A Mechanisation of Some Context-Free Language Theory in HOL4

Aditi Barthwal[a,*], Michael Norrish[b,**]

[a]*Australian National University*
[b]*Canberra Research Lab., NICTA*

Abstract

We describe the mechanisation of some foundational results in the theory of context-free languages (CFLs), using the HOL4 system. We focus on push-down automata (PDAs). We show that two standard acceptance criteria for PDAs ("accept-by-empty-stack" and "accept-by-final-state") are equivalent in power. We are then able to show that the pushdown automata (PDAs) and context-free grammars (CFGs) accept the same languages by showing that each can emulate the other. With both of these models to hand, we can then show a number of basic, but important results. For example, we prove the basic closure properties of the context-free languages such as union and concatenation. Along the way, we also discuss the varying extent to which textbook proofs (we follow Hopcroft and Ullman) and our mechanisations diverge: sometimes elegant textbook proofs remain elegant in HOL; sometimes the required mechanisation effort blows up unconscionably.

Keywords: context-free languages, context-free grammars, pushdown automata, closure properties, HOL4

1. Introduction

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

*Principal corresponding author
**Corresponding author
Email addresses: `Aditi.Barthwal@anu.edu.au` (Aditi Barthwal), `Michael.Norrish@nicta.com.au` (Michael Norrish)

study. Context-free grammars are also simple enough to allow the construction of efficient parsing algorithms using pushdown automata (PDAs). These "predicting machines" use knowledge about their stack contents to determine whether and how a given string can be generated by the grammar. For example, PDAs underlie the construction of efficient parsers for LR grammars.

The theory of context-free languages is well understood and elegant. It is also of clear practical importance, being the basis for the description and implementation of computer programming language syntax. For these reasons, we feel it is a natural object for mechanised study. Our contribution in this paper is to describe the mechanisation of a number of basic results in this area. The most significant of these is the equivalence of the CFG and PDA models for context-free languages. Needless to say, there is no question that this theory is in any doubt! Nonetheless, there *are* a number of other reasons for performing these mechanisations:

- We investigate the degree to which the mechanical system (HOL4 Gordon and Melham (1993); Slind and Norrish (2008) in our case) is capable of dealing with such important mathematics.

- We also provide an important basis for future, more complicated mechanised developments. Without basic results such as those proved here, mechanised developments such as our verification of generation of SLR automata (Barthwal and Norrish, 2009) would not be possible. The library of proofs, techniques and notations developed here provides the basis from which further work on verified language theory can proceed at a quickened pace.

- The mechanisation work is an engaging intellectual exercise in itself; performing proofs of this sort is a very good way to iron out any wrinkles in one's understanding of the material. As such, we highly recommend mechanisation to all!

The rest of the paper is structured as follows. We introduce the basic details of our types (languages, grammars and automata) in Sections 2 and 3. In Section 4 we describe the two ways in which a PDA can accept an input ("acceptance by empty stack" and "acceptance by final state"), and show that they are equivalent in power. In Section 5, we present some closure properties for context-free languages.

It turns out that closure under union, concatenation and Kleene closure (Sections 5.1 to 5.3) are easy to formalise. These results depend on the fact that given any two grammars, G_1 and G_2, one can rename the variables such that variables of G_1 and G_2 are disjoint. The main effort goes into establishing this 'disjoint' property. The remaining closure property illustrates how equivalence results for different models can be a great help: closure under substitution (Section 5.4) uses a 'parse tree' representation (Section 5.4.1) for derivations (rather than derivation lists).

Finally, in Sections 6 and 7 we describe the paper's most involved mechanisations: proofs of the result that the CFG and PDA formalisms are equivalent in power.

HOL4: Background, Theorems and Notation. The work described in this paper was all carried out in HOL4 (Gordon and Melham, 1993; Slind and Norrish, 2008), a modern interactive theorem-proving system, or proof assistant. Based on the "LCF philosophy", HOL4 has a small trusted code base, its proof kernel. All theorems proved in the system ultimately depend only on that kernel, providing a high degree of confidence in their validity.

In this paper, all statements identified as **HOL Theorems** are theorems mechanically proved in the system and automatically pretty-printed to LaTeX from the relevant theory in the HOL4 development. Any discussion of proofs accompanying these theorems will be in an informal style. The actual proof scripts consumed by the system are typically quite incomprehensible, for all that they do embody the mathematical essence of the proof. For this reason, the paper does not include any HOL4 source code directly. Of course, full sources are available for download (see the following "availability" section).

Notation specific to this paper is explained as it is introduced. Otherwise, HOL4 supports a notation that is a generally pleasant combination of predicate logic (quantifiers $\forall$, $\exists$, connectives $\wedge$, $\Rightarrow$, for example) and functional programming (λ for function abstraction, juxtaposition for function application).

Lists are written between square brackets, *e.g.,* `[1; 2]`. The length of a list ℓ is written $|\ell|$. The concatenation of ℓ_1 and ℓ_2 is written ℓ_1 ++ ℓ_2. Overloading notation, we use $\in$ (and $\notin$) to refer to membership (non-membership) of both lists and sets. Lists support a range of functions from the functional programming world such as `MAP` and `FLAT` (which takes a list of lists and returns the concatenation of all the member lists).

Concrete sets are written between braces, *e.g.,* `{1; 2}`. We can write set comprehensions in typical syntax: the expression $\{x \mid x < 4\}$ denotes the set of numbers less than 4. If R is a (curried) binary relation (such that we write $R\ x\ y$ when x and y are linked in the relation), then R^* is the reflexive and transitive closure of R.

Availability. Both HOL4 itself and the sources for this work are available from `github.com`. The paper's sources are at `http://github.com/mn200/CFL-HOL`, and build with the latest repository version of HOL4. They will also build with the next official release of HOL4, Kananaskis-8, due in August 2012.

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.

2. Context-Free Grammars

Symbols. Grammars use *symbols* of two types, terminals and non-terminals. We use HOL's parametric polymorphism to allow both sorts of symbol to draw from arbitrary types, which then become type-arguments to the binary type-operator `symbol`. The definition is

```
('nts, 'ts) symbol = NTS of 'nts | TS of 'ts
```

This means that the type of symbols has two constructors, `NTS` and `TS`, and that, for example, `TS` has type

```
'ts -> ('nts, 'ts) symbol
```

For values t from the type `'ts`, the term `TS` t is a (terminal) symbol.

Rules and Grammars. A *rule* is a pair of a non-terminal symbol and a possible expansion for that non-terminal. We then write $N \rightsquigarrow \mathit{rhs}$ to indicate that pairing of N and rhs, with N of type `'nts` and rhs a list of symbols, thus of type `('nts, 'ts) symbol list`. A *grammar* is then a pair of a start symbol with a list of rules. Traditional presentations of grammars often include separate sets corresponding to the grammar's terminals and non-terminals. It's 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 non-terminals, T is the list of terminals, P is the list of productions and S is the start symbol.

Definition 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 $A \rightsquigarrow \beta$ is one of the rules in the grammar. In HOL, relation $\mathit{sf}_1 \;\Rightarrow_g\; \mathit{sf}_2$ holds iff sentential form sf_1 can derive sentential form sf_2 with respect to grammar g.

HOL Definition 1.

$$
\begin{array}{l}
sf_1 \ \Rightarrow_g \ sf_2 \iff \\
\quad \exists \alpha \ \gamma \ \beta \ A. \\
\qquad \alpha \ \texttt{++} \ [\texttt{NTS} \ A] \ \texttt{++} \ \gamma = sf_1 \wedge \alpha \ \texttt{++} \ \beta \ \texttt{++} \ \gamma = sf_2 \wedge \\
\qquad A \leadsto \beta \in \texttt{rules} \ g
\end{array}
$$

We also write $sf_1 \ \Rightarrow_g^* \ sf_2$ to indicate that sf_2 is derived from sf_1 in zero or more steps.

We can also represent derivations more concretely using *derivation lists*. If an arbitrary binary relation R holds on adjacent elements of list ℓ which has x as its first element and y as its last element, then we write $R \vdash \ell \lhd x \to y$. For example a derivation sequence $\ell_1 \Rightarrow \ell_2 \Rightarrow \ell_3 \ldots \Rightarrow \ell_n$ can be represented using lists as $\texttt{derives} \ g \vdash \ell \lhd \ell_1 \to \ell_n$ where $\ell = \ell_1 \ell_2 \ldots \ell_n$, and $\texttt{derives} \ g$ is the binary relation underneath $sf_1 \Rightarrow_g sf_2$.

