A Formalisation of the Normal Forms of Context-Free Grammars in HOL4

Aditi Barthwal[1] and Michael Norrish[2]

[1] Australian National University
Aditi.Barthwal@anu.edu.au
[2] Canberra Research Lab., NICTA
Michael.Norrish@nicta.com.au

Abstract. We describe the formalisation of the Chomsky and Greibach normal forms for context-free grammars (CFGs) using the HOL4 theorem prover. We discuss the varying degrees to which proofs that are straightforward on pen and paper, turn out to be much harder to mechanise. While both proofs are of similar length in their informal presentations, the mechanised proofs for Greibach normal form blow-up considerably.

1 Introduction

A context-free grammar (CFG) provides a concise mechanism for describing the methods by which phrases in languages are built from smaller blocks, capturing the "block structure" of sentences in a natural way. The simplicity of this formalism makes it amenable to rigorous mathematical study.

CFGs form the basis of parsing. We have already mechanised some of the theory of CFGs [1]. Grammars can be *normalised*, resulting in rules that are constrained to be of a particular shape. These simpler, more regular, rules can help in subsequent proofs or algorithms. For example, using a grammar in Chomsky Normal Form (CNF), one can decide the membership of a string in polynomial time. Using a grammar in Greibach Normal Form (GNF), one can prove a parse tree for any string in the language will have depth equal to the length of the string.

The Chomsky and Greibach normal form results were first presented in [3] and [4] respectively. Here, as part of a wider program, we work from the presentation in Hopcroft and Ullman [5], a standard textbook.

By mechanising these results, we gain extra confidence in their correctness. Because the results are so basic, this may not seem much of an achievement, but the mechanised proofs do provide a foundation for the development of yet more mechanised theory. For example, the proof in Hopcroft and Ullman of the standard result equating grammars and push-down-automata (mechanised in [2]), assumes that the grammar is in GNF.

Moreover, simply assuming results such as these in order to pursue more complicated material defeats the motivation at the heart of mechanised mathematics.

Contributions

- The first mechanised proofs of termination and correctness for an algorithm that converts a CFG to Chomsky Normal Form (Section 3).
- The first mechanised proofs of termination and correctness for an algorithm that converts a CFG to Greibach Normal Form (Section 4).
- We also discuss the ways in which well-known, "classic" proofs can require considerable "reworking" when fully mechanised. Interestingly, though the proofs we mechanise here both expand dramatically from their length in Hopcroft and Ullman [5], the GNF proof expands a great deal more than the CNF proof.

All the assumptions and assertions in this paper have been mechanised and the HOL4 sources for the work are available at http://users.rsise.anu.edu.au/~aditi/.

2 Context-Free Grammars

A context-free grammar (CFG) is represented in HOL using the following type definitions:

```
('nts, 'ts) symbol = NTS of 'nts | TS of 'ts
('nts, 'ts) rule = rule of 'nts => ('nts, 'ts) symbol list
('nts, 'ts) grammar = G of ('nts, 'ts) rule list => 'nts
```

The `=>` arrow indicates curried arguments to an algebraic type's constructor. Thus, the `rule` constructor is a curried function taking a value of type `'nts` (the symbol at the head of the rule), a list of symbols (giving the rule's right-hand side), and returning an `('nts,'ts) rule`. The symbols are of two types, type `'nts` used for nonterminals and type `'ts` used for terminal symbols.

Thus, a rule pairs a value of type `'nts` with a symbol list. Similarly, a grammar consists of a list of rules and a value giving the start symbol. Traditional presentations of grammars often include separate sets corresponding to the grammar's terminals and nonterminals. Our grammar type does not include these sets explicitly as it is easy to derive these sets from the grammar's rules and start symbol, so we shall occasionally write a grammar G as a tuple (V, T, P, S) in the proofs to come. Here, V is the list of nonterminals or variables, T is the list of terminals, P is the list of productions and S is the start symbol.

Definition 1. *A list of symbols (or* sentential form*)* s derives t *in a single step if* s *is of the form* $\alpha A \gamma$, t *is of the form* $\alpha\beta\gamma$, *and if* $A \rightarrow \beta$ *is one of the rules in the grammar. In HOL:*

```
derives g lsl rsl ⟺
∃ s1 s2 rhs lhs.
  (s1 ++ [NTS lhs] ++ s2 = lsl) ∧ (s1 ++ rhs ++ s2 = rsl) ∧
  rule lhs rhs ∈ rules g
```

(The infix `++` *denotes list concatenation. The symbol* $\in$ *denotes membership.)*

We write `(derives g)`* sf_1 sf_2 to indicate that sf_2 is derived from sf_1 in zero or more steps, also written $sf_1 \Rightarrow^* sf_2$ (where the grammar g is assumed).

This is concretely represented using derivation lists. We write $R \vdash l \lhd x \rightarrow y$ to mean that the binary relation R holds between the successive pair of elements of l, which starts with x and end with y. Thus, $sf_1 \Rightarrow^* sf_2$ can be written as `derives` $g \vdash l \lhd sf_1 \rightarrow sf_2$ for some l. We also define the leftmost and the rightmost derivation relations, `lderives` ($\overset{l}{\Rightarrow}$) and `rderives` ($\overset{r}{\Rightarrow}$).

