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: $\mathtt{A} \longrightarrow \mathtt{BC}$, $\mathtt{A} \longrightarrow \mathtt{a}$, or $\mathtt{S} \longrightarrow \varepsilon$, 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;
- balancedness of a context-free language;
- inclusion between a context-free language and a regular hedge 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 s for some general CFG G we could proceed as follows:

- normalize G into a CNF G';
- parse s by using the certified implementation of the CYK algorithm and get a parse tree t for the grammar G';
- 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 *ε-rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \varepsilon$) (Section 3);
2. elimination all *unit rules* (i.e., rules of the form $\mathtt{A} \longrightarrow \mathtt{B}$) (Section 4);
3. replacing all rules $\mathtt{A} \longrightarrow \mathtt{X}_1\mathtt{X}_2 \ \ldots \ \mathtt{X}_k$ where $\mathtt{k} \geq 3$ with rules $\mathtt{A} \longrightarrow \mathtt{X}_1\mathtt{A}_1$, $\mathtt{A}_1 \longrightarrow \mathtt{X}_2\mathtt{A}_2$, $\mathtt{A}_{k\text{-}2} \longrightarrow \mathtt{X}_{k\text{-}1}\mathtt{X}_k$ where $\mathtt{A}_i$ are "fresh" nonterminals (Section 5.1);
4. for each terminal a, adding a new rule $\mathtt{A} \longrightarrow \mathtt{a}$ 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 [1]. 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).

CPP '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693177

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/`.

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:

```
data Symbol : Set where
  nt : N → Symbol
  tm : T → Symbol

RHS = List Symbol

data Rule : Set where
  _⟶_ : N → RHS → Rule

Rules = List Rule

Ts : Rules → List T
Ts Rs = { a | A ⟶ rhs ∈ Rs, tm a ∈ rhs }

NTs : Rules → List N
NTs Rs = { A | A ⟶ rhs ∈ Rs } ∪
         { B | A ⟶ rhs ∈ Rs, nt B ∈ rhs }

String = 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 `Rs` and is defined inductively as follows:

```
mutual
  data Tree (Rs : Rules) : N → String → Set where
    node : {A : N}{rhs : RHS}{s : String}
        → A ⟶ rhs ∈ Rs
        → Forest Rs rhs s → Tree Rs A s

  data Forest (Rs : Rules) :
                 RHS → String → Set where
   empty  : Forest Rs [] []
   _::t_  : {rhs : RHS}{s : String}
        → (t : T) →  Forest Rs rhs s
        → Forest Rs (tm t :: rhs) (t :: s)
   _::n_  : {rhs : RHS}{s1 s2 : String}{A : N}
        → Tree Rs A s1 → Forest Rs rhs s2
        → Forest Rs (nt A :: rhs) (s1 ++ s2)
```

(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 `Tree Rs A s` collects all parse trees for a string `s` for a grammar `Rs` and a nonterminal `A` at the root. The auxiliary type `Forest Rs rhs s` collects all parse forests for a string `s` whose constituent individual parse trees are rooted at the symbols in `rhs`.

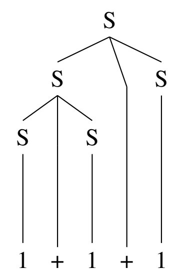
Let us look at the following example. Consider the following grammar `Rs` with two rules. Their proofs of membership in the grammar serve as names for these rules.

```
Rs : Rules
Rs = [ S ⟶ [ nt S , tm '+' , nt S ],
       S ⟶ [ tm '1' ] ]

fr : S ⟶ [ nt S , tm '+' , nt S ] ∈ Rs
sr : S ⟶ [ tm '1' ] ∈ Rs
```

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

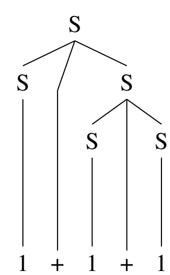
```
1T : Tree Rs S "1"
1T = node sr ('1' ::t empty)

1+1T : Tree Rs S "1+1"
1+1T = node fr (1T ::n '+' ::t 1T ::n empty)
```

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



`lft : Tree Rs S "1+1+1"`

```
lft : Tree Rs S "1+1+1"
lft = node fr (1+1T ::n '+' ::t 1T ::n empty)
```



`rgt : Tree Rs S "1+1+1"`

