A formalisation of the theory of context-free languages in higher order logic

A thesis submitted for the degree of
Doctor of Philosophy
of
The Australian National University

Aditi Barthwal

December 2010

This document was produced using TEX , LATEX and BIBTEX

I declare that the work in this thesis is entirely my own and that to the best of my knowledge it does not contain any materials previously published or written by another person except where otherwise indicated.

Aditi Barthwal
16 December 2010

To Daryel, the Keeper

Acknowledgements

The Little Person and the Quest for the PhD

Starring

Me as *Little Person (LP)*

Dr Michael Norrish in a double role as *Michael, Master of HOL* and *One of The Proof Readers*

Dr Daryel Akerlind as *Keeper*

Ms Katie Helen Sullivan as *Rani*

Srinivas as *TeXi*

My Mac as *MacDino*

S H Arkie as *Sharkie*

Fishy as Himself

Once upon a time, in a faraway land, LP lost herself, most likely the part that was the mind. (The quest for *P*ermanent *H*ead *D*amage is not to be taken lightly. It wrings you dry, it tears you apart...and that is just the beginning...or so they say...) They told her, *'go, search for this elixir. It is hard to find but when you have drunk of it, you shall be close to divine'*. And so LP left on a quest for the PhD, the most elusive.

She was small, and the road was long. With forests to brave and mountains to scale, she wanted someone to show her the way. LP met Michael, and he said *'I will help you. I will take you under my wing. I will teach you to hold the Sword of Reason and to slay the Proof monster'*. And so began the journey, long and hard...

The Little Person walked through the jungle and slayed the monsters but they were many. Michael, the Master of HOL, looked after the HOL universe. He would tell Little Person secrets of the dark Goal. He gave her Meeting, not once but many a times. But as the sun set every day, Michael had to go back, back to Where He Came From.

MacDino whirred, hot it got. But no matter what LP tried, in the sun-less night, the Doubt monster would creep closer and closer. LP ran and ran, and finally found the Keeper, the Keeper of Night and Day. The Keeper said, *'tell me, tell me where you want to go. I cannot step on to the Path, it is yours to tread, but I will help you find the Way'*.

'I do not have Understanding', said LP.

Keeper, *'You can find it, but you have to pay. The cost is heavy, but Steady, that is the Key'*.

LP too eager as always, *'I will find a way to pay, tell me how...'*

'Many an Hour, Night and Day', Keeper replied.

And so the deal was sealed. Into the dark night, she worked. The bright days passed, Gloom descended. Keeper was tired and LP wanted a friend. She wanted to chase the monsters away and lo behold Rani came running her way. Every time when the monster Hard knocked on the door, They put on the Belt, laced their Shoes, drank Energy and off they went together to kill Doubt and No-Result. And guess who they found romping around, it was Bling! And everyone knows Bling always travels with Work! And everyone knows, when you meet Work, the mountain where PhD is hidden is not far away. When LP and Rani were tired after their romps, TEXi would would feed them. The food was always Hot and Wholesome and so they would begin the next Long Run with a singing belly.

But Work is quiet, it is a solitary traveller, so can be boring company. LP and Rani wanted to find a Fun friend. Work was too busy so LP and Rani always looked out whenever they stopped to take a breath. They looked in the Pool, at the Wall, even in the TV but good things take time.

And one day it happened. It is not everyday, that one finds Sharkie. An elusive creature, to say the least, Sharkie only travels with Fishy, his constant regenerating companion, always with a good supply of OMEGA 3 on his hand. They are held by a Thread, but one that can never be broken. Once again, as everyone knows, Sharkie can light up Day and Night. His pizazz (more likely the jaws) have driven many a monsters away especially Grumpy.

And so the quest got better and better... and some even said it was Fun. Before one knew Draft ate Doubt and No-Result was trodden over by Paper and his friends. The peak loomed close. TEXi swept away Problem that plagued MacDino. The way was clear. As the end drew near, there were no more monsters to fear. MacDino was hot, CPU spinned. The three Wise Men waited. The Little Person quivered with excitement (and a lot of Coffee)...

Michael, without you this thesis would never have existed.

Daryel, for upkeep, maintenance and your unending belief in me.

Katie, without you I would not be the crazy, happy, 'blingy' person that I am.

Srinivas, for friendship, food and tech support.

