs} & = \text{sum lengths where} \\
& \text{lengths} = \{ \text{length \ rhs} | A \rightarrow \text{rhs} \in \text{Rs}, \\
& \text{length \ rhs} > 2 \}
\end{align*}
\]

So, to eliminate all long rules, we apply the function \text{nl-step} to the set of rules (\text{nl-measure \ Rs}) times.

\[
\begin{align*}
\text{norm-l} & : \text{Rules} \rightarrow \text{Rules} \\
\text{norm-l \ Rs} & = \text{fold \ Rs \ nl-step \ (nl-measure \ Rs)}
\end{align*}
\]

5.2 Right-hand sides containing terminals

In what follows, we describe how to eliminate rules \( A \rightarrow \text{rhs} \) where \( \text{rhs} \) contains terminals and \( \text{length \ rhs} > 1 \).

The function \text{nt-step \ Rs} \ a \ adds to the grammar \( \text{Rs} \) the rule \( \text{newnt \ Rs} \rightarrow \text{tm} \ a \) and substitutes the symbol \( \text{tm} \ a \) with the symbol \( \text{nt} \ (\text{newnt \ Rs}) \) in the right-hand side of every rule whose right-hand side is longer than 1.

\[
\begin{align*}
\text{nt-step} & : \text{Rules} \rightarrow T \rightarrow \text{Rules} \\
\text{nt-step \ Rs \ a} & = \text{let} \ F = \text{newnt \ Rs \ in} \\
& F \rightarrow \text{tm} \ a :: \\
& \{ A \rightarrow \text{subst \ (tm} \ a) \ (\text{nt} \ F) \ \text{rhs} \ | \ A \rightarrow \text{rhs} \in \text{Rs} \}
\end{align*}
\]

\[
\begin{align*}
\text{where} \\
\text{subst} & : \text{Symbol} \rightarrow \text{Symbol} \rightarrow \text{RHS} \rightarrow \text{RHS} \\
\text{subst \ X Y \ rhs} & = \\
& \text{if} \ \text{length \ rhs} \leq 1 \\
& \text{then} \ \text{rhs} \\
& \text{else \ map} \ (\text{A Z} \rightarrow \text{if} \ Z \equiv X \ \text{then} \ Y \ \text{else} \ Z) \ \text{rhs}
\end{align*}
\]

Finally, remove all terminals from right-hand sides longer than 1 by folding \( \text{Rs} \) with the function \text{nt-step}:

\[
\begin{align*}
\text{norm-t} & : \text{Rules} \rightarrow \text{Rules} \\
\text{norm-t \ Rs} & = \text{foldl} \ \text{nt-step \ Rs \ (Ts \ Rs)}
\end{align*}
\]

5.3 Correctness of final transformations

Correctness of both \text{norm-t} and \text{norm-l} is rather obvious due to the simple nature of these transformations. But we still state the correctness theorems to highlight the side conditions and progress claims (the details of the proofs could be found in the code).

The progress lemma for \text{norm-l} states that after the transformation there are no rules with right-hand sides of more than two symbols.

\[
\begin{align*}
\text{nl-progress} & : (\text{Rs} : \text{Rules}) \rightarrow (A : N) \\
& \rightarrow (\text{rhs} : \text{RHS}) \rightarrow A \rightarrow \text{rhs} \in \text{norm-l \ Rs} \\
& \rightarrow \text{length} \ \text{rhs} \leq 2
\end{align*}
\]

The progress lemma for \text{norm-t} states that, for any \( \text{Rs} \) for all rules \( A \rightarrow \text{rhs} \in \text{norm-t \ Rs} \), either \( \text{rhs} \equiv \{ \text{tm} \ a \} \) for some terminal \( a \) or \( \text{rhs} \) consists of nonterminals only.

\[
\begin{align*}
\text{nt-progress} & : (\text{Rs} : \text{Rules}) \rightarrow (A : N) \\
& \rightarrow (\text{rhs} : \text{RHS}) \rightarrow A \rightarrow \text{rhs} \in \text{norm-t \ Rs} \\
& \rightarrow \exists (a : T) \ \text{rhs} \equiv \{ \text{tm} \ a \} \lor \text{ntOnly} \ \text{rhs}
\end{align*}
\]