In the context of grammars, R relates sentential forms. Later we will use the same notation to relate derivations in a PDA. Using this very concrete formulation simplifies the mechanisation of the proofs of a number of theorems.

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

HOL Definition 2.

$$L \ g = \{ w \mid [\texttt{NTS} \ (\texttt{startSym} \ g)] \Rightarrow_g^* w \wedge \texttt{isWord} \ w \}$$

Function `isWord` w *returns true if all the elements in the sentential form* w *are terminal symbols.*

3. Pushdown Automata

We model PDAs as records containing four components: the start state (`start` or q_0); the starting stack symbol (`ssSym` or Z_0); the list of final states (`final` or F); and the next state transitions (`final` or δ).

```
pda =
  <| start  : 'state;
     ssSym  : 'ssym;
     next   : ('isym, 'ssym, 'state) trans list;
     final  : 'state list |>
```

The input alphabet (Σ), stack alphabet (Γ) and the states for the PDA (Q) can be easily extracted from the above information. In prose proofs, we will occasionally refer to a PDA M as the tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. We have used lists instead of sets to avoid unnecessary finiteness constraints in our proofs.

The `trans` type describes a single transition, or link in the state machine's control graph. Such a transition is a tuple of an optional input symbol, a stack symbol and a state, and the next state along with the stack symbols (possibly none) to be added onto the current stack. The *next* field of the PDA record is a list of such transitions.

```
trans = ('isym option # 'ssym # 'state) # ('state # 'ssym list)
```

In HOL, a PDA transition in machine M is expressed using a binary relation on "instantaneous descriptions" of the tape, the machine's stack, and its internal state. We write

$$(q, i::\alpha, s) \vdash_M (q', i', s')$$

to mean that in state q, looking at input i with stack s, M can transition to state q', with the input becoming i' and the stack becoming s'. The input i' is either the same as $i::\alpha$ (referred to as an ϵ move) or is equal to α. Here, consuming the input symbol i corresponds to `SOME` i and ignoring the input symbol is `NONE` in the `trans` type.

Using the concrete derivation list notation, we write `ID` $M \vdash \ell \lhd x \to y$ to mean that the list ℓ is a sequence of valid instantaneous descriptions for machine M, starting with description x and ending with y. Transitions are not possible in states where the stack is empty and only ϵ moves are possible in states where the input is empty.

There are two ways in which a PDA can accept its input. The first way in which a PDA recognises an input is "acceptance by final state". This gives us the *l*anguage *a*ccepted by *f*inal *s*tate (`lafs`). In this scenario, the automata reaches an accepting state after it is has finished reading the input, and the stack contents are irrelevant.

HOL Definition 3 (lafs).

```
lafs $M$ =
  {$w$ |
   $\exists$ $state$  $stack$.
     ($M$.start, $w$, [$M$.ssSym]) $\vdash^*_M$ ($state$, [], $stack$) $\wedge$
     $state$ $\in$ $M$.final}
```

The second is "acceptance by empty stack". This gives us the *l*anguage *a*ccepted by *e*mpty *s*tack (`laes`). In this case the automata empties its stack when it is finished reading the input. We shall see that the two criteria for acceptance have equivalent power.

HOL Definition 4 (laes).

```
laes $M$ = {$w$ | $\exists$$state$. ($M$.start, $w$, [$M$.ssSym]) $\vdash^*_M$ ($state$, [], [])}
```

To be consistent with the notation in Hopcroft and Ullman, in what follows, function `laes` is also referred to as N (thus $N(M)$ is the language accepted by M using the empty stack criterion), and function `lafs` is simply L.

4. Equivalence of acceptance by final state and empty stack

The first property we establish is that the languages accepted by PDA by final state are exactly the languages accepted by PDA by empty stack. This is done by establishing that PDAs of one type can be emulated by PDAs of the other.

4.1. PDA construction for acceptance by empty stack

Theorem 4.1. *For every machine M_2 accepting language L by final state, there is a machine M_1 such that $N(M_1) = L$.*

Proof Let PDA $M_2 = (Q, \Sigma, \Gamma, \delta, q_0, m, Z_0, F)$. We invent new states q_e and q_0', and a new stack symbol X_0, and then let $M_1 = (Q \cup \{q_e, q_0'\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q_0', X_0, \phi)$, where δ' is defined by the following rules:

Rule 1 $\delta'(q_0', \epsilon, X_0) = (q0, Z_0X_0)$.

Rule 2 For all q in F, and Z in $\Gamma \cup X_0$, $\delta'(q, \epsilon, Z)$ contains (q_e, ϵ).

Rule 3 For all Z in $\Gamma \cup X_0$, $\delta'(q_e, \epsilon, Z)$ contains (q_e, ϵ).

Rule 4 $\delta'(q, a, Z)$ includes the elements of $\delta(q, a, Z)$ for all q, a and Z.

M_1 simulates M_2 by first putting a M_2's stack marker (Z_0) on its stack (Rule 1). Below this, M_2 preserves its own bottom of stack marker X_0. This ensures that M_1 does not accidentally accept if M_2 empties its stack without entering a final state. Rule 4 allows M_1 to process the input in exactly the same manner as M_2. Rule 2 gives M_1 the opportunity to enter the state q_e (triggering Rule 3) when M_2 enters a final state. Rule 3 allows M_1 to pop off the remaining stack contents once M_1 has accepted the input, thus accepting the input by empty stack criterion.

In HOL, we define a function `newm` that constructs the new machine. It takes not just the input machine M_2 as a parameter, but also new states (q'_0 and q_e) and the new stack symbol X_0.

HOL Definition 5 (`newm`).

```
newm M₂ (q'₀, X₀, qe) =
  (let d =
        [((NONE, X₀, q'₀), M₂.start, [M₂.ssSym; X₀])] ++ M₂.next ++
        finalStateTrans qe M₂.final (X₀::stkSymsList M₂ M₂.next) ++
        newStateTrans qe (X₀::stkSymsList M₂ M₂.next)
   in
     ⟨start := q'₀; ssSym := X₀; next := d; final := []⟩)
```

where `finalStateTrans` implements Rule 2 of the construction, and `newStateTrans` implements Rule 3.

We first prove that if $x \in L(M_2)$ then $x \in N(M_1)$. As x is accepted by final state, $(q_0, x, Z_0) \vdash^*_{M_2} (q, \epsilon, \gamma)$ for some $q \in F$, and $\gamma \in \Gamma^*$. Now consider M_1's behaviour on input x. Rule 1 gives $(q'_0, x, X_0) \vdash_{M_1} (q_0, x, Z_0X_0)$.

By Rule 2, every move of M_2 is a legal move for M_1, so we also have $(q_0, x, Z_0X_0) \vdash^*_{M_1} (q, \epsilon, \gamma X_0)$. In Hopcroft and Ullman, this last step is justified by

If a PDA can make a sequence of moves from a given ID, it can make the same sequence of moves from any ID obtained from the first by inserting a fixed string of stack symbols below the original stack contents.

Simple asides such as the above end up requiring a proof first that single transitions can have arbitrary symbols added underneath their stacks, followed by an induction to show

HOL Theorem 1.

$$(q, x, stk) \vdash^*_M (q', x', stk') \Rightarrow \forall \ell.\ (q, x, stk \mathbin{++} \ell) \vdash^*_M (q', x', stk' \mathbin{++} \ell)$$

Returning to the proof, by Rules 3 and 4, $(q, \epsilon, \gamma X_0) \vdash^*_{M_1} (q_e, \epsilon, \epsilon)$. Therefore, $(q_0', x, X_0) \vdash^*_{M_1} (q_e, \epsilon, \epsilon)$, and M_1 accepts x by empty stack. Our final HOL theorem is

HOL Theorem 2.

$X_0 \notin$ `stkSyms` $M_2 \land q_0' \notin$ `states` $M_2 \land qe \notin$ `states` $M_2 \Rightarrow$

$x \in$ `lafs` $M_2 \Rightarrow$

$x \in$ `laes (newm` M_2 `(`q_0'`,`X_0`,`qe`))`

(Note the requirement that the new states and stack symbol must be suitably fresh.)

The converse, if $x \in N(M_1)$ then $x \in L(M_2)$, is straightforward, and we can then conclude with a HOL version of Theorem 4.1

HOL Theorem 3.