```
rgt : Tree Rs S "1+1+1"
rgt = node fr (1T ::n '+' ::t 1+1T ::n empty)
```

3. ε-rule elimination and its correctness

The main consequence of the presence of ε-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 `Rs`, if one can construct a parse tree for the empty string with `A` at the root, i.e., an inhabitant of the type `Tree Rs A []`. We describe the transformation of ε-rule elimination:

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

For example, for the grammar

```
S ⟶ AbA | B
B ⟶ b | c
A ⟶ ε | d
```

the transformation produces the following grammar:

```
S ⟶ AbA | Ab | bA | b | B
B ⟶ b | c
A ⟶ d
```

Note that the transformation makes all nonterminals non-nullable: for a nonterminal nullable for the given grammar, the language of this nonterminal in the transformed grammar differs from its language in the original grammar by the absence of the empty word.

3.1 Nullable nonterminals

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 ⟶ rhs ∈ Rs` such that `rhs` consists only of nullable nonterminals (in particular, it is also possible that `rhs ≡ []`). Therefore, to find all nullable nonterminals, we iteratively build all trees for the empty string. Here is the algorithm:

```
nlbls : Rules → ℕ → List N
nlbls Rs zero    = start
nlbls Rs (suc n) = collect (nlbls Rs n)
  where
   start = { A | A ⟶ [] ∈ Rs }
   collect ans
    = { A | A ⟶ rhs ∈ Rs ,
               (B : N) → nt B ∈ rhs → B ∈ ans }
```

Clearly, the algorithm is sound (by construction):

```
nlbls-snd : (Rs : Rules) → (A : N) → (n : ℕ)
               → A ∈ nlbls Rs n → Tree Rs A []
```

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

```
nlbls-cmplt-weak : (Rs : Rules) → (A : N)
  → (t : Tree Rs A []) → A ∈ nlbls Rs (height t)
```

By induction on the height of the parse tree, we can easily prove this lemma. But the lemma is too weak, because it depends on the height of the input parse tree and this is not bounded. We need to find a number of iterations that is sufficient for every possible parse tree.

We prove that `length Rs` (denotes the number of the rules in the grammar) many iterations is enough:

```
nlbls-cmplt : (Rs : Rules) → (A : N)
  → Tree Rs A [] → A ∈ nlbls Rs (length Rs)
```

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

Finally, we define an abbreviation:

```
nullables : Rules → List N
nullables Rs = nlbls Rs (length Rs)
```

3.2 Subsequences

In this section, we describe how to compute certain subsequences of a list. More precisely, given some list `xs : List X` and some predicate `P : X → Bool`, we would like to compute all subsequences of `xs` obtainable by dropping some elements satisfying `P`.

```
allSubSeq : {X : Set} → (X → Bool)
                      → List X → List (List X)
allSubSeq P xs
  = foldr (λ x res →
           if P x then res ++ (map (_::_ x) res)
                  else map (_::_ x) res)
          [ [] ] xs
```

For an explanation, let us look at the example:

```
allSubSeq (≡ nt A) [ nt A, nt B, nt A, nt C ] ⇒
  [ [ nt A, nt B, nt A, nt C ],
    [       nt B, nt A, nt C ],
    [ nt A, nt B,       nt C ],
    [       nt B,       nt C ] ]
```

To generate all subsequences of some list `xs`, one could call `allSubSeq (λ _ → true) xs`. The function `allSubSeq` function is sound and complete in the sense that it generates all desired subsequences and nothing else (a formalization can be found in our development).

3.3 ε-rule elimination

Finally, to eliminate ε-rules, we combine the `allSubSeq` and `nullables`:

```
norm-e : Rules → Rules
norm-e Rs = { A ⟶ rhs' | A ⟶ rhs ∈ Rs ,
    rhs' ∈ allSubSeq (∈ nullables Rs) rhs ,
    rhs' ≢ [] }
```

First, we find all nullable nonterminals in the grammar. Then, for each rule `A ⟶ rhs` in `Rs`, we compute all subsequences `rhs'` of `rhs` obtainable by dropping some nullable nonterminals in `rhs`. Finally, for all nonempty `rhs'`, the rule `A ⟶ rhs'` is added to the resulting grammar.

3.4 Correctness

Progress Observe that the function `norm-e` explicitly excludes rules with empty right-hand sides. Therefore, it is simple to show that, for any grammar `Rs`, the normalized grammar `norm-e Rs` has no ε-rules:

```
ne-progress : (Rs : Rules) → (A : N)
                           → A ⟶ [] ∉ norm-e Rs
```

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

```
ne-snd : (Rs : Rules) → (A : N) → (s : String)
           → Tree (norm-e Rs) A s → Tree Rs A s
```

Proof The proof is by induction on the height of `t : Tree (norm-e Rs) A s`. Pattern matching yields `f : Forest (norm-e Rs) rhs s` and `r : A ⟶ rhs ∈ norm-e Rs` such that `t ≡ node r f`. For each tree `t'` of type `Tree (norm-e Rs) B s'` such that `t' ∈ 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 ⟶ rhs`. If `r' : A ⟶ rhs ∈ Rs`, then the proof is completed by the witness `node r' f'`. If `A ⟶ rhs ∉ Rs`, then by the definition of `norm-e` there exists some rule `r' : A ⟶ rhs' ∈ Rs` such that `rhs ∈ 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 `rhs'`.

Completeness Conversely, given a parse tree for some non-empty string (recall that `norm-e` makes all nonterminals non-nullable) in the original grammar, we can convert it into a parse tree in the normalized grammar:

```
ne-cmplt : (Rs : Rules) → (A : N) → (s : String)
  → Tree Rs A s → s ≢ [] → Tree (norm-e Rs) A s
```

Proof The proof is by induction on the height of the parse tree `t : Tree Rs A s`. By pattern matching, we have `t ≡ node r f` where `f : Forest Rs rhs s` and `r : A ⟶ rhs ∈ Rs`. For each tree `t' : Tree Rs B s'` such that `t' ∈ f`, let us analyze the possible cases:

- If `s' ≢ []`, then by the induction hypothesis, we can construct `Tree (norm-e Rs) A s'`.
- If `s' ≡ []`, then by `nlbls-cmplt`, we have that `B ∈ nullables Rs`.

Therefore, `f' : Forest (norm-e Rs) rhs' s` can be constructed where `rhs'` is a subsequence of `rhs` (the positions at which `f : Forest Rs rhs s` contains trees for the empty string are skipped). If `rhs' ≢ []`, then by completeness of `nullables` and `allSubSeq`, we get that `r' : A ⟶ rhs' ∈ norm-e Rs` and the proof is completed by the witness `node r' f'`. If `rhs' ≡ []`, then `s` should be empty (all positions are nulled), but this contradicts the assumption that `s ≢ []`.

3.5 Example

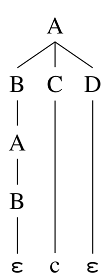
Consider the following grammar `Rs`:

```
A ⟶ BCD | B
B ⟶ ε | A
C ⟶ c
D ⟶ ε
```

Since `B` and `D` are the only nullable nonterminals, the grammar `norm-e Rs` has the following rules:

```
A ⟶ BCD | B | CD | BC | C
B ⟶ A
C ⟶ c
```

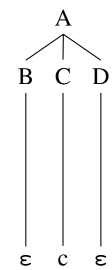
The nonterminal `D` in the grammar `norm-e Rs` is *nonproductive* (i.e., `(s : String) → Tree (norm-e Rs) D s → ⊥`). Let us look at the example of a tree `t` for the original grammar `Rs` and its counterpart for the ε-normalized grammar `norm-e Rs`:



`t : Tree Rs A "c"`

`ne-cmplt t : Tree (norm-e Rs) A "c"`

Note, that for the converse direction `ne-snd (ne-cmplt t) ≢ t` (there are many ways to construct subtrees for empty word):



`ne-snd (ne-cmplt t) : Tree Rs A "c"`

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:

```
nu-step : Rules → N → Rules
nu-step Rs A
 = { rule' | rule ∈ Rs, rule' ∈ step-f Rs A rule }
  where
    step-f : Rules → N → Rule → Rules
    step-f Rs A (B ⟶ rhs) =
       if rhs ≡ [ nt A ] then
        { B ⟶ rhs' | A ⟶ rhs' ∈ Rs,
                         rhs' ≢ [ nt A ] }
       else [ B ⟶ rhs ]
```

Compared to the grammar `Rs`, in the grammar `nu-step Rs A` every rule of the form `B ⟶ [ nt A ]` is replaced with all rules of the form `B ⟶ rhs'`, where `rhs'` stands for a right-hand side such that `A ⟶ rhs' ∈ Rs` and `rhs' ≢ [ nt A ]`. Now, full unit rule elimination is achieved by applying this procedure to all nonterminals:

```
norm-u : Rules → Rules
norm-u Rs = foldl nu-step Rs (NTs Rs)
```

Recall that `NTs Rs` is an enumeration of all nonterminals appearing in the grammar `Rs`.

4.2 Correctness

Progress First, we show that `nu-step` gains some progress:

```
nu-step-progress : (Rs : Rules) → (A B : N)
                 → A ⟶ [ nt B ] ∉ nu-step Rs B
```

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

```
nu-progress : (Rs : Rules) → (A B : N)
            → A ⟶ [ nt B ] ∉ norm-u Rs
```

Soundness We start by proving a lemma about possible shapes of rules in the original grammar:

```
nu-sound-main : (Rs : Rules) → (A B : N)
  → (rhs : RHS) → A ⟶ rhs ∈ nu-step Rs B
  → A ⟶ rhs ∈ Rs
       ∨ (A ⟶ [ nt B ] ∈ Rs × B ⟶ rhs ∈ Rs)
```

This lemma shows that, if a rule `A ⟶ rhs` belongs to a normalized grammar `nu-step Rs B`, then either the rule `A ⟶ rhs` belongs to `Rs` or the rules `A ⟶ [ nt B ]` and `B ⟶ rhs` do.

Now, we show how soundness follows from `nu-sound-main`:

```
nu-step-sound : (Rs : Rules) → (A B : N)
   → (s : String)
   → Tree (nu-step Rs B) A s → Tree Rs A s
```

Proof The proof is by induction on the height of the tree `t : 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 : A ⟶ rhs ∈ (nu-step Rs B)` such that `t ≡ node p f`. Next, for all trees `t' : Tree (nu-step Rs B) C s'` such that `t' ∈ f`, by the induction hypothesis, we turn `t'` into `t'' : Tree Rs C s'`. Therefore, by induction on the length of `f`, a forest `f' : Forest Rs rhs s` can be constructed. Finally, by `nu-sound-main` we have two cases:

- `p' : A ⟶ rhs ∈ Rs`. Then the proof is completed by constructing the witness `node p' f' : Tree Rs A s`.
- `p' : A ⟶ [ nt B ] ∈ Rs` and `p'' : B ⟶ rhs ∈ Rs`. Then the proof is completed by the giving the witness `node ((node p'' f') ::`$_n$` empty) p'` which has type `Tree Rs A s`.

Soundness of `norm-u` follows trivially from `nu-step-sound`:

```
nu-snd : (Rs : Rules) → (A : N) → (s : String)
          → Tree (norm-u Rs) A s → Tree Rs A s
```

Completeness We start again by observing special properties:

```
nu-cmplt' : (Rs : Rules) → (A B : N)
  → (rhs : RHS) → A ⟶ [ nt B ] ∈ Rs
  → B ⟶ rhs ∈ Rs → rhs ≢ [ nt B ]
  → A ⟶ rhs ∈ nu-step Rs B

nu-cmplt'' : (Rs : Rules) → (A B : N)
   → (rhs : RHS) → A ⟶ rhs ∈ Rs
   → rhs ≢ [ nt B ] → A ⟶ rhs ∈ nu-step Rs B
```

The `nu-cmplt'` lemma states that, if rules `A ⟶ [ nt B ]` and `B ⟶ rhs` belong to `Rs` and `rhs ≢ [ nt B ]`, then the rule `A ⟶ rhs` belongs to the normalized grammar `nu-step Rs B`. At the same time the lemma `nu-cmplt''` establishes that rules `A ⟶ rhs` where `rhs ≢ [ 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.