Next, \text{nl-snd} and \text{nt-snd} state that each tree for normalized grammar that is rooted by some nonterminal present in the original grammar can be transformed into a tree for the original grammar:

\[
\begin{align*}
\text{nl-snd} & : (\text{Rs} : \text{Rules}) \rightarrow (A : N) \\
& \rightarrow (s : \text{String}) \rightarrow A \in \text{NTs} \ \text{Rs} \\
& \rightarrow \text{Tree} \ (\text{norm-l \ Rs}) \ A \ s \rightarrow \text{Tree} \ \text{Rs} \ A \ s
\end{align*}
\]

\[
\begin{align*}
\text{nt-snd} & : (\text{Rs} : \text{Rules}) \rightarrow (A : N) \\
& \rightarrow (s : \text{String}) \rightarrow A \in \text{NTs} \ \text{Rs} \\
& \rightarrow \text{Tree} \ (\text{norm-t \ Rs}) \ A \ s \rightarrow \text{Tree} \ \text{Rs} \ A \ s
\end{align*}
\]
The side condition $A \in \text{NTs } Rs$ is important, because a tree rooted
by some “freshly” added nonterminal has no corresponding tree in
the original grammar, where the fresh nonterminal is not present.
Conversely, any parse tree for Rs could be mapped to parse trees
for norm-l Rs and norm-t Rs.

$$\text{nlt-cmplt : (Rs : Rules) } \rightarrow (A : N) \rightarrow (s : \text{String}) \rightarrow \text{Tree Rs A s } \rightarrow \text{Tree (norm-l Rs) A s}$$

$$\text{nt-cmplt : (Rs : Rules) } \rightarrow (A : N) \rightarrow (s : \text{String}) \rightarrow \text{Tree Rs A s } \rightarrow \text{Tree (norm-t Rs) A s}$$

6. Full normalization and correctness

6.1 Full normalization function

Finally, we are ready to define the full normalization function:

$$\text{norm} = \text{norm-u} \circ \text{norm-e} \circ \text{norm-t} \circ \text{norm-l}$$

The function norm is a composition of the four transformations
we have introduced. In the order that these transformations are
chained matters. For example, norm-e can add new unit rules, so
norm-u must be performed after norm-e.

Progress The question of progress of norm boils down to the
questions about preservation of the progress properties of individual
constituent transformations by those transformations that follow:

1. Since norm-t never increases the length of the right-hand side
of any rule, norm-t preserves the progress made by norm-l. We
prove that, if the right-hand side of every rule in Rs is shorter
that some $n : \mathbb{N}$, then the same holds for all rules in
norm-t Rs:

$$\text{nt-eft : (Rs : Rules) } \rightarrow (n : \mathbb{N}) \rightarrow ((A : N) \rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{Rs} \rightarrow \text{length rhs } \leq n)$$

$$\rightarrow (A : N) \rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{norm-t Rs} \rightarrow \text{length rhs } \leq n$$

2. We show that, if $A \rightarrow \text{rhs } \in \text{norm-e Rs}$, then rhs must
be a subsequence of some rhs' such that $A \rightarrow \text{rhs' } \in \text{Rs}$. Since
the progress properties of norm-l and norm-t are closed
under the subsequence relation, norm-e preserves the progress
achieved by norm-l and norm-t:

$$\text{ne-eft : (Rs : Rules) } \rightarrow (A : N) \rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{norm-e Rs} \rightarrow A \rightarrow \text{rhs } \notin \text{Rs} \rightarrow A \rightarrow \text{rhs' } \in \text{Rs} \times \text{nullables Rs (rhs' )}$$

3. Since norm-u does not introduce any new right-hand sides into
a grammar, it preserves the progress properties of all other
transformations. Formally, we prove that, if there is some predicate
that holds for all RHSs in the grammar Rs, then it will also
hold for all RHSs in the grammar norm-u:

$$\text{nu-eft : (P : RHS } \rightarrow \text{Set) } \rightarrow (Rs : Rules) \rightarrow ((A : N) \rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{Rs } \rightarrow P \text{ rhs}) \rightarrow (A : N) \rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{norm-u Rs} \rightarrow P \text{ rhs}$$