Definition 2. *The* language *of a grammar consists of all the words (lists of only terminal symbols) that can be derived from the start symbol.*

$$\texttt{L}\ g\ \texttt{=}\ \{\, tsl\ \texttt{|}\ \texttt{(derives}\ g\texttt{)}^*\ \texttt{[NTS (startSym}\ g\texttt{)]}\ tsl\ \wedge\ \texttt{isWord}\ tsl\,\}$$

(Predicate `isWord` is true of a sentential form if it consists of only terminal symbols.)

The choice of the derivation relation (`derives`, `lderives` or `rderives`) does not affect the language generated by a grammar. We say $x \overset{l}{\Rightarrow} y$ if y is obtained by expanding the leftmost non-terminal in x. Similarly, $x \overset{r}{\Rightarrow} y$ if y is obtained by the expansion of the rightmost non-terminal in x. We use this equivalence for the proof of normalisation to GNF. This equivalence forms a part of our background mechanisation.

3 Chomsky Normal Form

CFGs can be simplified by restricting the format of productions in the grammar without changing the language. Some such restrictions, which are shared by the normal forms we consider, are summarised below.

- Removing symbols that do not generate a terminal string or are not reachable from the start symbol of the grammar (useless symbols);
- Removing ϵ-productions (as long as ϵ is not in the language generated by the grammar);
- Removing unit productions, i.e. ones of the form $A \rightarrow B$ where B is a nonterminal symbol.

ϵ represents the empty word in the language of a grammar. An ϵ-production is one with an empty right-hand side.

The proofs that these restrictions can always be made without changing the language are available in our online resources.

In this section we concentrate on Chomsky Normal Form, assuming the grammar has already gone through the above simplifications.

Theorem 1 (Chomsky Normal Form). *Any context-free language without ϵ is generated by a grammar in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$. Here A, B, C are variables and a is a terminal.*

Proof. Let $g_1 = (V, T, P, S)$ be a context-free grammar. We can assume P contains no useless symbols, unit productions or ϵ-productions using the above simplifications. If a production has a single symbol on the right-hand side, that symbol must be a terminal. Thus, that production is already in an acceptable form. The remaining productions in g_1 are converted into CNF in two steps.

The first step is called `trans1Tmnl`, wherein a terminal occurring on the right side of a production gets replaced by a nonterminal in the following manner. We replace the productions of the form $l \rightarrow pts$ (p or s is nonempty and t is a terminal) with productions $A \rightarrow t$ and $l \rightarrow pAs$.

```
trans1Tmnl nt t g g' ⟺
  ∃ℓ r p s.
    rule ℓ r ∈ rules g ∧ r = p ++ [t] ++ s ∧
    (p ≠ [] ∨ s ≠ []) ∧ isTmnlSym t ∧
    NTS nt ∉ nonTerminals g ∧
    rules g' =
      delete (rule ℓ r) (rules g) ++
      [rule nt [t]; rule ℓ (p ++ [NTS nt] ++ s)] ∧
    startSym g' = startSym g
```

(Function `delete` removes an element from a list. The `;` is used to separate elements in a list.)

We prove that multiple applications of the above transformation preserve the language.

HOL Theorem 1

```
∀g g'. (λx y. ∃A t. trans1Tmnl A t x y)* g g' ⇒ (L g = L g')
```

We want to obtain a grammar $g_2 = (V', T, P', S)$ which only contains productions of the form $A \rightarrow a$, where a is a terminal symbol or $A \rightarrow A_1...A_n$ where A_i is a nonterminal symbol. We prove that such a g_2 can be obtained by repeated applications of `trans1Tmnl`. The multiple applications are denoted by taking the reflexive transitive closure of `trans1Tmnl`.

We define the constant `badTmnlsCount`, which counts the terminals occurring in the RHSs of all productions in the grammar which have more than one terminal symbol present in the RHS.

```
badTmnlsCount g = SUM (MAP ruleTmnls (rules g))
```

(`SUM` adds the count over all the productions, `MAP` f l applies f to each element in l.)

The auxiliary `ruleTmnls` is characterised:

```
ruleTmnls (rule ℓ r) = if |r| ≤ 1 then 0 else |FILTER isTmnlSym r|
```

($|r|$ denotes the length of a list r.)

Each application of the process should decrease the number of `ruleTmnls` unless there are none to change (*i.e.*, the grammar is already in the desired form). By induction on `badTmnlsCount` we prove that by a finite number of applications of

`trans1Tmnl` we can get grammar g_2. This follows from the fact that the set of symbols in a grammar is finite.

HOL Theorem 2

```
INFINITE (:α) ⇒
∃g'. (λx y. ∃nt t. trans1Tmnl nt t x y)* g g' ∧
  badTmnlsCount g' = 0
```

(Note the use of the assumption `INFINITE` $\mathcal{U}$. Here $\mathcal{U}$ represents the universal set for the type of nonterminals (α) in the grammar g and g'. The transformation process involves introducing a new nonterminal symbol. To be able to pick a fresh symbol, the set of nonterminals has to be infinite.)