```
nu-step-complete : (Rs : Rules)
 → (A B : N) → (s : String)
 → Tree Rs A s → 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 : Forest Rs rhs s` and `p : A ⟶ rhs ∈ Rs` such that `t ≡ node p f`. Next, for all trees `t' : Tree Rs C s'` such that `t' ∈ f`, by the induction hypothesis, we construct `t'' : Tree (nu-step Rs B) C s'`. So, by induction on the length of `f`, we get `f' : Forest (nu-step Rs B) rhs s`. Finally, let us analyze two cases:

- If `rhs ≢ [ nt B ]`, then by `nu-cmplt''` we have `p' : A ⟶ rhs ∈ nu-step Rs B` and proof is finished by the witness `node p' f' : Tree (nu-step Rs B) A s`.
- If `rhs ≡ [ nt B ]`, then the previously constructed `f'` satisfies `f' ≡ (node q f'') ::`$_n$` empty` where `f''` is of type `Forest (nu-step Rs B) rhs' s` and `q` is of type `B ⟶ rhs' ∈ nu-step Rs B`. By `nu-step-progress` we know that `rhs' ≢ [ nt B ]`, therefore, by `nu-cmplt'` we get `p' : A ⟶ rhs' ∈ 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:

```
nu-cmplt : (Rs : Rules) → (A : N) → (s : String)
        → Tree Rs A s → Tree (norm-u Rs) A s
```

4.3 Example

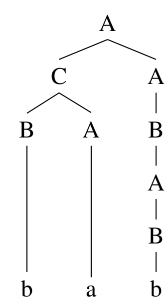
Consider the grammar

```
A ⟶ CA | B | a
B ⟶ b | A
C ⟶ BA
```

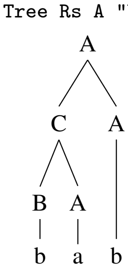
After the `norm-u` transformation we have:

```
A ⟶ CA | a | b
B ⟶ b | CA | a
C ⟶ BA
```

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

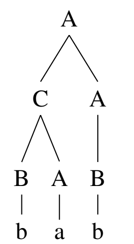


`t : Tree Rs A "bab"`



`nu-cmplt t : Tree (norm-u Rs) A "bab"`

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



```
nu-snd (nu-cmplt t) : Tree Rs A "bab"
```

Making the exact relationship between maps `nu-cmplt` and `nu-snd` precise is a possible future work.

5. Final transformations

5.1 Long right-hand sides

Next, we describe how to eliminate rules `A ⟶ rhs` where `length rhs` > 2—so-called *long rules*.

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

```
newnt : Rules → N
```

and a proof that `newnt Rs` does not occur anywhere in the grammar `Rs`:

```
newnt-lem : (Rs : Rules) → newnt Rs ∉ NTs Rs
```

The above states that `newnt Rs` is a "fresh" nonterminal. Note that there are no side effects involved here, the expression `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 `newnt` and `Rs` as follows:

```
newnt : Rules → N
newnt Rs = 1 + max (NTs Rs)

Rs : Rules
Rs = [ 1 ⟶ [ nt 2, tm 3, nt 4 ] ]

Rs' : Rules
Rs' = 1 ⟶ [ nt (newnt Rs) ] :: Rs
```

In that case, `newnt Rs ≡ 5`. Also, if we define `Rs'` by adding the rule `1 ⟶ [ nt (newnt Rs) ]` to `Rs`, then `newnt Rs' ≡ 6`.

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

```
nl-step' : Rules → N → Rules
nl-step' ((A ⟶ X :: Y :: Z :: rhs) :: Rs) F =
   (A ⟶ nt F :: Z :: rhs) ::
         (F ⟶ X :: Y :: []) :: Rs
nl-step' ((A ⟶ rhs) :: Rs) F =
   (A ⟶ rhs) :: nl-step' Rs F
nl-step' [] F = []

nl-step : Rules → Rules
nl-step Rs = nl-step' Rs (newnt Rs)
```

The function `nl-step` looks for the first long rule of the form `A ⟶ X :: Y :: Z :: rhs` and replaces it with rules `A ⟶ nt F :: Z :: rhs` and `F ⟶ X :: Y :: []` where `F` is fresh.

After applying the function `nl-step` to the grammar `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 `nl-step` needs to be applied to the grammar `Rs`.

```
nl-measure : Rules → ℕ
nl-measure Rs = sum lengths
  where
    lengths = { length rhs | A ⟶ rhs ∈ Rs,
                             length rhs > 2 }
```

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

```
norm-l : Rules → Rules
norm-l Rs = fold Rs nl-step (nl-measure Rs)
```

5.2 Right-hand sides containing terminals

In what follows, we describe how to eliminate rules `A ⟶ rhs` where `rhs` contains terminals and `length rhs` > 1.