Sharkie and Fishy, without you there would be no Fun.

The Proof Readers, Michael, Daryel, Jeremy, Clem and Malcolm for vanquishing Errors.

The Financers, The ANU and NICTA.

MacDino, muchas gracias for not blowing up!

EPILOGUE: And everyone existed normally ever after...

Abstract

We present a formalisation of the theory of context-free languages using the HOL4 theorem prover. The formalisation of this theory is not only interesting in its own right, but also gives insight into the kind of manipulations required to port a pen-and-paper proof to a theorem prover. The mechanisation proves to be an ideal case study of how intuitive textbook proofs can blow up in size and complexity, and how details from the textbook can change during formalisation.

The mechanised theory provides the groundwork for our subsequent results about SLR parser generation. The theorems, even though well-established in the field, are interesting for the way they have to be "reproven" in a theorem prover. Proofs must be recast to be concrete enough for the prover: patching deductive gaps which are relatively easily grasped in a text proof, but beyond the automatic capabilities of contemporary tools. The library of proofs, techniques and notations developed here provides a basis from which further work on verified language theory can proceed at a quickened pace.

We have mechanised classical results involving context-free grammars and pushdown automata. These include but are not limited to the equivalence between those two formalisms, the normalisation of CFGs, and the pumping lemma for proving a language is *not* context-free. As an application of this theory, we describe the verification of SLR parsing. Among the various properties proven about the parser we show, in particular, *soundness*: if the parser results in a parse tree on a given input, then the parse tree is valid with respect to the grammar, and the leaves of the parse tree match the input; and *completeness*: if the input belongs in the language of the grammar then the parser constructs the correct parse tree for the input with respect to the grammar. In addition, we develop a version of the algorithm that is executable by automatic translation from HOL to SML. This alternative version of the algorithm requires some interesting termination proofs.

We conclude with a discussion of the issues we faced while mechanising pen-and-paper proofs. Carefully written formal proofs are regarded as rigorous for the audience they target. But when such proofs are implemented in a theorem prover, the level of detail required increases dramatically. We provide a discussion and a broad categorisation of the causes that give rise to this.

Contents

List of Figures

List of Tables

Publications

Parts of this thesis have been published as the papers listed below, Chapter 2, Chapter 3 and Chapter 5, respectively.

- ◇ Aditi Barthwal and Michael Norrish. A Formalisation of the Normal Forms of Context-Free Grammars in HOL4. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23 to 27, 2010. Proceedings, volume 6247 of Lecture Notes in Computer Science*, pages 95–109. Springer, August 2010.
- ◇ Aditi Barthwal and Michael Norrish. Mechanisation of PDA and Grammar Equivalence for Context-Free Languages. In Anuj Dawar and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010, volume 6188 of Lecture Notes in Computer Science*, pages 125–135, 2010.
- ◇ 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.

Chapter 1

Introduction

'To Start Press Any Key'. Where's the ANY key?

Homer Simpson

Contents

1.1 Automating reasoning

The pinnacle of the technological revolution that was heralded by programmable machines is long considered to be the ability to replicate human thinking. The idea of intelligent machines caught our imagination and inspired a flood of science fiction. But in reality, the effort to reach such a level is still ongoing. The pursuit of this idea gave birth to the field of Artificial Intelligence.

Mathematical reasoning is a particular form of human thinking. It is characterised by its attempt to achieve extreme precision wherein all steps in the process are a series of logical deductions based on a set of axioms. This makes it potentially well-suited to automation. As early as the 1960s (Gabbay et al. [23]), computer programs were being developed to perform mathematical reasoning. Such programs went beyond numerical calculations and juggling of data. They made logical deductions, linked concepts and joined results to generate valid statements.

The field of automated reasoning grew out of these early efforts. It seeks to automate the inference-based process as practised in mathematics and formal logic. As we will see later on, the mechanisation of logical abstraction has its own set of problems.