The above process gives us a simplified grammar g_2 such that `badTmnlsCount` g_2 `= 0`. By HOL Theorem 1 we have that $L(g_1) = L(g_2)$. We now apply another transformation on g_2 which gives us our final CNF. The final transformation is called `trans2NT` and works by replacing two adjacent nonterminals in the right-hand side of a rule by a single nonterminal symbol. Repeated application on g_2 gives us a grammar where all productions conform to the CNF criteria.

```
trans2NT nt nt1 nt2 g g' ⟺
  ∃ℓ r p s.
    rule ℓ r ∈ rules g ∧ r = p ++ [nt1; nt2] ++ s ∧
    (p ≠ [] ∨ s ≠ []) ∧ isNonTmnlSym nt1 ∧ isNonTmnlSym nt2 ∧
    NTS nt ∉ nonTerminals g ∧
    rules g' =
      delete (rule ℓ r) (rules g) ++
      [rule nt [nt1; nt2]; rule ℓ (p ++ [NTS nt] ++ s)] ∧
    startSym g' = startSym g
```

We prove that the language remains the same after zero or more such transformations.

We follow a similar strategy as with `trans1Tmnl` to show that applications of `trans2NT` will result in grammar (g_3) where all rules with nonterminals on the RHS have exactly two nonterminals, *i.e.* rules are of the form $A \rightarrow A_1 A_2$.

To wrap up the proof we show two results. First, that applications of `trans1Tmnl` followed by applications of `trans2NT`, leaves the language of the grammar untouched, i.e. $L(g_1) = L(g_3)$. Second, that the transformation `trans2NT` does not introduce productions to change `badTmnlsCount`. We can then apply the two transformations to obtain our grammar in CNF where all rules are of the form $A \rightarrow a$ or $A \rightarrow A_1 A_2$ (asserted by the `isCnf` predicate). The HOL theorem corresponding to Theorem 1 is:

HOL Theorem 3

```
INFINITE (:α) ∧ [] ∉ language g ⇒
∃g'. isCnf g' ∧ language g = language g'
```

4 Greibach Normal Form

If ϵ does not belong in the language of a grammar then it can be transformed into Greibach Normal Form. The existence of GNF for a grammar simplifies many proofs,

such as the result that every context-free language can be accepted by a non-deterministic pushdown automata. Productions in GNF are of the form $A \rightarrow a\alpha$ where a is a terminal symbol and α is list (possibly empty) of nonterminals.

We assume an implicit ordering on the nonterminals of the grammar. The various stages in the conversion to GNF are summarised below.

Preprocessing Remove useless symbols, ϵ and unit productions from the grammar and convert it into Chomsky Normal Form.

Stage 1 For each ordered nonterminal A_k do the following:

- **Stage 1.1** Return rules of the form such that if $A_k \rightarrow A_j\alpha$ then $j \geq k$. This result is obtained using `aProds` lemma (Section 4.1).
- **Stage 1.2** Convert left recursive rules in the grammar to right recursive rules. This is based on `left2Right` lemma (Section 4.2).

Stage 2 Eliminate the leftmost nonterminal from the RHS of all the rules to obtain a grammar in GNF.

In the discussion to follow we assume the grammar already satisfies the Preprocessing requirements (following from the results already covered). We start at Stage 1 which is implemented using relation `r49` in Section 4.4. This is Phase 1 of the GNF algorithm. Stage 1 depends on two crucial results, Stage 1.1 and Stage 1.2 which are established separately. Stage 2 can be further subdivided into two parts covered in (Sections 4.5 and 4.6), the Phase 2 and Phase 3 of the algorithm, respectively.

All the stages preserve the language of the grammar. We devote much of our discussion to mechanising the more interesting stages for eliminating left recursion and putting together Stage 1 and Stage 2 to get the GNF algorithm.

4.1 Eliminating the leftmost nonterminal

Let A-productions be those productions whose LHS is the nonterminal A. We define the function `aProdsRules` A Bs $rset$ to transform the set of productions $rset$: the result is a set where occurrences of non-terminals in the list Bs no longer occur in leftmost position in A-productions. Instead, they have been replaced by their own right-hand-sides.

HOL Definition 1

```
aProdsRules A l ru =
ru DIFF {rule A ([NTS B] ++ s) | (B,s) |
          B ∈ l ∧ rule A ([NTS B] ++ s) ∈ ru} ∪
{rule A (x ++ s) | (x,s) |
 ∃B. B ∈ l ∧ rule A ([NTS B] ++ s) ∈ ru ∧ rule B x ∈ ru}
```

(The notation s_1 `DIFF` s_2 represents set-difference. Notation $|(p, B, s)|$ denotes that p, B and s are the bound variables.)

Lemma 1 (“`aProds` lemma”). *For all possible nonterminals A, and lists of nonterminals Bs, if* `rules` $g' =$ `aProdsRules` A Bs (`rules` g) *and the start symbols of g and g' are equal, then $L(g) = L(g')$.*

4.2 Replacing left recursion with right recursion

Left recursive rules may already be present in the grammar, or they may be introduced by the elimination of leftmost nonterminals (using the aProds lemma). In order to deal with such productions we transform them into right recursive rules. We show that this transformation preserves the language equivalence.