Finally, we show the following progress property of norm:

$$\text{progress : (Rs : Rules) } \rightarrow (A : N)$$

$$\rightarrow (\text{rhs : RHS}) \rightarrow A \rightarrow \text{rhs } \in \text{norm Rs} \rightarrow \exists (B C : N) \rightarrow \text{rhs } \equiv \{ \text{nt B, nt C }\} \lor \exists (a : \text{T}) \rightarrow \text{rhs } \equiv \{ \text{tm a }\}$$

It states that, for any rule $A \rightarrow \text{rhs } \in \text{norm Rs}$, either
rhs $\equiv \{ \text{nt B, nt C }\}$ for some nonterminals B and C or
rhs $\equiv \{ \text{tm a }\}$ for some terminal a.

Soundness To show soundness of norm, we only need to chain
the soundness results of the individual transformations:

$$\text{soundness : (Rs : Rules) } \rightarrow (A : N)$$

$$\rightarrow (\text{rhs : String}) \rightarrow A \in \text{NTs Rs} \rightarrow \text{Tree (norm Rs) A s } \rightarrow \text{Tree Rs A s}$$

Completeness As in the case of soundness, completeness of norm
is proved by chaining the completeness results of the small trans-
formations:

$$\text{completeness : (Rs : Rules) } \rightarrow (A : N)$$

$$\rightarrow (\text{rhs : String}) \rightarrow A \in \text{NTs Rs} \rightarrow \text{Tree (norm Rs) A s } \rightarrow \text{Tree Rs A s}$$

6.2 Grammars with a start nonterminal

Next, we define a context-free grammar as a set of rules with a fixed
start nonterminal:

$$\text{record Grammar : Set where field}$$

$$S : N$$

$$\text{Rs : Rules}$$

(Given some $G : \text{Grammar}$, we write $S \in G$ and $\text{Rs } \in G$ for projections
of the start terminal and the list of rules respectively.)

Next, the language of the grammar G is defined as:

$$\text{TreeS : Grammar } \rightarrow \text{String } \rightarrow \text{Set}$$

$$\text{TreeS G s } = \text{Tree (Rs G) (S G) s}$$

Next, we implement normalization of context-free grammars:

$$\text{normS : Grammar } \rightarrow \text{Grammar}$$

$$\text{normS G = record } \{$$

$$\text{S } = \text{S'};$$

$$\text{Rs } = \text{if S G } \in \text{nullables (Rs G) then S' } \rightarrow [] :: \text{Rs'}$$

$$\text{else Rs'}$$

$$\}$$

where

$$\text{S' } = \text{newnt (Rs G)}$$

$$\text{Rs' } = \text{norm ((S' } \rightarrow [ \text{nt (S G) }] :: \text{Rs G})}$$

To normalize a context-free grammar we have the following algo-
rithm:

1. Declare newnt (Rs G) as a new starting nonterminal.

2. Normalize the set of rules Rs G extended by the rule
newnt (Rs G) $\rightarrow [ \text{nt (S G) }]$. Since newnt (Rs G) is fresh, it is clear that its language is same as the language of
nonterminal S G and it will not affect the language of any other
nonterminal (this step guarantees that new starting nonterminal
does not appear on the right hand sides of the rules).

3. Finally, if the starting nonterminal of the original grammar
was nullable then add the rule newnt (Rs G) $\rightarrow []$ to the
normalized set of rules to retain the empty string in the language of
normalized grammar. Intuitively, it is safe to do so, because
new (Rs G) does not appear in the right-hand sides of the other
rules.