`INFINITE` $\mathcal{U}$`(:'ssym)` $\land$ `INFINITE` $\mathcal{U}$`(:'state)` $\Rightarrow$

$\forall M_2.\ \exists M_1.$ `lafs` M_2 `=` `laes` M_1

($\mathcal{U}$`(:'ssym)` is the universal set of all possible stack symbols and $\mathcal{U}$`(:'state)` is the universal set of all possible states.)

The two extra conditions in the premise of the HOL statement are sufficient to ensure that we will always be able to pick fresh states q_0' and q_e, as well a fresh stack symbol X_0. The requirement to be explicit with details such as this are entirely typical of the mechanisation process.

4.2. PDA construction for acceptance by final state

Briefly, we present the construction of a "final-state-accepting" PDA that accepts the same inputs as an "empty-stack-accepting" PDA.

Theorem 4.2. *If L is $N(M_1)$ for some PDA M_1, then L is $L(M_2)$ for some PDA M_2.*

Proof We simulate M_1 using M_2 and detect when M_1 empties its stack, M_2 enters a final state when and only when this occurs. Let PDA M_1 be $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, \phi)$. Given fresh states q_0' and q_f, and a fresh stack symbol X_0, let $M_2 = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta', q_0', X_0, \{q_f\})$, where δ' is defined by the following rules:

Rule 1 $\delta'(q_0', \epsilon, X_0) = \{(q_0, Z_0X_0)\}$.

Rule 2 for all q in Q, a in $\Sigma \cup \epsilon$, and Z in Γ, $\delta'(q, a, Z) = \delta(q, a, Z)$.

Rule 3 for all q in Q, $\delta'(q, \epsilon, X_0)$ contains (q_f, ϵ).

HOL Definition 6 (`newm'`).

```
newm' $M_1$ ($q_0'$, $X_0$, $qf$) =
  (let $d$ =
        [((NONE,$X_0$,$q_0'$),$M_1$.start,[$M_1$.ssSym; $X_0$])] ++
        $M_1$.next ++
        MAP (toFinalStateTrans $X_0$ $qf$) (statesList $M_1$)
   in
     ⟨start := $q_0'$; ssSym := $X_0$; next := $d$; final := [$qf$]⟩)
```

Rule 1 causes M_2 to enter the initial ID of M_1, except that M_2 will have its own bottom-of-stack marker X_0, which is below the symbols of M_1's stack. Rule 2 allows M_2 to simulate M_1. Should M_1 ever erase its entire stack, then M_2, when simulating M_1, will erase its entire stack except the symbol X_0 at the bottom. Rule 3 causes M_2, when the X_0 appears, to enter a final state thereby accepting the input x.

We proceed in a similar manner to the proof of Theorem 4.1, establishing $L(M_2) = N(M_1)$:

HOL Theorem 4.

```
INFINITE $\mathcal{U}$(:'ssym) ∧ INFINITE $\mathcal{U}$(:'state) ⇒
∀$M_1$. ∃$M_2$. laes $M_1$ = lafs $M_2$
```

Again, we must use the precondition that the universes of the types of symbols and states are infinite.

5. Closure properties

Context-free languages are closed under the following operations (Hopcroft and Ullman, 1979). That is, if L and P are context-free languages and D is a regular language, the following languages are context-free as well:

- the Kleene star L^* of L
- the image (L) of L under a homomorphism
- the concatenation $L \frown P$ of L and P

- the union $L \cup P$ of L and P
- language obtained by substituting a language for a terminal in a second language

In this section we go through the HOL formalisation for proving closure of CFGs under union, concatenation, Kleene star operation, substitution. The closure under homomorphism follows from closure under the substitution operation.

We provide only a brief overview of the first two since they were straightforward to mechanise. First we establish the 'disjoint' property which allows renaming of variables in a grammar without affecting the language of the grammar. This forms the crucial part of proving the closure properties.

HOL Theorem 5.

```
INFINITE 𝒰(:α) ⇒
∃g′. L g = L g′ ∧ DISJOINT (nonTerminals g) (nonTerminals g′)
```

This theorem corresponds to the text statement *"we may rename variables at will without changing the language generated"* in Hopcroft and Ullman. This theorem is a necessary assumption for the closure properties that follow. Note also that, since we are renaming variables (by picking new ones), we need the premise that the type universe of the non-terminal symbols be infinite.

Closure properties typically merge rules of two different grammars in a particular way. For example, the union of two grammars, $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ results in grammar $G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$, where P_3 is $P_1 \cup P_2$ plus the productions $S \rightarrow S_1|S_2$. Here S is not in V_1 or V_2. In order to prove $L(G_1) \cup L(G_2) = L(G)$ we need to be able to distinguish the derivations of G_1 from G_2. This distinction is clear if the non-terminals of G_1 and G_2 do not overlap. Hence, the need for the disjoint property.

Proof We first define renaming a single variable. Function `rename` returns the new value (x') if x is the variable we are interested in, *i.e.* the variable e.

HOL Definition 7 (`rename`).

```
rename x x′ e = if e = x then x′ else e
```

Using `rename`, we can rename the non-terminal nt to nt' for a particular rule.

HOL Definition 8 (`ruleNt2Nt′`).

```
ruleNt2Nt′ nt nt′ (ℓ⤳r) =
  (rename nt nt′ ℓ⤳MAP (rename (NTS nt) (NTS nt′)) r)
```

Now given a new replacement value (nt') for a non-terminal nt, we systematically rename all nts to nt' in our old grammar `G` p s (we write `G` for the pairing function that takes a list of productions p and a start symbol s and returns the corresponding grammar). Note that we need to rename the start symbol as well. This is our function `grNt2Nt'`.

HOL Definition 9 (`grNt2Nt'`).

```
grNt2Nt′ nt nt′ (G p s) =
  G (MAP (ruleNt2Nt′ nt nt′) p) (rename nt nt′ s)
```

We then prove that such a single-step transformation preserves the language of the grammar.

HOL Theorem 6.

```
INFINITE 𝒰(:′nts) ⇒
NTS nt′ ∉ nonTerminals g ⇒
L g = L (grNt2Nt′ nt nt′ g)
```

Then, in order to get a new grammar g' starting from the old grammar g such that the non-terminals are disjoint, all we need to do is rename all the non-terminals in g such that the new names introduced are not part of g.

This is achieved by repeatedly renaming the non-terminals away, using the following (and the fact that the non-terminals of a grammar are a finite set):

HOL Theorem 7.

```
INFINITE 𝒰(:′nts) ⇒
∀s.
  FINITE s ⇒
  ∀g. ∃g′. L g′ = L g ∧ DISJOINT (nonTerminals g′) s
```

The proof for the disjoint property (unlike the one line statement that sufficed in the text) was ~440 lines of code.

After establishing this property we can now work on the various closure properties.

5.1. Union

Theorem 5.1. *Context-free languages are closed under union.*

Let L_1 and L_2 be CFLs generated by $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$, respectively. Since we may rename variables at will (proven above) without changing the language generated, we assume V_1 and V_2 are disjoint. Assume also that S_3 is not in V_1 or V_2.

For $L_1 \cup L_2$ construct grammar $G_3 = (V_1 \cup V_2 \cup S_3, T_1 \cup T_2, P_3, S_3)$, where P_3 is $P_1 \cup P_2$ plus the productions $S_3 \to S_1|S_2$. Given grammars G_1 and G_2, function `grUnion` constructs such a grammar G_3.

HOL Definition 10 (`grUnion`).

```
grUnion s0 g1 g2 =
  G
    (rules g1 ++ rules g2 ++ [s0⤳[NTS (startSym g1)]] ++
     [s0⤳[NTS (startSym g2)]]) s0
```

Proof If w is in L_1, then the derivation $S_3 \Rightarrow_{G_3} S_1 \Rightarrow^*_{G_1} w$ is a derivation in G_3, as every production of G_1 is a production of G_3. Similarly, every word in L_2 has a derivation in G_3 beginning with $S_3 \Rightarrow S_2$. Thus, $L_1 \cup L_2 \subseteq L(G_3)$.

For the converse let w be in $L(G_3)$. Then the derivation $S_1 \Rightarrow w$ begins with either $S_3 \Rightarrow_{G_3} S_1 \Rightarrow^*_{G_3} w$ or $S_3 \Rightarrow_{G_3} S_2 \Rightarrow^*_{G_3} w$. In the former case, as V_1 and V_2 are disjoint, only symbols of G_1 may appear in the derivation $S_1 \Rightarrow^*_{G_3} w$. Thus $S_1 \Rightarrow^*_{G_1} w$, and w is in L_1. Analogously, if the derivation starts $S_3 \Rightarrow^*_{G_3} S_2$, we may conclude w is in L_2. Hence, $L(G_3) \subseteq L_1 \cup L_2$, so $L(G_3) = L_1 \cup L_2$, as desired.