Theorem 2 (“`left2Right` lemma”). *Let $g = (V, T, P, S)$ be a CFG. Let $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_r$ be the set of left recursive A-productions. Let $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$ be the remaining A-productions. Then we can construct $g' = (V \cup \{B\}, T, P_1, S)$ such that $L(g) = L(g')$ by replacing all the left recursive A-productions by the following productions:*

Rule 1 $A \rightarrow \beta_i$ *and* $A \rightarrow \beta_i B$ *for* $1 \leq i \leq s$
Rule 2 $B \rightarrow \alpha_i$ *and* $B \rightarrow \alpha_i B$ *for* $1 \leq i \leq r$

Here, B is a fresh nonterminal that does not belong in g. This is our HOL Theorem 4.

Relation `left2Right` A B g g' holds iff the rules in g' are obtained by replacing all left recursive A-productions with rules of the form given by Rule 1 and Rule 2 (implemented by the `l2rRules` function).

HOL Definition 2

```
left2Right A B g g' ⟺
  NTS B ∉ nonTerminals g ∧ startSym g = startSym g' ∧
  set (rules g') = l2rRules A B (set (rules g))
```

In the textbook it is observed that a sequence of productions of the form $A \rightarrow A\alpha_i$ will eventually end with a production $A \rightarrow \beta_j$. The sequence of replacements

$$A \Rightarrow A\alpha_{i_1} \Rightarrow A\alpha_{i_2}\alpha_{i_1} \Rightarrow \ldots \Rightarrow A\alpha_{i_p} \ldots \alpha_{i_1} \Rightarrow \beta_j\alpha_{i_p} \ldots \alpha_{i_1} \tag{1}$$

in g can be replaced in g' by

$$A \Rightarrow \beta_j B \Rightarrow \beta_j\alpha_{i_p} B \Rightarrow \ldots \Rightarrow \beta_j\alpha_{i_p} \ldots \alpha_{i_2} B \Rightarrow \beta_j\alpha_{i_p} \ldots \alpha_{i_2}\alpha_{i_1} \tag{2}$$

Since it is clear that the reverse transformation is also possible, it is concluded that $L(g) = L(g')$. This is illustrated graphically in Figure 1.

HOL Theorem 4

```
∀g g'. left2Right A B g g' ⇒ (L g = L g')
```

This is a good example of a “proof” where the authors rely on “obvious” details to make their point: the proof in Hopcroft and Ullman consists of little more than equations (1) and (2), and a figure corresponding to our Figure 1. Unfortunately, a figure does not satisfy a theorem prover’s notion of a proof; moreover it fails to suggest any strategies that might be used for rigorous treatment (such as automation) of the material.

In the following section, we describe the proof strategy used to mechanise this result in HOL4. For the purposes of discussion, we will assume that we are removing left recursions in A-productions in g, using the new nonterminal B, producing the new grammar g'.

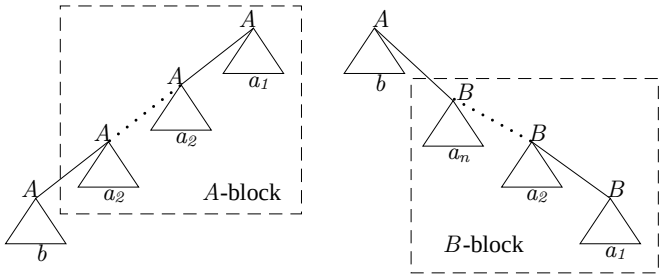

Fig. 1. A left recursive derivation $A \to Aa_1 \to Aa_2a_1 \to \cdots \to A_n \ldots a_2a_1 \to ba_n \ldots a_2a_1$ can be transformed into a right recursive derivation $A \to bB \to ba_n \to \cdots \to ba_n \ldots a_2 \to ba_n \ldots a_2a_1$. Here the RHS b does not start with an A.

Proof of the "if" direction We use a leftmost derivation with concrete derivation lists to show that if $x \overset{l}{\Rightarrow}{}^*_g\ y$, where y is a word, then $x\ \Rightarrow^*_{g'}\ y$.

HOL Theorem 5

`left2Right` $A\ B\ g\ g' \wedge$ `lderives` $g \vdash dl \lhd x \to y \wedge$ `isWord` $y \Rightarrow$
$\exists\, dl'$. `derives` $g' \vdash dl' \lhd x \to y$

The proof is by induction on the number of times A occurs as the leftmost symbol in the derivation dl. This is given by `ldNumNt` $A\ dl$.

Base Case If there are no As in the leftmost position (*i.e.* `ldNumNt` $NTS\ A\ dl\ =\ 0$) then the derivation in g can also be done in g'.

Step Case The step case revolves around the notion of a *block*. A block in a derivation is defined as a (nonempty) section of the derivation list where each expansion is done by using a left recursive rule of A. As such, the sentential forms in the block always have an A as their leftmost symbol. Figure 1 shows the A and B-blocks for leftmost and rightmost derivations.

If there is more than one instance of A in the leftmost position in a derivation, then we can divide it into three parts: dl_1 which does not have any leftmost As, dl_2 which is a block and dl_3 where the very first expansion is a result of one of the non-recursive rules of A. This is shown in Figure 2.

The division is given by HOL Theorem 6. The second $\exists$-clause of this theorem refers to the side conditions on the composition of the expansions shown in Figure 2.

In the absence of any leftmost As, a derivation is easily replicated in g'. Thus, the dl_1 portion can be done in g'. The derivation corresponding to dl_3 follows by our inductive hypothesis.