Let us look at the final versions of progress, soundness and
completeness properties:
Progress

\[\text{normS-progress : (G : Grammar) \rightarrow (A : N)}\]
\[\rightarrow (\text{rhs : RHS})\]
\[\rightarrow \text{let G' = normS G in A \rightarrow rhs \in Rs G'}\]
\[\rightarrow (\exists (B : C : N) \text{ rhs } \equiv [ \text{ nt B, nt C } ] \times \text{ B } \neq S G' \times \text{ C } \neq S G') \lor\]
\[\land (\exists (a : T) \text{ rhs } \equiv [ \text{ tm a } ] ) \lor\]
\[\text{ (rhs } \equiv [ ] \times A \equiv S G')\]

For any rule \(A \rightarrow \text{ rhs } \in Rs (\text{normS G})\), the right-hand side \(\text{ rhs}\) is either \([ \text{ nt B, nt C } ]\) for some nonterminals \(B\) and \(C\) where neither \(B\) nor \(C\) are starting nonterminals or \([ \text{ tm a } ]\) for some terminal \(a\), or \([ ]\) with the condition that \(A \equiv S (\text{normS G})\).

Soundness and completeness

\[\text{normS-snd : (G : Grammar) \rightarrow (s : String)}\]
\[\rightarrow S G \in \text{NTs (Rs G)}\]
\[\rightarrow \text{TreeS (normS G)} s \rightarrow \text{TreeS G s}\]

\[\text{normS-cmplt : (G : Grammar) \rightarrow (s : String)}\]
\[\rightarrow \text{TreeS G s} \rightarrow \text{TreeS (normS G)} s\]

If a given grammar is well-formed (i.e., the start nonterminal actually appears in the given list of rules), then normalization preserves the language of the grammar.

7. Related Work and Conclusions

While a number of authors have formalized various parts of the theory of regular grammars or expressions and finite automata, efforts in the direction of context-free grammars seem fewer.

Several authors have considered parsing of context-free grammars. Barthwal and Norrish [6] formalized SLR parsing with the HOL4 theorem prover. Ridge [10] has formalized the correctness of a general CFG parser constructor in HOL4.

Koprowski and Binskič [4] have formalized parsing expression grammars (PEGs), a formalism for specifying recursive descent parsers, in Coq. Jourdan, Pottier and Leroy [7] have presented a validator that checks if a context-free grammar and an LR(1) parser agree; they have proved the validator correct in Coq.


Regarding normalization of context-free grammars, Barthwal and Norrish [5] described a formalisation of the Chomsky and Greibach normal forms for context-free grammars with the HOL4 theorem prover. They showed how to solve the problems which arise from mechanising the straightforward pen and paper proofs. The non-constructive setting gave the advantage of the power of extensional and classical reasoning, but also the significant drawback that it did not deliver actual functions for normalizing grammars or converting parse trees between grammars.

We have proved in Agda that a general CFG and its Chomsky normal form accept the same language. As a program, the proof consists of functions for conversion of parse trees between the original and normalized grammars. This is a typical added benefit of formalization in a language like Agda: e.g., a proof that a CFG and the corresponding pushdown automaton accept the same language would give functions for conversion between parse trees and accepting runs.

Combined with the CYK parser we have written previously [8], the code of this paper gives us a parser for CFGs in general form. There is, however, a caveat: we do not get all parse trees of the grammar; moreover, it is not entirely obvious which parse trees we get and which are lost.

To make this precise, we plan to extend this work as follows. Instead of unnamed rules (a rule is identified by the left-hand non-terminal and the right-hand list of symbols), we name rules. This gives us finer control over parse trees. Now we expect that the conversion of a parse tree in the normalized grammar to the original grammar and back again will be identity while the conversion of a parse of the original grammar to the normalized grammar and back will be an idempotent function—a kind of normalizer of parse trees that truncates nullable paths and removes unit cycles. Normalization of parse trees by passing through the normalized grammar can then be seen as a form of normalization-by-evaluation.

Overall, the constructive approach allows one to give parse trees a first-class status: knowing that a string is in a language includes knowing a proof of this, i.e., a parse tree. These proofs become objects of analysis and manipulation.

Acknowledgement We thank our anonymous referees for the useful feedback. This research was supported by the ERDF funded Estonian CoE project EXCS and the Estonian Research council target-financed research theme No. 0140007s12 and grant No. 9475.

References