The HOL expression of Theorem 5.1 is:

HOL Theorem 8.

```
INFINITE 𝒰(:'nts) ⇒ ∀g1 g2. ∃g. L g = L g1 ∪ L g2
```

5.2. Concatenation

Theorem 5.2. *Context free grammars are closed under concatenation.*

Let L_1 and L_2 be CFLs generated by the CFGs $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$, respectively. Since we may rename variables at will without changing the language generated, we assume V_1 and V_2 are disjoint. Assume also that S_3 is not in V_1 or V_2.

For concatenation, let $G_3 = (V_1 \cup V_2 \cup S_3, T_1 \cup T_2, P_3, S_3)$ where P_3 is $P_1 \cup P_2$ plus the production $S_3 \to S_1 S_2$.

In HOL this is expressed using function `grConcat`.

HOL Definition 11 (`grConcat`).

```
grConcat s₀ g₁ g₂ =
  G
    (rules g₁ ++ rules g₂ ++
     [s₀ ⇝ [NTS (startSym g₁); NTS (startSym g₂)]]) s₀
```

A proof that $L(G_3) = L(G_1)L(G_2)$ follows.

Proof P_3 is $P_1 \cup P_2$ plus the productions $S_3 \to S_1S_2$. If w is in L_1L_2, then $w = w_1w_2$ such that w_1 is in L_1 and w_2 is in L_2. The derivation $S_3 \Rightarrow_{G_3} S_1S_2 \Rightarrow^*_{G_3} (w_1w_2)$ is a derivation in G_3, such that $S_1 \Rightarrow^* w_1$ and $S_2 \Rightarrow^* w_2$, as every production of both G_1 and G_2 is a production of G_3. Thus $L_1L_2 \subseteq L(G_3)$.

For the converse let w be in $L(G_3)$. Then the derivation $S_1 \Rightarrow w$ begins with $S_3 \Rightarrow_{G_3} S_1S_2 \Rightarrow^*_{G_3} w$. As V_1 and V_2 are disjoint, we can divide w into two parts, say w_1w_2 such that w_1 is derived from S_1 and w_2 from S_2.

Only symbols of G_1 may appear in the derivation $S_1 \Rightarrow^*_{G_3} w_1$. Thus $S_1 \Rightarrow^*_{G_1} w_1$, and w_1 is in L_1. Analogously we have $S_2 \Rightarrow^*_{G_3} w_2$ and we may conclude w_2 is in L_2. Hence, $L(G_3) \subseteq L_1L_2$, so $L(G_3) = L_1L_2$, as desired.

The statement for Theorem 5.2 in HOL is:

HOL Theorem 9.

```
INFINITE 𝒰(:'nts) ⇒ ∀g₁ g₂. ∃g. L g = conc (L g₁) (L g₂)
```

where

```
conc S₁ S₂ = {s | ∃u v. u ∈ S₁ ∧ v ∈ S₂ ∧ s = u ++ v}
```

5.3. Kleene closure

The Kleene closure of set P is represented by P^*. Let G be a grammar and let S be its start symbol. Then the Kleene closure of the language of G, $L(G)^*$, contains all the words generated using the grammar G_1 which contains all the rules from the original grammar plus the additional rules $S_0 \to SS_0$ and $S_0 \to \epsilon$. Here S_0 is the start symbol of G_1 and does not occur in G.

Theorem 5.3. *Context free languages are closed under Kleene closure.*

In HOL, the Kleene closure is defined using an inductive relation over the following rules:

HOL Definition 12 (`Kleene`).

$$\frac{}{\texttt{[]} \in A^*} \qquad \frac{s \in A}{s \in A^*} \qquad \frac{s_1 \in A \quad s_2 \in A^*}{s_1 \texttt{ ++ } s_2 \in A^*}$$

Let L be a CFL generated by the CFG $G = (V, T, P, S)$. We define a new grammar G_1 that generates all the strings which are in the Kleene closure of grammar G. Let $G_1 = (V \cup S_1, T, P_1, S_1)$ where P_1 is P plus the production $S_1 \rightarrow SS_1$ and $S_1 \rightarrow \epsilon$.

In HOL this construction is done using function `grClosure`.

HOL Definition 13 (`grClosure`).

```
grClosure $s_0$ $g$ =
  G (rules $g$ ++ [$s_0$ $\rightsquigarrow$ [NTS (startSym $g$); NTS $s_0$]] ++ [$s_0$ $\rightsquigarrow$ []])
    $s_0$
```

A proof that $L(G)^* = L(G_1)$ follows a similar methodology as used in proofs above. In HOL, Theorem 5.3 becomes

HOL Theorem 10.

`INFINITE` $\mathcal{U}$`(:'nts)` $\Rightarrow \forall g.\ \exists g'.\ L\ g' = (L\ g)^*$

5.4. Substitution

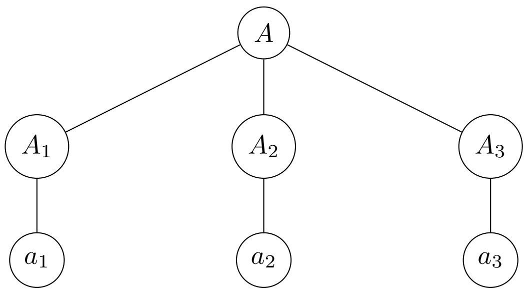
A more interesting closure proof is that of the substitution operation. The proof of this property is based on the notion of parse trees. We first present a brief overview of the implementation of parse trees in HOL.

5.4.1. Derivation (or parse) trees

In Section 2 we introduced derivation lists for modeling derivations in a grammar. Now we introduce derivation or parse trees, an alternate formalisation for representing derivations in a grammar. When explaining a derivation using pen and paper it is common to show multiple expansions in parallel. In such a case each derivation step involves a one-step expansion of all the non-terminals. The derivation

$$A \Rightarrow A_1A_2A_3 \Rightarrow a_1A_2A_3 \Rightarrow a_1a_2A_3 \Rightarrow a_1a_2a_3$$

in a grammar G, where a_1, a_2, a_3 are terminals and A,A_1,A_2,A_3 are non-terminals, can be represented using the following diagram.



Here, non-terminal A is the root node and terminals a_1, a_2, a_3 are the leaf nodes. The rules in G that allow this derivation are $A \to A_1A_2A_3$, $A_1 \to a_1$, $A_2 \to a_2$ and $A_3 \to a_3$.

This structure on the derivable strings in a grammar is called a *derivation tree* or a *parse tree*.

A tree is recursively defined as either a leaf node (no expansion possible hereafter) or a node which can expand to multiple derivation trees. In HOL:

```
('nts, 'ts) ptree
  = Leaf of 'ts | Node of 'nts ⇒ ('nts, 'ts) ptree list
```

The terms *leaves*, *fringe* or the *yield* of a tree all stand for the leaf nodes that do not have any children. This is slightly different from a leaf node represented as a node with an empty (ϵ) node as its only child, the definition in Hopcroft and Ullman.

HOL Definition 14 (`fringe`).

```
fringe (Leaf tm) = [tm]
fringe (Node x ptl) = FLAT (MAP (λ a. fringe a) ptl)
```

Relationship between derivation trees and derivations A tree is a correct derivation tree for a grammar if and only if it is *valid* with respect to the rules in the grammar (`validptree`). A tree is considered valid with respect to grammar G if each expansion step corresponds to some rule in G.

HOL Definition 15 (`validptree`).

```
validptree g (Node n ptl) ⟺
  n⤳getSymbols ptl ∈ rules g ∧
  ∀ e. e ∈ ptl ⇒ isNode e ⇒ validptree g e
validptree g (Leaf tm) ⟺ F
```

(`getSymbols` *ptl returns the symbols corresponding to the top level nodes of the parse*

tree list ptl. Function `isNode` *tree returns true if and only if tree is a node, i.e. corresponds to a non-terminal.)*

We differ slightly from Hopcroft and Ullman in what we consider to be a derivation tree. Hopcroft and Ullman state that for a tree to be a valid derivation tree for G, amongst other conditions, the root node has to be the start symbol of G and the root node derives a word. We instead define a looser version where in a derivation tree is valid as long as each expansion is a valid rule in G. Thus, the root node does not have to be the start symbol of G but the derived string has to be composed of only terminals.