The proof falls through if derivation dl_2 can somehow be shown to have an equivalent in g'. This is shown by proving HOL Theorem 7. The theorem states that for a derivations in g of the form given by Equation (1), there is an equivalent derivation in g' in the form of Equation (2).

To show the remaining "only if" part, ($x \overset{r}{\Rightarrow}{}^*_{g'}\ y$, where y is a word, then $x \overset{r}{\Rightarrow}{}^*_g\ y$), we mirror the leftmost derivation strategy. In this case we rely on the rightmost derivation

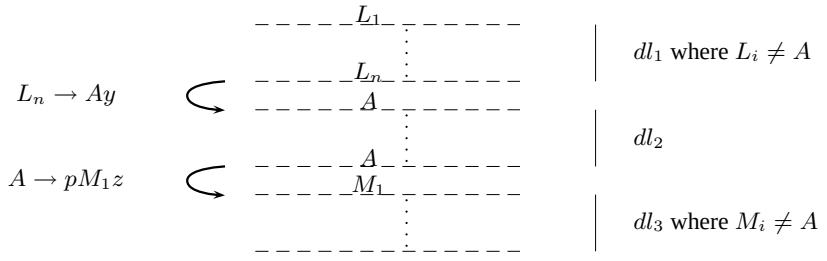

Fig. 2. We can split dl into dl_1, dl_2 and dl_3 such that dl_1 has no A-expansions, dl_2 consists of only A-expansions and dl_3 breaks the sequence of A-expansions so that the very first element in dl_3 is not a result of an A-expansion. The dashed lines are elements in the derivation list showing the leftmost nonterminals ($L_1 \ldots L_n, A, M_1$). $L_n \to Ay$ is the first A-expansion in dl and $A \to pM_1z$ (p is a word), breaks the sequence of the consecutive A-expansions in dl_2.

HOL Theorem 6

```
lderives g ⊢ dl ◁ x → y ∧ ldNumNt (NTS A) dl ≠ 0 ∧ |dl| > 1 ⇒
∃ dl1 dl2 dl3 .
  dl = dl1 ++ dl2 ++ dl3 ∧ ldNumNt (NTS A) dl1 = 0 ∧
  (∀ e1 e2 p s . dl2 = p ++ [e1; e2] ++ s ⇒ |e2| ≥ |e1|) ∧
  ∃ pfx .
    isWord pfx ∧
    (∀ e . e ∈ dl2 ⇒ ∃ sfx . e = pfx ++ [NTS A] ++ sfx) ∧
    dl2 ≠ [] ∧
    (dl3 ≠ [] ⇒
     |LAST dl2| ≤ |HD dl3| ⇒
     ¬(pfx ++ [NTS A]  HD dl3))
```

(Here x <<= y holds iff x is a prefix of y.)

and the notion of a B-block, wherein B is always the rightmost nonterminal. We omit the details due to the similarity with the proof of the if direction.

4.3 Stitching the pieces together

Using the `aProds` lemma and eliminating of left recursive rules, it is clear that any grammar can be transformed into Greibach Normal Form. The textbook achieves this by providing a concrete algorithm for transforming rules in the grammar into an intermediate form where left recursion has been eliminated. This is Phase 1 of our HOL implementation. We model this transformation with a relation. From this point, multiple applications of the `aProds` lemma transform the grammar into GNF. These applications correspond to the Phase 2 and Phase 3. Each phase brings the grammar a step closer to GNF.

Let $g = (V, T, P, S)$ and $V = A_1 \ldots A_n$ be the ordered nonterminals in g. We will need at least n fresh Bs that are not in g when transforming the left recursive rules into right recursive rules. Let $B = B_1, \ldots, B_n$ be these distinct nonterminals. The three phases are applied in succession to the grammar and achieve the following results.

HOL Theorem 7

```
left2Right A B g g' ∧
lderives g ⊢ dl ◁ pfx ++ [NTS A] ++ sfx → y ∧
lderives g y y' ∧ isWord pfx ∧
(∀ e. e ∈ dl ⇒ ∃ sfx. e = pfx ++ [NTS A] ++ sfx) ∧
(∀ e1 e2 p s. dl = p ++ [e1; e2] ++ s ⇒ |e2| ≥ |e1|) ∧
(|y| ≤ |y'| ⇒ ¬(pfx ++ [NTS A]  y')) ⇒
∃ dl'. derives g' ⊢ dl' ◁ pfx ++ [NTS A] ++ sfx → y'
```

Phase 1 Transform the rules in g to give a new grammar $g_1 = (V_1, T, P_1, S)$ such that if $A_i \rightarrow A_j\alpha$ is a rule of g_1, then $j > i$. Since i not equal to j, we have removed the left recursive rules in g, introducing m new B non-terminals, where $m \leq n$. Thus, $V_1 \subseteq V \cup \{B_1, \ldots, B_n\}$. This is done using multiple applications of the `aProds` transformation followed by a single application of `left2Right`. This process is applied progressively to each of the nonterminals.

Phase 2 All the rules of the form $A_i \rightarrow A_j\beta$ in g_1 are replaced by $A_i \rightarrow a\alpha\beta$, where $A_j \rightarrow a\alpha$ to give a new grammar $g_2 = (V_1, T, P_2, S)$. This is done progressively for each of the nonterminals in V by using the `aProds` lemma.