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

```
nt-step : Rules → T → Rules
nt-step Rs a = let F = newnt Rs in
 F ⟶ tm a ::
 { A ⟶ subst (tm a) (nt F) rhs |
                               A ⟶ rhs ∈ Rs }
 where
  subst : Symbol → Symbol → RHS → RHS
  subst X Y rhs =
   if length rhs ≤ 1
   then rhs
   else map (λ Z → if Z ≡ X then Y else Z) rhs
```

Finally, remove all terminals from right-hand sides longer than 1 by folding `Rs` with the function `nt-step`:

```
norm-t : Rules → Rules
norm-t Rs = foldl nt-step Rs (Ts Rs)
```

5.3 Correctness of final transformations

Correctness of both `norm-t` and `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 `norm-l` states that after the transformation there are no rules with right-hand sides of more than two symbols.

```
nl-progress :  (Rs : Rules) → (A : N)
    → (rhs : RHS) →  A ⟶ rhs ∈ norm-l Rs
    → length rhs ≤ 2
```

The progress lemma for `norm-t` states that, for any `Rs` for all rules `A ⟶ rhs ∈ norm-t Rs`, either `rhs ≡ [ tm a ]` for some terminal `a` or `rhs` consists of nonterminals only.

```
nt-progress : (Rs : Rules) → (A : N)
    → (rhs : RHS) → A ⟶ rhs ∈ norm-t Rs
    → ∃(a : T) rhs ≡ [ tm a ] ∨ ntOnly rhs
```

Next, `nl-snd` and `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:

```
nl-snd : (Rs : Rules) → (A : N)
   → (s : String) → A ∈ NTs Rs
   → Tree (norm-l Rs) A s → Tree Rs A s

nt-snd : (Rs : Rules) → (A : N)
  → (s : String) → A ∈ NTs Rs
  → Tree (norm-t Rs) A s → Tree Rs A s
```

The side condition A ∈ 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.

```
nl-cmplt : (Rs : Rules) → (A : N) → (s : String)
    → Tree Rs A s → Tree (norm-l Rs) A s

nt-cmplt : (Rs : Rules) → (A : N) → (s : String)
    → Tree Rs A s → 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:

```
norm : Rules → Rules
norm = norm-u ∘ norm-e ∘ norm-t ∘ norm-l
```

The function norm is a composition of the four transformations we have introduced. The order in which 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 : ℕ, then the same holds for all rules in norm-t Rs:

```
nt-efct : (Rs : Rules) → (n : ℕ)
    → ((A : N) → (rhs : RHS)
                → A ⟶ rhs ∈ Rs
                → length rhs ≤ n)
    → (A : N) → (rhs : RHS)
    → A ⟶ rhs ∈ norm-t Rs
    → length rhs ≤ n
```

2. We show that, if A ⟶ rhs ∈ norm-e Rs, then rhs must be a subsequence of some rhs' such that A ⟶ rhs' ∈ 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:

```
ne-efct : (Rs : Rules) → (A : N) → (rhs : RHS)
  → A ⟶ rhs ∈ norm-e Rs → A ⟶ rhs ∉ Rs
  → ∃(rhs' : RHS) A ⟶ rhs' ∈ Rs ×
      rhs ∈ allSubSeq (∈ 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:

```
nu-efct : (P : RHS → Set) → (Rs : Rules)
  → ((A : N) → (rhs : RHS)
             → A ⟶ rhs ∈ Rs → P rhs)
  → (A : N) → (rhs : RHS)
  → A ⟶ rhs ∈ norm-u Rs
  → P rhs
```

Finally, we show the following progress property of norm:

```
norm-progress : (Rs : Rules) → (A : N)
    → (rhs : RHS) → A ⟶ rhs ∈ norm Rs
    → (∃(B C : N) rhs ≡ [ nt B, nt C ]) ∨
       (∃(a : T) rhs ≡ [ tm a ])
```

It states that, for any rule A ⟶ rhs ∈ norm Rs, either rhs ≡ [nt B, nt C] for some nonterminals B and C or rhs ≡ [tm a] for some terminal a.

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

```
norm-snd : (Rs : Rules) → (A : N)
  → (s : String)  → A ∈ NTs Rs
  → Tree (norm Rs) A s → Tree Rs A s
```

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

```
norm-cmplt : (Rs : Rules) → (A : N)
  → (s : String) → s ≢ []
  → Tree Rs A s  → Tree (norm Rs) A s
```

6.2 Grammars with a start nonterminal

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

```
record Grammar : Set where
  field
    S : N
    Rs : Rules
```

(Given some G : Grammar, we write S G and Rs G for projections of the start terminal and the list of rules respectively.)