Automated reasoning systems We are interested in a particular sub-field of automated reasoning: Interactive Theorem Proving (ITP). Systems that are used to prove mathematical theorems are referred to as automated reasoning systems systems. Such mathematical/logical software allows the discovery of a proof and the checking a given proof. This can be done using computation, automatic proofs or interactive proofs. For computation, one provides definitions and questions to get a solution. Some examples of this are Mathematica and Maple. For automatic proofs, definitions are supplemented by axioms and statements, and the prover certifies the statements based on the given axioms. Example of such provers are Otter and Vampire. Such provers are called *automatic* provers. They search a large space in order to make logical deductions. For interactive proofs, the prover is called a *proof assistant* or an *interactive prover*. The proof assistant requires an element of interaction with the user before it certifies a proof as correct or otherwise. Proof assistants tend to be less specialised than automatic provers since they can harness the knowledge of the user and thereby span a larger body of applications. Note that most of the actual theorem provers are not pure in terms of their classification in the above mentioned categories.

The central theme of automated reasoning systems is *Proof checking*. This is the validation of proofs using either automatic or interactive provers. The difference between the two is the extent of interaction required between the automated prover and the human developer. One scenario is where the prover takes a set of criteria from the user and comes up with a solution on its own as is the case in model checking. The second scenario is where the proof system acts as an assistant to the human user. In this case the deductive reasoning for proving a goal can be done by a human directing the overall proof development by using a theorem prover. Under this interactive strategy the human is responsible for formalising the intuitive ideas while the prover provides a checking mechanism for developing a sound proof (Krauss [41]). A plethora of such interactive theorem provers exist to act as artificial mathematical assistants. They use numerous theorem proving techniques such as natural deduction, term rewriting and mathematical induction (Wiedijk [77]). These techniques are based on using various logic formalisms. Some of the commonly used interactive theorem provers are HOL, Isabelle, Coq, Mizar. In 'The Seventeen Provers of the World' [77], Wiedijk compares the styles of different proof assistants for formalising mathematics. This is done based on the mechanisation of Pythagoras' proof of the irrationality of $\sqrt{2}$. Various other comparisons have been done between other provers such as PVS and Isabelle/HOL (Griffioen and Huisman [29]), Coq and HOL (Zammit [82]) and Mizar and Isar (Wenzel and Wiedijk [75]).

What makes a good theorem prover? On a very broad level two characteristics may sum up what is offered by a theorem prover. The first one is the basic implementation. A theorem prover should be *sound*, *i.e.* it should not allow invalid theorems to be proved. It should also provide an adequate type system to allow the proofs one needs to verify. The expressiveness of the type system can have an impact on how easily the user can state what needs to be proved. Note that ACL2 ([1]) which is widely used for software and hardware verification does not have a static type system. A strong type system is not necessarily a requirement for a good theorem prover. What is necessary for a system is the flexibility or ability to deal with different types of values such as integers, floats, etc, be it in the form of static typing or otherwise.

The second characteristic is the usability of the theorem prover. This consists of the underlying capabilities of the system and the user-end facilities. The underlying capabilities refer to the library of theories and the extent to which the prover allows importing and exporting theories from other systems. The user-end facilities refer to the user-interface. This is particularly relevant for interactive theorem provers. If the prover is going to be a helpful assistant then it should make the task of translating proofs easier. This depends on the language of the prover, how easy it is to learn and how close it is to the mathematical language. For example, the Isabelle proof system provides an interface that is closer to how proofs are expressed in natural language compared to a system such as HOL. What a theorem prover designer considers to be a good system is not always the same as the users of the system (Kadoda [39]). Aitken *et al.* [2] provided an empirical study of user interaction with the HOL theorem prover using intermediate users and advanced users of HOL. According to Kadoda *et al.* a poor quality user interface was found to be one of the main obstacles to theorem provers being used more widely.

Along with an expressive language it should also be relatively easy to guide the prover towards the goal and in the case where one's argument is not accepted, the point of failure should be easy to locate.

What are theorem provers good for? Automated reasoning has been used to solve a multitude of problems in formal logic, mathematics, and software and hardware verification, especially for digital artifacts such as chips. Logical formalism is well suited for representing various models and through this automated reasoning is increasingly being harnessed to solve problems in fields ranging from natural language processing to bio-informatics. In cryptography, the application of formal methods has moved towards application of the formal theory to realistic protocols for increased security (Meadows [46]). Since its inception, the number and the complexity of problems that a theorem prover can handle has seen a significant increase. Complex hardware problems, software and requirements modeling, mathematical analysis are just a handful of the many fields that have adopted formalisation [79].