Phase 3 All the rules of the form $B_k \rightarrow A_i\beta$ in g_2 are replaced with $B_k \rightarrow a\alpha\beta$, where $A_i \rightarrow a\alpha$ to give $g_3 = (V_1, T, P_3, S)$ such that g_3 is in Greibach Normal Form. Again, applying the `aProds` lemma progressively for each of the Bs gives us a grammar in GNF.

4.4 Phase 1—Ordering the A_i-productions

Phase 1 is represented by the HOL relation `r49`. This corresponds to Figure 4.9 in Hopcroft and Ullman showing the first step in the GNF algorithm. The `r49` relation relates two states where the second state is the result of transforming rules for a single A_k.

HOL Definition 3

```
r49 (bs0, nts0, g0, seen0, ubs0) (bs, nts, g, seen, ubs) ⟺
∃ Ak b rules0 rules1 .
  (nts0 = Ak::nts) ∧ (bs0 = b::bs) ∧ (ubs = ubs0 ++ [b]) ∧
  (seen = seen0 ++ [Ak]) ∧ (nts = TL nts0) ∧
  (r49Elem Ak)* (seen0,rules g0,[]) ([],rules0,seen0) ∧
  (rules1 = l2rRules Ak b (set rules0)) ∧
  (startSym g = startSym g0) ∧ (set (rules g) = rules1)
```

(Here bs_0 consists of fresh B_is not in grammar g_0, nts_0 are the ordered nonterminals (increasing) in g_0, $seen_0$ holds the nonterminals and ubs_0 holds the B_is that have been already processed. The relation, `r49`, holds if the rules of g are obtained by transforming rules for a single nonterminal (A_k) and using up a fresh non-terminal b to eliminate (possible) left recursion for A_k. The b is used up irrespective of whether a left recursion

elimination is required or not. This simplifies both the definition and reasoning for the relation.)

There are two parts to `r49`. The first part is the relation `r49Elem`. This works on a single nonterminal and progressively eliminates rules of the form $A_k \rightarrow A_j\gamma$ where j has a lower ranking than k and is in $seen_0$. We do this for each element of $seen_0$ starting from the lowest ranked. $seen_0$ consists of ordered nonterminals having a lower ranking than k.

HOL Definition 4

```
r49Elem A_k (seen_0, ru_0, sl_0) (seen, ru, sl) ⟺
∃ A_j. (seen_0 = A_j::seen) ∧ (sl = sl_0 ++ [A_j]) ∧
     (set ru = aProdsRules A_k [A_j] (set ru_0))
```

Using the closure, `r49Elem`*, we can repeatedly do this transformation for all the elements in $seen_0$ to obtain the new set of rules $rules_0$ in the `r49` definition. At the end of the above transformation, we have productions of the form $A_k \rightarrow A_j\gamma$, where $j \geq i$. In the second part (corresponding to `l2rRules`), we replace productions of the form $A_k \rightarrow A_k\alpha$ with their right recursive counterparts to obtain a new set of rules using the `l2rRules` function. The above process is repeated for each nonterminal in g by taking the closure of `r49`. Thus, if `r49`* $(B, V, g, [\,], [\,])$ $(B_1, [\,], g_1, V, B_2)$ holds then g_1 has successfully been transformed to satisfy the Phase 1 conditions. More explicitly, as mentioned in Hopcroft and Ullman, the rules in g_1 should now be of the form:

- **(C1) Ordered A_i rules** - if $A_i \rightarrow A_j\gamma$ is in rules of g, then $j > i$.
- **(C2) A_i rules in GNF** - $A_i \rightarrow a\gamma$, where a is T.
- **(C3) B_i rules** - $B_i \rightarrow \gamma$, where y is in $(V \cup B_1, B_2, \ldots, B_n)^*$.

Automating an algorithm That Phase 1 has achieved these succinctly stated conditions is "obvious" to the human reader because of the ordering imposed on the nonterminals. A theorem prover, unfortunately, cannot make such deductive leaps. In an automated environment, the only assertions are the ones already present as part of the system or what one brings, *i.e.* verifies, as part of mechanising a theory. In particular, we need to define and prove invariant a number of conditions on the state of the system as it is transformed.

The composition of the rules from conditions (**C1**) and (**C2**) is asserted using the invariant `rhsTlNonTms`:

HOL Definition 5

```
rhsTlNonTms ru ntsl bs ⟺
∀ e. e ∈ set ntsl DIFF set bs ⇒
  ∀ r. rule e r ∈ ru ⇒
    ∃ h t. (r = h::t) ∧ EVERY isNonTmnlSym t ∧
      ∀ nt. (h = NTS nt) ⇒
        nt ∈ set ntsl DIFF set bs ∧
        ∃ nt_1 t_1. (t = NTS nt_1::t_1) ∧ nt_1 ∈ set ntsl DIFF set bs
```

Invariant `seenInv` asserts the ordering ($j > i$) part of (`C1`):

HOL Definition 6

```
seenInv ru s ⟺
∀i. i < |s| ⇒
  ∀nt rest. rule (EL i s) (NTS nt::rest) ∈ ru ⇒
    ∀j. j ≤ i ⇒ EL j s ≠ nt
```

(The notation `EL i ℓ` *denotes the* i^{th} *element of* ℓ*.)*

The invariant `rhsBNonTms` ensures (**C3**). This is stronger than what we need (at least at this stage of the process), since it also states that the very first nonterminal in the RHS has to be one of the A_is. This is observed in the textbook as part of later transformations (our own Phase 3), but actually needs to be proved at this stage.