If a tree is a valid parse tree with respect to a grammar then one can construct a corresponding derivation from the yield from the root non-terminal.

HOL Theorem 11.

$\texttt{validptree}\ g\ t \Rightarrow \texttt{[root}\ t\texttt{]} \Rightarrow_g^* \texttt{MAP TS (fringe}\ t\texttt{)}$

Similarly, if a terminal string can be derived from a non-terminal one can construct a parse tree for the derivation.

HOL Theorem 12.

$\texttt{derives}\ g \vdash dl \lhd \texttt{[NTS}\ A\texttt{]} \rightarrow y \Rightarrow$

$\texttt{isWord}\ y \Rightarrow$

$\exists t.\ \texttt{validptree}\ g\ t \wedge \texttt{MAP TS (fringe}\ t\texttt{)} = y \wedge \texttt{root}\ t = \texttt{NTS}\ A$

Theorem 5.4. *Let $G = (V, T, P, S)$ be a context-free grammar. Then $S \Rightarrow^* \alpha$ if and only if there is a derivation tree in grammar G with yield α.*

HOL Theorem 13.

$w \in L\ g \iff$

$\quad \exists t.$

$\quad\quad \texttt{validptree}\ g\ t \wedge \texttt{MAP TS (fringe}\ t\texttt{)} = w \wedge$

$\quad\quad \texttt{root}\ t = \texttt{NTS (startSym}\ g\texttt{)}$

With the help of the above framework we can now proceed with a proof of closure under substitution.

Theorem 5.5. *Context-free grammars are closed under substitution.*

Let $G = (V, T, P, S)$. The substitution involves creating a new grammar $G^{'}$ from the original grammar G in the following manner. The start symbol of G' is the same as the start symbol of G. Each terminal symbol a in G gets associated with another grammar G_a. This means that for every rule $A \to \alpha$ in G, any occurrence of a in α is substituted with the start symbol of grammar G_a. Thus, replacing terminal a in the words generated by G with any of the words of G_a gives us the words generated by $G^{'}$. We will show that the $L(G^{'}) = replace\ a\ w_a\ s_g$, where $replace$ substitutes the word w_a for terminal a in sentence s_g, $w_a \in L(G_a)$ and $s_g \in (V \cup T)^*$. Substitution is easy to visualise using the parse tree framework. Given a derivation tree for some input in the original grammar, the substitution operation results in replacing each of the leaf nodes by a derivation tree of some input belonging in the language of another grammar.

Proof Function `substGr` is responsible for the construction of G'. Here, $gsub$ is the grammar whose start symbol gets substituted for terminal tm in original grammar g. The substitution is done for each rule (`substRule`) in g. Again, without loss of generality we assume that non-terminals in g and $gsub$ are disjoint.

HOL Definition 16 (`substGr`).

```
substGr (tm, gsub) g =
  G
    (rules gsub ++
     MAP (substRule (TS tm,NTS (startSym gsub))) (rules g))
    (startSym g)
```

We then define the `replace` function in HOL. Function `replace` substitutes word s for symbol sym in the given sentence and returns a set of all possible substitutions.

HOL Definition 17 (`replace`).

```
replace [] sym s = {[]}
replace (NTS x::rst) sym s =
  IMAGE (CONS (NTS x)) (replace rst sym s)
replace (TS t::rst) sym s =
  if t ≠ sym then
    IMAGE (CONS (TS t)) (replace rst sym s)
  else
    conc s (replace rst sym s)
```

To prove the closure, we have to establish:

HOL Theorem 14.

```
DISJOINT (nonTerminals g) (nonTerminals gsub) ⇒
(w′ ∈ L (substGr (tm,gsub) g) ⟺
   ∃w. w ∈ L g ∧ w′ ∈ replace w tm (L gsub))
```

For the "if" direction we prove:

HOL Theorem 15.

```
DISJOINT (nonTerminals g) (nonTerminals gsub) ∧ w ∈ L g ∧
w′ ∈ replace w tm (L gsub) ⇒
w′ ∈ L (substGr (tm,gsub) g)
```

For the "only if" direction we use the notion of derivation trees to assert membership in the language of the grammar. For a derivation tree valid with respect to grammar $gsub$, one can construct a derivation tree valid with respect to grammar g such that replacing the terminal in yield of g by some yield w of $gsub$ gives a yield for G'.

HOL Theorem 16.

```
validptree (substGr (sym,gsub) g) t ∧
root t ∈ nonTerminals g ∧
DISJOINT (nonTerminals g) (nonTerminals gsub) ⇒
∃t′ w.
  MAP TS (fringe t′) = w ∧
  MAP TS (fringe t) ∈ replace w sym (L gsub) ∧
  validptree g t′ ∧ root t′ = root t
```

The correspondence between derivation trees and derivations lets us derive the "only if" statement.

HOL Theorem 17.

```
DISJOINT (nonTerminals g) (nonTerminals gsub) ∧
w′ ∈ L (substGr (tm,gsub) g) ⇒
∃w. w ∈ L g ∧ w′ ∈ replace w tm (L gsub)
```

Thus, we now have the closure under substitution.

Corollary 5.6 (Closure under homomorphism). *The property that CFLs are closed under homomorphism follows directly from closure under substitution since homomorphism is just a special type of substitution.*

6. Constructing a PDA for a CFG

In the next two sections, we discuss the paper's most involved proof: that of the equivalence of PDAs and CFGs. In fact, constructing a PDA for a CFG is a straightforward process so most of the space is given to just one direction of the equivalence: the construction of a CFG from a PDA.

For the simpler direction, we follow Hopcroft and Ullman and assume the grammar is in Greibach normal form. Our mechanised proof that all grammars can be put into this normal form is discussed in our earlier conference paper (Barthwal and Norrish (2010)).

So, let $G = (V, T, P, S)$ be a context-free grammar in Greibach normal form generating L. We construct machine M such that $M = (q, T, V, \delta, q, S, \phi)$, where $\delta(q, a, A)$ contains (q, γ) whenever $A \rightarrow a\gamma$ is in P. Every production in a grammar that is in GNF has to be of the form $A \rightarrow a\alpha$, where a is a terminal symbol and α is a string (possibly empty) of non-terminal symbols (`isGnf`). The automaton for the grammar is constructed by creating transitions from the grammar productions, $A \rightarrow a\alpha$ that read the head symbol of the RHS (a) and push the remaining RHS (α) on to the stack. The terminals are interpreted as the input symbols and the non-terminals are the stack symbols for the PDA.

```
trans q (ℓ⇝r) = ((SOME (HD r),NTS ℓ,q),q,TL r)
grammar2pda g q =
  (let ts = MAP (trans q) (rules g)
   in
     ⟨start := q; ssSym := NTS (startSym g); next := ts;
      final := []⟩)
```

(Here `HD` returns the first element in the list and `TL` returns the remaining list. Function `MAP` applies a given function to each element of a list.)

The PDA M simulates leftmost derivations of G. Since G is in Greibach normal form, each sentential form in a leftmost derivation consists of a string of terminals x followed by a string of variables α. M stores the suffix α of the left sentential form on its stack after processing the prefix x. Formally we show that

$$S \overset{l}{\Rightarrow}^* x\alpha \text{ by a leftmost derivation if and only if } (q, x, A) \ \rightarrow_M^* \ (q, \epsilon, \alpha) \tag{1}$$

This turns out to be straightforward process in HOL and is done by representing the grammar and the machine derivations using derivation lists. Let dl represent the grammar derivation from S to $x\alpha$ and dl' represent the derivation from (q, x, A) to (q, ϵ, α) in the machine. Then an induction on dl gives us the "if" portion of (1) and induction on dl' gives us the "only if" portion of (1). Thus, we can conclude the following,

HOL Theorem 18.

$\forall g.$ `isGnf` $g \Rightarrow \exists m.\ \forall x.\ x \in L\ g \iff x \in$ `laes` m

7. Constructing a CFG from a PDA

The CFG for a PDA is constructed by encoding every possible transition step in the PDA as a rule in the grammar. The LHS of each production encodes the starting and final state of the transition while the RHS encodes the contents of the stack in the final state.

Let M be the PDA $(Q, \delta, q_0, Z_0, \phi)$ and Σ and Γ the derived input and stack alphabets, respectively. We construct $G = (V, \Sigma, P, S)$ such that V is a set containing the new symbol S and objects of the form $[q, A, p]$; for q and p in Q, and A in Γ.