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

```
TreeS : Grammar → String → Set
TreeS G s = Tree (Rs G) (S G) s
```

Next, we implement normalization of context-free grammars:

```
normS : Grammar → Grammar
normS G = record {
  S  = S';
  Rs = if S G ∈ nullables (Rs G)
       then S' ⟶ [] :: Rs'
       else Rs'
  }
   where
     S'  = newnt (Rs G)
     Rs' = norm ((S' ⟶ [ nt (S G) ]) :: Rs G)
```

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

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) ⟶ [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) ⟶ [] 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

```
normS-progress : (G : Grammar) → (A : N)
  → (rhs : RHS)
  → let G’ = normS G in A ⟶ rhs ∈ Rs G’
  → (∃(B C : N) rhs ≡ [ nt B, nt C ]
        × B ≢ S G’
        × C ≢ S G’) ∨
     (∃(a : T) rhs ≡ [ tm a ])  ∨
     (rhs ≡ [] × A ≡ S G’)
```

For any rule `A ⟶ rhs ∈ Rs (normS G)`, the right-hand side `rhs` is either `[ nt B, nt C ]` for some nonterminals `B` and `C` where neither `B` nor `C` are starting nonterminals or `[ tm a ]` for some terminal `a`, or `[]` with the condition that `A ≡ S (normS G)`.

Soundness and completeness

```
normS-snd : (G : Grammar) → (s : String)
   → S G ∈ NTs (Rs G)
   → TreeS (normS G) s → TreeS G s

normS-cmplt : (G : Grammar) → (s : String)
   → TreeS G s → 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 Binsztok [4] have formalized parsing expression grammars (PEGs), a formalism for specifying recursive descent parses, 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.

Danielsson [2] has implemented a library of parser combinators in Agda treating left recursion with coinduction. Sjöblom [11] has formalized an aspect of Valiant’s parsing algorithm.

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 [3], 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 nonterminal 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

[1] J. E. Hopcroft, J. D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[2] N. A. Danielsson. Total parser combinators. In *Proc. of 15th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP ’10*, pp. 285–296. ACM, 2010.

[3] D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. *J. of Log. and Algebr. Meth. in Program.*, v. 83(5–6), pp. 459–468, 2014.

[4] A. Koprowski, H. Binsztok. TRX: A formally verified parser interpreter. *Log. Meth. in Comput. Sci.*, v. 7(2), article 18, 2011.

[5] A. Barthwal, M. Norrish. A formalisation of the normal forms of context-free grammars in HOL4. In A. Dawar, H. Veith, eds., *Proc. of 24th Int. Wksh. on Computer Science Logic, CSL 2010*, v. 6247 of *Lect. Notes in Comput. Sci.*, pp. 95–109. Springer, 2010.

[6] A. Barthwal, M. Norrish. Verified, executable parsing. In G. Castagna, ed., *Proc. of 18th Europ. Symp. on Programming, ESOP 2009*, v. 5502 of *Lect. Notes in Comput. Sci.*, pp. 160–174. Springer, 2009.

[7] J.-H. Jourdan, F. Pottier, X. Leroy. Validating LR(1) parsers. In H. Seidl, ed., *Proc. of 21st Europ. Symp. on Programming, ESOP 2012*, v. 7211 of *Lect. Notes in Comput. Sci.*, pp. 397–416. Springer, 2012.

[8] Y. Minamide. Verified decision procedures on context-free grammars. In K. Schneider, J. Brandt, eds., *Proc. of 20th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLS 2007*, v. 4732 of *Lect. Notes in Comput. Sci.*, pp. 173–188. Springer, 2007.

[9] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, S. D. Swierstra, eds., *Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008*, v. 5832 of *Lect. Notes in Comput. Sci.*, pp. 230–266. Springer, 2009.

[10] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In J.-P. Jouannaud, Z. Shao, eds., *Proc. of 1st Int. Conf. on Certified Programs and Proofs, CPP 2011*, v. 7086 of *Lect. Notes in Comput. Sci.*, pp. 103–118. Springer, 2011.

[11] T. B. Sjöblom. An Agda proof of the correctness of Valiant’s algorithm for context free parsing. Master’s thesis, Dept. of Computer Sci. and Engin., Chalmers University of Technology, 2013.