There are three overlapping applications for which proof systems are used. First, proof of safety critical or expensive systems; with both designs and software alike one needs to be able to predict behaviour of computer systems on which human life and security depend. The systems to be verified can themselves come with a formal proof which can then be verified by a theorem prover or a *certifying compiler* consisting of a compiler and a certifier. Such a software mechanism is referred to as *Proof-Carrying Code* whereby the host system can certify a program supplied by an untrusted source (Necula [50]). It was originally described by Necula and Lee in 1996.

Second, certification of pen and paper proofs. Mistakes in proofs are not rare, especially as the size and complexity of the proof increases. Despite immense efforts, occasionally mathematicians or logicians find mistakes in published proofs. What would help is a more objective process of validating a proof. In MacKenzie's [43] words, "Mechanized proofs, they reasoned, would be easier, more dependable, and less subject to human wishful thinking than proofs conducted using the traditional mathematicians tools of pencil and paper".

More recently the third area where theorem provers can be a handy assistant has emerged: teaching. Blanc *et al.* have used the Coqweb environment to explore how proof assistants can help teachers to explain the concept of a proof and how to search for one [9]. Coqweb [17] provides a language that is close to standard mathematical language. It also provides an interface for solving exercises using Coq. Proofs are essentially performed by clicking. Such web-based approaches that rope in automated proof techniques are being used to teach subjects like discrete mathematics and logic at the undergraduate level. Coq has also been used by Benjamin Pierce to teach courses at University of Pennsylvania [60]. Going beyond teaching theorem proving as a course in its own right, he has used it as a framework to teach programming language foundations [59].

So how good are the theorem provers with reasoning? Mechanising a proof involves a lot of time and effort, both for training in the use of the system (a one-off cost) and setting up the system to 'know' about the required theory. Before attacking the proofs, one has to set up the background theory inside the prover. The next step is proving the theorem, which involves informing the prover about how to establish the validity of the theorem. This step generally requires establishing numerous lemmas and sub-theorems before the final theorem can be proven. A prover is almost never able to prove a theorem independently. Working with a prover amounts to basically "reproving" a theorem since the extent of the details required to convince such a tool of an argument is substantial. The prover usually requires a lot of human guidance and ingenuity to validate the final goal. A non-trivial theorem can take several years. For example, Thomas Hales is anticipating twenty years to certify his proof of the Kepler Conjecture [77], first presented in 1998. The 'Flyspeck' project 'Formal proof of kepler' is to convince the referees of the Annals of Mathematics of the correctness of his proof.

In addition, formalisation of a theory does not necessarily entail that the said theory is correct (with respect to its specification). When results are proven in the context of a particular formalisation, they are proven based on the definitions in question. Errors in encoding large and complicated definitions are easy to make. For example, the assumptions for a theorem may actually imply falsity, thereby making it possible to prove anything. Thus, checking whether the underlying definitions are consistent is an essential task.

Similarly, the soundness of the prover is crucial to developing correct formalisms. This is easier to deal with since over time unsoundness will become obvious when working with the theorem prover. One can also take a more formal approach for establishing soundness as outlined by Harrison in [33] wherein he explains self-verification of the kernel of HOL Light prover.

To sum it up, formalisation is hard, time consuming and like pen-and-paper proofs it is not foolproof. In light of the above issues, the key question is *why bother with formalisation?*

First, formalisation is a very interesting experience in its own right. Some fields definitely are inherently more suited to automation than others. In spite of the intensive nature of the mechanisation process, large bodies of work including a lot of complex theories have been formalised in various proof systems. Some of these include basic mathematics by Harrison [31], number theory by Avigad [3], basic category theory by O'Keefe [54], Newton's Principia Mathematics by Fleuriot *et al.* [21], prime number theorem formalised by Avigad [4] and [20] and the Church-Rosser theorem by Shankar [68]. The rapid development in proof checking has resulted in automated proofs of these theorems which have been for a long time beyond the reach of such technology.

The formalisation process can lead to a better understanding of the proof structure and the reasoning and can contribute to 'classical' mathematics by revealing unknown or overlooked aspects of the proof construction as was the case with John Harrison's thesis [32]. In his thesis, Harrison provides a detailed comparison of different reals constructions, which at that time had not been attempted. His analysis of the transcendental functions was also a novel development. Because formal proofs are so difficult, more care must be taken in stating and proving a goal so that the proof is detailed and complete. One also tends to get into the habit of identifying commonalities which can be reused elsewhere [31]. The extra confidence gained from constant feedback about the correctness of the intermediate steps during the proof development is highly valuable.