The productions P are of the following form: (**Rule 1**) $S \rightarrow [q_0, Z_0, q]$ for each q in Q; and (**Rule 2**) $[q, A, q_{m+1}] \rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3]...[q_m, B_m, q_{m+1}]$ for each $q, q_1, q_2, ..., q_{m+1}$ in Q, each a in $\Sigma \cup \{\epsilon\}$, and $A, B_1, B_2, ..., B_m$ in Γ, such that $\delta(q, a, A)$ contains $(q_1, B_1 B_2 ... B_m)$ (if $m = 0$, then the production is $[q, A, q_1] \rightarrow a$). The variables and productions of G have been defined so that a leftmost derivation in G of a sentence x is a simulation of the PDA M when fed the input x. In particular, the variables that appear in any step of a leftmost derivation in G correspond to the symbols on the stack of M at a time when M has seen as much of the input as the grammar has already generated.

From text to automated text:. For **Rule 1** we only have to ensure that the state q is in Q. On the other hand, there are multiple constraints underlying the statement of **Rule 2** which will need to be isolated for mechanisation and are summarised below.

C2.1 The states q, q_1 and p belong in Q (a similar statement for terminals and non-terminals can be ignored since they are derived);

C2.3 the corresponding machine transition is based on the values of a and m and steps from state q to some state q_1 replacing A with $B_1...B_m$;

C2.3 the possibilities of generating the different grammar rules based on whether $a = \epsilon$, $m = 0$ or a is a terminal symbol;

C2.4 if $m > 1$ *i.e.* more than one non-terminal exists on the RHS of the rule then

- **C2.4.1** α is composed of only non-terminals;
- **C2.4.2** a non-terminal is an object of the form $[q, A, p]$ for PDA from-state q and to-state p, and stack symbol A;
- **C2.4.3** the from-state of the first object is q_1 and the to-state of the last object is q_{m+1};
- **C2.4.4** the to-state and from-state of adjacent non-terminals must be the same;
- **C2.4.5** the states encoded in the non-terminals must belong to Q.

Whether we use a functional approach or a relational one, the succinctness of the above definition is hard to capture in HOL. Using relations we can avoid concretely computing every possible rule in the grammar and thus work at a higher level of abstraction. The extent of details to follow are characteristic of mechanising such a proof. The relation `pda2grammar` captures the restrictions on the rules for the grammar corresponding to a PDA.

HOL Definition 18.

```
pda2grammar M g ⟺
  pdastate (startSym g) ∉ statesList M ∧
  set (rules g) = p2gStartRules M (startSym g) ∪ p2gRules M
```

The non-terminals are a tuple of a from-state, a stack symbol and a to-state, the states and the stack symbols belonging to the PDA. As long as one of the components is not in the PDA, our start symbol will be new and will not overlap with the symbols constructed from the PDA. The first conjunct of `pda2grammar` ensures this. The function `p2gStartRules` corresponds to **Rule 1** and the function (`p2gRules`) ensures that each rule conforms with **Rule 2**. As already mentioned, **Rule 2** turns out to be more complicated to mechanise due to the amount of detail hidden behind the concise notation.

The p2gRules predicate (see Figure 1) enforces the conditions **C2.1**, **C2.2**, **C2.3** (capturing the four possibilities for a rule, $A \rightarrow \epsilon$; $A \rightarrow a$, $A \rightarrow a\alpha$, where a is a terminal symbol and $A \rightarrow \alpha$ for non-terminals α).

HOL Definition 19.

```
p2gRules M =
  {(q,A,q₁) ⇝ [] | ((NONE,A,q),q₁,[]) ∈ M.next} ∪
  {(q,A,q₁) ⇝ [TS ts] | ((SOME (TS ts),A,q),q₁,[]) ∈ M.next} ∪
  {(q,A,p) ⇝ [TS ts] ++ L |
   L ≠ [] ∧
   ∃mrhs q₁.
     ((SOME (TS ts),A,q),q₁,mrhs) ∈ M.next ∧
     ntslCond M (q₁,p) L ∧ MAP transSym L = mrhs ∧
     p ∈ statesList M} ∪
  {(q,A,p) ⇝ L |
   L ≠ [] ∧
   ∃mrhs q₁.
     ((NONE,A,q),q₁,mrhs) ∈ M.next ∧ ntslCond M (q₁,p) L ∧
     MAP transSym L = mrhs ∧ p ∈ statesList M}
```

Figure 1: Definition of p2gRules.

Condition ntslCond captures **C2.4** by describing the structure of the components making up the RHS of the rules when α is nonempty (*i.e.* has one or more non-terminals). The component $[q, A, p]$ is interpreted as a non-terminal symbol and q (frmState) and p (toState) belong in the states of the PDA (**C2.4.2**), the conditions on q' and q_l that reflects **C2.4.3** condition on q_1 and q_{m+1} respectively, **C2.4.4** using relation adj and **C2.4.5** using the last conjunct.

HOL Definition 20.

$\texttt{ntslCond}\ M\ \texttt{(}q'\texttt{,}ql\texttt{)}\ ntsl \iff$

$\quad\texttt{EVERY isNonTmnlSym}\ ntsl\ \wedge$

$\quad(\forall e_1\ e_2\ p\ s.\ ntsl = p\ \texttt{++}\ [e_1\texttt{;}\ e_2]\ \texttt{++}\ s \Rightarrow \texttt{adj}\ e_1\ e_2)\ \wedge$

$\quad\texttt{frmState (HD}\ ntsl\texttt{)} = q'\ \wedge\ \texttt{toState (LAST}\ ntsl\texttt{)} = ql\ \wedge$

$\quad(\forall e.\ e \in ntsl \Rightarrow \texttt{toState}\ e \in \texttt{statesList}\ M)\ \wedge$

$\quad\forall e.\ e \in ntsl \Rightarrow \texttt{frmState}\ e \in \texttt{statesList}\ M$

(The `LAST` *function returns the last element in a list.)*

The constraints described above reflect exactly the information corresponding to the two criteria for the grammar rules. On the other hand, it is clear that the automated definition looks and is far more complex to digest. Concrete information that is easily gleaned by a human reader from abstract concepts has to be explicitly stated in a theorem prover.

Now that we have a CFG for our machine we can plunge ahead to prove the following.

Theorem 7.1. *If L is $N(M)$ for some PDA M, then L is a context-free language.*

To show that $L(G) = N(M)$, we prove by induction on the number of steps in a derivation of G or the number of moves of M that

$$(q, x, A)\ \rightarrow_M^*\ (p, \epsilon, \epsilon) \text{ iff } [q, A, p] \overset{l}{\Rightarrow}_G^* x\ . \qquad (2)$$

7.1. Proof of the "if" portion of (2)

First we show by induction on i that if $(q, x, A) \rightarrow^i (p, \epsilon, \epsilon)$, then $[q, A, p] \Rightarrow^* x$.

HOL Theorem 19.

$\texttt{ID}\ M \vdash dl \lhd \texttt{(}q\texttt{,}x\texttt{,[}A\texttt{])} \rightarrow \texttt{(}p\texttt{,[],[])}\ \wedge\ \texttt{isWord}\ x\ \wedge$

$\texttt{pda2grammar}\ M\ g \Rightarrow$

$\texttt{[NTS (}q\texttt{,}A\texttt{,}p\texttt{)]} \Rightarrow_g^* x$

Proof The proof is based on induction on the length of dl. The crux of the proof is breaking down the derivation such that a single stack symbol gets popped off after reading some (possibly empty) input.

Let $x = a\gamma$ and $(q, a\gamma, A) \rightarrow (q_1, \gamma, B_1B_2...B_n) \rightarrow^{i-1} (p, \epsilon, \epsilon)$. The single step is easily derived based on how the rules are constructed. For the $i - 1$ steps, the induction hypothesis can be applied as long as the derivations involve a single symbol on the stack. The string γ can be written $\gamma = \gamma_1\gamma_2...\gamma_n$ where γ_i has the effect of popping B_j from the stack, possibly after a long sequence of moves. Note that B_1 need not be the n^{th} stack symbol from the bottom during the entire time γ_1 is being read by M. In general, B_j remains on the stack unchanged while $\gamma_1, \gamma_2...\gamma_{j-1}$ is read. There exist states $q_2, q_3, ..., q_{n+1}$, where $q_{n+1} = p$, such that $(q_j, \gamma_j, B_j) \;\; \rightarrow^* \;\; (q_j, \epsilon, \epsilon)$ by fewer than i moves (q_j is the state entered when the stack first becomes as short as $n - j + 1$). These observations are easily assumed by Hopcroft and Ullman or for that matter any human reader. The more concrete construction for mechanisation is as follows.