HOL Definition 7

```
rhsBNonTms ru ubs ⟺
∀B. B ∈ ubs ⇒ ∀r. rule B r ∈ ru ⇒
                  EVERY isNonTmnlSym r ∧ r ≠ [] ∧
                  ∃nt. (HD r = NTS nt) ∧ ¬(nt ∈ ubs)
```

(Function `HD` *returns the first element of a list.)*

Most of the reasoning in the textbook translates to providing such specific invariants. These assertions, easily and convincingly made in text, have hidden assumptions that need to be identified and proved before proceeding with the automation.

A straightforward example is one concerning the absence of ϵ-productions. From the construction, it is clear that there are no ϵ-rules in the grammar (because it is in CNF), and that the construction does not introduce any. One does not realise the need for such a trivial property to be established until its absence stops the proof midway during automation. There are ten invariants that had to be established as part of the proof. This has to be done both for the single step case and for the closure of the relation.

Proof of language equivalence With all the required properties established, we can now go on to prove:

HOL Theorem 8

```
r49* (bs0,nts0,g0,seen0,ubs0) (bs,nts,g,seen,ubs) ∧
  |bs0| ≥ |nts0| ∧ ALL_DISTINCT bs0 ∧ ALL_DISTINCT nts0 ∧
  (set (ntms g0) ∩ set bs0 = ∅) ∧ (set bs0 ∩ set ubs0 = ∅) ∧
  (set nts0 ∩ set seen0 = ∅) ⇒
  (L g0 = L g)
```

(The distinct nonterminals in a grammar g *are given by* `ntms g`*.)*

In order to reason about which nonterminals have already been handled, we maintain the seen nonterminals and the seen B_is as part of our states. Because of this, extra assertions about them have to provided. Once 'seen', a nonterminal or a B_i cannot be seen again (citing the uniqueness of the B_is and the nonterminals in g). These are reflected in the various conditions of the form $s_1 \cap s_2 = \emptyset$.

Proof. The proof is by induction on number of applications of `r49`.

The above proof becomes trivially true if relation `r49` fails to hold. To counter this concern, we show that such a transformation does exist for any start state. This step is necessary because we have modelled transformations using relations (which can be partial) rather than by functions (which must be total in HOL).

HOL Theorem 9

$|bs_0| \geq |nts_0| \wedge$ ALL_DISTINCT $bs_0 \wedge$ ALL_DISTINCT $nts_0 \wedge$
(set $nts_0 \cap$ set $seen_0$ = $\emptyset$) $\wedge$ (set (ntms g_0) $\cap$ set bs_0 = $\emptyset$) $\wedge$
(set $bs_0 \cap$ set ubs_0 = $\emptyset$) $\Rightarrow$
$\exists g$. r49* (bs_0, nts_0, g_0, $seen_0$, ubs_0)
(DROP $|nts_0|$ bs_0, [], g, $seen_0$ ++ nts_0, ubs_0 ++ TAKE $|nts_0|$ bs_0)

(Function DROP n ℓ *drops* n *elements from the front of* l *and* TAKE n ℓ *takes* n *elements from the front of* l*.)*

Proof. The proof is by induction on nts_0.

Note: Everywhere a relation such as `r49`, `r49Elem` is used we have provided proofs that the relation always holds.

4.5 Phase 2—Changing A_i-productions to GNF

We have already established that removing useless nonterminals does not affect the language of the grammar. The nonterminals in g_1 are ordered such that the RHS of a nonterminal cannot start with a nonterminal with lower index. Since g is in CNF and only has useful nonterminals, rule $A_n \rightarrow a$ is in g_1, where a is in T. A_n is the highest ranked nonterminal and as such cannot expand to any nonterminal.

Thus, if we transform nonterminals V using `aProds`, starting from the highest rank, we are bound to get rules of the form, $A_k \rightarrow a\alpha$, for a in T and α in V_1. This is done by repeated applications of `fstNtm2Tm` until all the nonterminals in V have been transformed.

HOL Definition 8

fstNtm2Tm ($ontms_0$, g_0, $seen_0$) ($ontms$, g, $seen$) $\Longleftrightarrow$
$\exists A_k$ $rules_0$. ($ontms_0$ = $ontms$ ++ [A_k]) $\wedge$ ($seen$ = A_k::$seen_0$) $\wedge$
(r49Elem A_k)* ($seen_0$,rules g_0,[]) ([], $rules_0$, $seen_0$)
$\wedge$ (rules g = $rules_0$) $\wedge$ (startSym g = startSym g_0)

(Here $ontms_0$ *are nonterminals with indices in decreasing order* $(A_n, A_{n-1} \ldots A_1)$ *and* $seen_0$ *contains the nonterminals that have already been processed.)*

To prove that all the 'seen' nonterminals in V are in GNF, we establish that `gnfInv` invariant holds through the multiple applications of the relation.