***Filling in the gaps*:.** For a derivation of the form, $(q_1, \gamma, B_1B_2...B_n) \;\; \rightarrow^i \;\; (p, \epsilon, \epsilon)$, this is asserted in HOL by constructing a list of objects $(q0, \gamma_j, B_j, q_n)$ (combination of the object's from-state, input, stack symbols and to-state), such that $(q0, \gamma_j, B_j) \;\; \rightarrow^i \;\; (q_n, \epsilon)$, where $i > 0$, γ_j is input symbols reading which stack symbol B_j gets popped off from the stack resulting in the transition from state q_0 to q_n. The from-state of the first object in the list is q_1 and the to-state of the last object is p. Also, for each adjacent pair e_1 and e_2, the to-state of e_1 is the same as the from-state of e_2. This process of popping off the B_j stack symbol turns out to be a lengthy one and is reflected in the proof statement of HOL Theorem 20.

To be able to prove this, it is necessary to provide the assertion that each derivation in the PDA can be divided into two parts, such that the first part (list dl_0) corresponds to reading n input symbols to pop off the top stack symbol. This is our HOL Theorem 21.

The proof of above is based on another HOL theorem that if $(q, \gamma\eta, \alpha\beta) \rightarrow^i (q', \eta, \beta)$ then we can conclude $(q, \gamma, \alpha) \;\; \rightarrow^i \;\; (q', \epsilon, \epsilon)$ (proved in HOL). This is a good example of a proof where most of the reasoning is "obvious" to the reader. This when translated into a theorem prover results in a cascading structure where one has to provide the proofs for steps that are considered "trivial". The gaps outlined here are just the start of the bridging process between the text proofs and the mechanised proofs.

***Proof resumed*:.** Once these gaps have been taken care of, we can apply the inductive hypothesis to get

$$[q_j, B_j, q_{j+1}] \overset{l}{\Rightarrow}^* \gamma_j \text{ for } 1 \leq j \leq n. \tag{3}$$

This leads to, $a[q_1, B, q_2][q_2, B_2, q_3]...[q_n, B_n, q_{n+1}] \overset{l}{\Rightarrow}^* x$.

HOL Theorem 20.

$$
\begin{array}{l}
\texttt{ID}\ M \vdash dl \lhd (q, inp, stk) \to (qf, [\,], [\,]) \Rightarrow \\
\exists \ell. \\
\quad inp = \texttt{FLAT}\ (\texttt{MAP tupinp}\ \ell) \land stk = \texttt{MAP tupstk}\ \ell \land \\
\quad (\forall e.\ e \in \texttt{MAP tuptost}\ \ell \Rightarrow e \in \texttt{statesList}\ M) \land \\
\quad (\forall e.\ e \in \texttt{MAP tupfrmst}\ \ell \Rightarrow e \in \texttt{statesList}\ M) \land \\
\quad (\forall h\ t. \\
\qquad \ell = h :: t \Rightarrow \\
\qquad \texttt{tupfrmst}\ h = q \land \texttt{tupstk}\ h = \texttt{HD}\ stk \land \\
\qquad \texttt{tuptost}\ (\texttt{LAST}\ \ell) = qf) \land \\
\quad \forall e_1\ e_2\ pfx\ sfx. \\
\qquad \ell = pfx \mathbin{++} [e_1;\ e_2] \mathbin{++} sfx \Rightarrow \\
\qquad \texttt{tupfrmst}\ e_2 = \texttt{tuptost}\ e_1 \land \\
\qquad \forall e. \\
\qquad\quad e \in \ell \Rightarrow \\
\qquad\quad \exists m. \\
\qquad\qquad m < |dl| \land \\
\qquad\qquad \texttt{NRC}\ (\texttt{ID}\ M)\ m\ (\texttt{tupfrmst}\ e, \texttt{tupinp}\ e, [\texttt{tupstk}\ e]) \\
\qquad\qquad\quad (\texttt{tuptost}\ e, [\,], [\,])
\end{array}
$$

(Relation NRC $R\ m\ x\ y$ is the RTC closure of R from x to y in m steps.)

HOL Theorem 21.

$$
\begin{array}{l}
\texttt{ID}\ p \vdash dl \lhd (q, inp, stk) \to (qf, [\,], [\,]) \Rightarrow \\
\exists dl_0\ q_0\ i_0\ s_0\ spfx. \\
\quad \texttt{ID}\ p \vdash dl_0 \lhd (q, inp, stk) \to (q_0, i_0, s_0) \land |s_0| = |stk| - 1 \land \\
\quad (\forall q'\ i'\ s'.\ (q', i', s') \in \texttt{FRONT}\ dl_0 \Rightarrow |stk| \le |s'|) \land \\
\quad ((\exists dl_1. \\
\qquad\quad \texttt{ID}\ p \vdash dl_1 \lhd (q_0, i_0, s_0) \to (qf, [\,], [\,]) \land |dl_1| < |dl| \land \\
\qquad\quad |dl_0| < |dl|) \lor \\
\qquad (q_0, i_0, s_0) = (qf, [\,], [\,]))
\end{array}
$$

(Predicate FRONT ℓ *returns the list ℓ minus the last element.)*

Since $(q, a\gamma, A) \rightarrow (q_1, \gamma, B_1B_2...B_n)$, we know

$$[q, A, p] \overset{l}{\Rightarrow} a[q_1, B, q_2][q_2, B_2, q_3]...[q_n, B_n, q_{n+1}]$$

and so finally we can conclude

$$[q, A, p] \overset{l}{\Rightarrow}^* a\gamma_1\gamma_2...\gamma_n = x$$

The overall structure of the proof follows Hopcroft and Ullman, though at rather greater length. The proofs in this section were quite involved, and we have only shown a small subset of them due to space restrictions.

7.2. Proof of the "only if" portion of (2)

Now suppose $[q, A, p] \Rightarrow^i x$. We show by induction on i that $(q, x, A) \rightarrow^* (p, \epsilon, \epsilon)$.

HOL Theorem 22.

```
derives g ⊢ dl ◁ [NTS (q,A,p)] → x ∧ q ∈ statesList M ⇒
isWord x ⇒
pda2grammar M g ⇒
(q,x,[A]) ⊢*_M (p,[],[])
```

Proof The basis, $i = 1$, is immediate, since $[q, A, p] \rightarrow x$ must be a production of G and therefore $\delta(q, x, A)$ must contain (p, ϵ). Note x is ϵ or in Σ here. In the inductive step, there are three cases to be considered. The first is the trivial case, $[q, A, p] \Rightarrow a$, where a is a terminal. Thus, $x = a$ and $\delta(q, a, A)$ must contain (p, ϵ). The other two possibilities are

$$[q, A, p] \Rightarrow a[q_1, B_1, q_2]...[q_n, B_n, q_{n+1}] \Rightarrow^{i-1} x$$

where $q_{n+1} = p$ or

$$[q, A, p] \Rightarrow [q_1, B_1, q_2]...[q_n, B_n, q_{n+1}] \Rightarrow^{i-1} x$$

where $q_{n+1} = p$. The latter case can be considered a specialisation of the first one such that $a = \epsilon$. Then x can be written as $x = ax_1x_2...x_n$, where $[q_j, B_j, q_{j+1}] \Rightarrow^* x_j$ for $1 \leq j \leq n$ and possibly $a = \epsilon$. This has to be formally asserted in HOL. Let α be of length n. If $\alpha \Rightarrow^m \beta$, then α can be divided into n parts, $\alpha = \alpha_1\alpha_2...\alpha_n$ and $\beta = \beta_1\beta_2...\beta_n$, such that $\alpha_i \Rightarrow^i \beta_i$ in $i \leq m$ steps.

HOL Theorem 23.

```
derives $g$ ⊢ $dl$ ◁ $x$ → $y$ ⇒
∃ℓ.
  $x$ = MAP FST ℓ ∧ $y$ = FLAT (MAP SND ℓ) ∧
  ∀ $a$ $b$.
    ($a$,$b$) ∈ ℓ ⇒
    ∃ $dl'$. $|dl'|$ ≤ $|dl|$ ∧ derives $g$ ⊢ $dl'$ ◁ [$a$] → $b$
```

(The `FLAT` *function returns the elements of (nested) lists,* `SND` *returns the second element of a pair.)*

Inserting $B_{j+1}...B_n$ at the bottom of each stack in the above sequence of IDs gives us,

$$(q_j, x_j, B_jB_{j+1}...B_n) \ \rightarrow^* \ (q_{j+1}, \epsilon, B_{j+1}...B_n). \tag{4}$$

The first step in the derivation of x from $[q, A, p]$ gives us,

$$(q, x, A) \ \rightarrow \ (q_1, x_1x_2...x_n, B_1B_2...B_n) \tag{5}$$

is a legal move of M. From this move and (4) for $j = 1, 2, ..., n$, $(q, x, A) \ \rightarrow^* \ (p, \epsilon, \epsilon)$ follows. In Hopcroft and Ullman, the above two equations suffice to deduce the result we are interested in.

Unfortunately, the sequence of reasoning here is too coarse-grained for HOL4 to handle. The intermediate steps need to be explicitly stated for the proof to work out using a theorem prover.

These steps can be further elaborated as follows.[1] By our induction hypothesis,

$$(q_j, x_j, B_j) \ \rightarrow^* \ (q_{j+1}, \epsilon, \epsilon). \tag{6}$$

Now consider the first step, if we insert $x_2...x_n$ after input x_1 and $B_2...B_n$ at the bottom of each stack, we see that

$$(q_1, x_1...x_n, B_1...B_n) \ \rightarrow^* \ (p, \epsilon, \epsilon). \tag{7}$$

Another fact that needs to be asserted explicitly is reasoning for (7).

This is done by proving the affect of inserting input/stack symbols on the PDA transitions. Now from the first step, (5) and (7), $(q, x, A) \ \rightarrow^* \ (p, \epsilon, \epsilon)$ follows.

[1]Their HOL versions can be found as part of the source code

Equation (2) with $q = q_0$ and $A = Z_0$ says $[q_0, Z_0, p] \;\Rightarrow^*\; x$ iff $(q_0, x, Z_0) \;\rightarrow^*\; (p, \epsilon, \epsilon)$. This observation, together with **Rule 1** of the construction of G, says that $S \;\Rightarrow^*\; x$ if and only if $(q_0, x, Z_0) \;\rightarrow^*\; (p, \epsilon, \epsilon)$ for some state p. That is, x is in $L(G)$ if and only if x is in $N(M)$ and we have

HOL Theorem 24.

$\texttt{pda2grammar}\ M\ g \land \texttt{isWord}\ x \Rightarrow$

$\texttt{([NTS (startSym}\ g\texttt{)]}\ \Rightarrow_g^*\ x \iff$

$\quad \exists p.\ (M.\texttt{start},x,\texttt{[}M.\texttt{ssSym])}\ \vdash_M^*\ \texttt{(}p\texttt{,[],[]))}$

To avoid the above being vacuous, we additionally prove the following:

HOL Theorem 25.

$\texttt{INFINITE}\ \mathcal{U}\texttt{(:'pdastate)} \Rightarrow \forall m.\ \exists g.\ \texttt{pda2grammar}\ m\ g$

The pre-condition is on the type of state in the PDA. This is necessary to be a able to choose a fresh state (not in the PDA) to create the start symbol of the grammar as mentioned before.

8. Related work

In the general area of mechanised language theory, the earliest work we are aware of is by Nipkow (1998). That paper describes a verified and executable lexical analyzer generator. Focusing on core language theory as it does, we feel this work is the closest in nature to our own mechanisation, though of course it covers regular rather than context-free languages.

Apart from our own earlier work on SLR parsing, there have recently been a number of papers directly concerned with mechanisation and verification of specific approaches to parsing. For example, Koprowski and Binsztok (2011) describe the construction of a verified parser for *expression grammars*. This formalism allows for parsers for a large class of languages to be generated from natural specifications that look a great deal like context-free grammars. Another example is Ridge (2011), which work presents a verified parser for all possible context-free grammars, using an admirably simple algorithm. The drawback is that, as presented, the algorithm is of complexity $O(n^5)$.

Parser combinators, popular in functional programming languages, are another approach to the general parsing problem, and there has been some mechanisation work in this area. For example, Danielsson (2010) presents a library of parser combinators that have been verified (in the Agda system) to guarantee termination of parsing.

Theory	**LOC**	**#Definitions**	**#Proofs**
CFGs—background	3680	36	189
PDAs—background	1846	15	47
Empty Stack Acceptance $\iff$ Final State Acceptance	1795	6	75
PDA $\iff$ CFG	2598	16	50
Closure properties	1686	12	91

Table 1: Summary of proof effort

9. Conclusions

We have mechanised a large portion of the background theory of context-free grammars and pushdown automata. This theory is fundamental to an important part of computer science, and had not been mechanised previously. The work was pursued as part of a larger project (the first author's PhD): to mechanise one of the field's standard textbooks. And, just as a textbook lays the foundation for future research, we hope our mechanised theories provide a platform for others' work in the area of context-free languages.

Inevitably, the mechanisation work occasioned a great many instances of "proof blowup", where something handled briefly in prose turns out to require a great deal of toil. Most of this was simple tedium (necessary inductions are very easy for humans to glide past); at other times, the effort seem to require rather more human ingenuity. A numerical summary of the blood, sweat and tears *per* theory appears in Table 1. The total person-time taken was in the order of one PhD student multiplied by 6 months.

Despite these issues around proof-size explosion, we also hope our mechanisation might be a good basis for teaching language theory. Others, such as Blanc et al. (2007) and Pierce (2010) have used the Coq system as part of courses on logic, mathematics and topics in theoretical computer science. Given its central part in the curriculum, and given our success in the mechanisation of Hopcroft and Ullman, we now believe language theory to be an area well-suited to mechanised pedagogy.

References

Barthwal, A., Norrish, M., March 2009. Verified, executable parsing. In: Castagna, G. (Ed.), Programming Languages and Systems: 18th European Symposium on Programming. Vol. 5502 of Lecture Notes in Computer Science. Springer, pp. 160–174.

Barthwal, A., Norrish, M., August 2010. A formalisation of the normal forms of context-free grammars in HOL4. In: Dawar, A., Veith, H. (Eds.), Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings. Vol. 6247 of Lecture Notes in Computer Science. Springer, pp. 95–109.

Blanc, J., Giacometti, J., Hirschowitz, A., Pottier, L., Jun. 2007. Proofs for freshmen with Coqweb. In: Geuvers, H., Courtieu, P. (Eds.), Proceedings of the International Workshop on Proof Assistants and Types in Education, June 25th, 2007, associated workshop of the 2007 Federated Conference on Rewriting, Deduction and Programming. CNAM Paris, France, pp. 93–107.
URL `http://www.cs.ru.nl/~herman/PUBS/proceedingsPATE.pdf`

Danielsson, N. A., 2010. Total parser combinators. In: Proceedings of the International Conference on Functional Programming, ICFP 2010. ACM, New York, NY, USA, pp. 285–296.
URL `http://doi.acm.org/10.1145/1863543.1863585`

Gordon, M. J. C., Melham, T. (Eds.), 1993. Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press.

Hopcroft, J. E., Ullman, J. D., 1979. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Ma., USA.

Koprowski, A., Binsztok, H., 2011. TRX: a formally verified parser interpreter. Logical Methods in Computer Science 7 (2), 1–26.
URL `http://dx.doi.org/10.2168/LMCS-7(2:18)2011`

Nipkow, T., 1998. Verified lexical analysis. In: Grundy, J., Newey, M. (Eds.), Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98). Vol. 1479 of Lecture Notes in Computer Science. Springer, Canberra, Australia, pp. 1–15.
URL `citeseer.ist.psu.edu/nipkow98verified.html`

Pierce, B. C., Jul. 2010. Proof assistant as teaching assistant: A view from the trenches. Keynote address at *International Conference on Interactive Theorem Proving (ITP)*.

Ridge, T., December 2011. Simple, functional, sound and complete parsing for all context-free grammars. In: Jouannaud, J.-P., Shao, Z. (Eds.), Proceedings of Certified Programs and Proofs, CPP 2011. Vol. 7086 of Lecture Notes in Computer Science. Springer, pp. 103–118.

Slind, K., Norrish, M., 2008. A brief overview of HOL4. In: Mohamed, O. A., Muñoz, C., Tahar, S. (Eds.), Theorem Proving in Higher Order Logics. Vol. 5170 of Lecture Notes in Computer Science. Springer, pp. 28–32, see also the HOL website at `http://hol.sourceforge.net`.