HOL Definition 9

gnfInv ru s $\Longleftrightarrow$
$\forall i$. $i < |s| \Rightarrow$
$\forall r$. rule (EL i s) $r \in ru \Rightarrow$ validGnfProd (rule (EL i s) r)

(Predicate validGnfProd (rule ℓ r) *holds iff* $r = a\alpha$ *for terminal* a *and (possibly empty) list of nonterminals* α*.)*

Multiple applications of this process result in g_2 satisfying the `Phase 2` condition that all the rules for nonterminals in V are now in GNF.

4.6 Phase 3—Changing B_i-productions to GNF

The final phase is concerned with the rules corresponding to the B_is which are introduced as part of the left to right transformation. We follow a similar strategy to Phase 2 to convert all B_i-productions to GNF.

At the end of Phase 2, all rules involving nonterminals in V are of the form $A_i \rightarrow a\alpha$, for terminal a and list (possibly empty) of nonterminals α. From the invariant `rhsBNonTms`, we have that the B_i rules are of the form $B_i \rightarrow A_k\beta$ where β is a list (possibly empty) of nonterminals. The `aProds` lemma is now used to obtain rules of the form $B_i \rightarrow a\alpha\beta$ which satisfy GNF, done by establishing that `gnfInv` holds for seen B_is.

HOL Definition 10

`fstNtm2TmBrules (`ubs_0`,`$ontms_0$`,`g_0`,`$seen_0$`) (`ubs`,`$ontms$`,`g`,`$seen$`)` $\iff$
$\exists\, b\ rules_0$ `.`
`(`ubs_0 `=` b`::`ubs`)` $\wedge$ `(`$ontms_0$ `=` $ontms$`)` $\wedge$ `(`$seen$ `=` $seen_0$ `++ [`b`])` $\wedge$
`(`$rules_0$ `= aProdsRules` b $ontms$ `(set (rules` g_0 `)))` $\wedge$
`(set (rules` g`) =` $rules_0$`)` $\wedge$ `(startSym` g `= startSym` g_0`)`

($ubs0$ *contains* $B_1 \ldots B_n$*,* $ontms_0 = V$*, the nonterminals in the original grammar* g *and* $seen_0$ *is used to keep a record of the* B_i*s that have been handled.)*

We show that the above transformation resulting in grammar g_3 preserves the language of the grammar.

Finally, the three phases can be combined to show that any grammar that can be transformed into Chomsky Normal Form can be subsequently transformed into a grammar in Greibach Normal Form.

This transformation resulting in grammar g_3 also preserves the language of the grammar and the invariant `gnfInv` over the seen B_is.

Finally, the three phases can be combined to show that any grammar that can be transformed into Chomsky Normal Form can be subsequently transformed into a grammar in Greibach Normal Form.

HOL Theorem 10

`INFINITE` $\mathcal{U}$`(:`α`)` $\wedge$ `[]` $\notin$ `language` g $\wedge$ `language` $g \neq \emptyset \Rightarrow$
$\exists\, g'$`. isGnf` g' $\wedge$ `language` g `= language` g'

(The predicate `isGnf` g *holds iff predicate* `gnfInv` *is true for the rules and nonterminals of* g*.)*

5 Related work and conclusions

In the field of language theory, Nipkow [6] has provided a verified and executable lexical analyzer generator. This is the closest in nature to the mechanisation we have done.

The proof for CNF is ~1400 lines and GNF is ~6000, which is excluding proofs that are in the HOL library as well as the library maintained by us. Only parts of the proofs have been shown. The proof for CNF only covers half a page in the textbook. On the other hand, GNF covers almost three pages (including the two lemmas). This includes diagrams to assist explanation and an informal, high level reasoning. All of this is beyond the reach of automation in its current state. Issues such as finiteness and termination, which do not arise in a textual proof, become central when mechanising it. Similarly, choice of data structures and the form of definitions (relations vs functions) have a huge impact on the size of the proof as well as the ease of automation. These do not necessarily overlap. We have only presented the key theorems that are relevant to understanding and filling some of the deductive gaps in the textbook proofs. These theorems also cover the intermediate results needed because of the particular mechanisation technique. The size of these gaps also depends on the extent of detail in the text proof, which in our case is very sparse. It is hard to frame general techniques when the majority of the results require carefully combing the fine details in the text and making deductions about the omitted steps in the reasoning. From deducing and implementing the structure for induction for the `left2Right` lemma to establishing the numerous invariants for the final step of GNF algorithm, the problems for automation are quite diverse. Having extensive libraries is possibly the best way to tackle such highly domain specific problems. Like typical software development, the dream of libraries comprehensive enough to support all needed development fails as soon as one steps outside of the already conquered areas. The need for ever more libraries has never really gone away.

The simplification of CFGs (including CNF and GNF) is ~14000 lines. It took a year to complete the work which includes over 700 lemmas/theorems.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Aditi Barthwal and Michael Norrish. Verified, executable parsing. In Giuseppe Castagna, editor, *Programming Languages and Systems: 18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, March 2009.
2. Aditi Barthwal and Michael Norrish. Mechanisation of pda and grammar equivalence for context-free languages. In *To appear in Proceedings of WoLLIC*. Springer, 2010.
3. N Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.
4. Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.
5. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Ma., USA, 1979.
6. Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, pages 1–15, Canberra, Australia, 1998. Springer-Verlag LNCS 1479.