Second, some problems are just too computationally intensive to be feasible without the aid of a crunching machine, be it numbers or possibilities. Robbin's problem, first posed by Herbert Robbins in the 1930s, was proved by EQP in 1996, a theorem prover for equational logic [44]. Computer proofs have aided progress on problems that otherwise might have remained unsolved such as the proof of XCB, the last shortest single axiom for the equivalent calculus, by Wos *et al.* [80]. They used the automated reasoning tool Otter for the proof of XCB.

Last but not the least, a theorem prover can act as a proof certifier by validating techniques and results. Proof checking by hand, especially of a non-trivial theorem, is a long and arduous process. The subjectivity involved also makes it prone to flaws as is constantly seen from proofs being refuted even after they have been published. Once proven using a sound proof checker, the assurance of the proof's validity is nearly complete. Nearly, because the premises of the proof can still be incorrect. But now we have reduced the problem of checking the whole proof to a much smaller problem of just making sure that the definitions (and premises) are correct. Wim H. Hesselink, one of the many who crossed the bridge from mathematics to automated mathematics, sums up the proof development using theorem provers quite nicely: that in a way computer verification of mathematical theories is just a "strengthening of the social process of acceptance of the result with its accompanying proof by the mathematical community" [34].

1.2 Mechanisation – the story so far

Formal verification has expanded its ambitions to include mechanising individual proofs of complex theorems as well as massive projects aiming to cover all of mathematics. Computer verification of (mathematical) proofs started in the Netherlands in 1967 with the Automath project of N.G. de Bruijn (Nederpelt [51]). The project aimed at designing a language for expressing mathematics in a way that would allow a computer to verify its correctness. The system developed was tested by treating a full text book, Landau's book 'Grundlagen der Analyse'. The verification was done by L.S. van Benthem as part of his doctoral thesis [74].

In 1973, the Mizar project started with the aim of formalizing and verifying all mathematics (Rudnicki [67]). The Mizar project has formalised a large part of undergraduate mathematics. Along similar lines, the idea of the QED project was put forth in 1994. As stated in the *QED manifesto*, the goal is to build a system that represents mathematical knowledge and techniques [64] by using mechanical methods to check proofs for correctness. Unfortunately, the project has not taken off since its inception. Wiedijk, in [78], discusses the reasons why the QED project has stalled in light of existing proof systems.

The most recent of these attempts is the 'vdash' project started by Cameron Freer [62]. vdash is a wiki of formally verified mathematics. According to Freer, it is one approach towards a *maths commons*, a site with all mathematics in one place in a common language which can be edited by anyone. The main components are a computer proof assistant, a library of proofs and a web interface. The idea is to have something akin to Wikipedia where all submissions are verified using a theorem prover before they are accepted. The initial version is to consist of a distributed revision control system such as Monotone or Git, a wiki interface initially seeded by content from IsarMathLib with a simple script by which the proof assistant Isabelle approves new contributions. The core spirit of these projects, aimed at developing software to support development of mechanised proofs by having a proof database, is close to our own philosophy behind the mechanisation of background theories that provide a platform for bigger developments.

While projects such as the ones mentioned above were aiming to formalise mathematics gradually from the basics up, some individuals undertook the mechanisation of complicated individual theorems. Theorems such as Kantorovitch's theorem, which gives sufficient conditions for convergence of Newton's method, has been verified in the Coq proof assistant by Paşca [55]. With their applicability in a wide range of fields, theorem provers have matured considerably. Théry [73] has formalised Huffman's algorithm using the Coq proof assistant. Huffman's algorithm is a procedure for constructing a binary tree with minimum weighted path length. His proof sketch does not follow the usual text treatment. On the other hand, Blanchette's [11] formalisation of the same (in Isabelle/HOL) follows the treatment in standard algorithm textbooks.

Various attempts have been made to formalise the theory of formal languages in a proof assistant. In the field of language theory, Nipkow [52] provided a verified, executable lexical analyzer generator. Courant and Filliâtre have formalised some of the theory of regular and context-free languages in Coq [18]. The formalisation includes results such as that all context-free languages can be recognised by some pushdown automata and the union of two context-free languages is also context-free. Rival *et al.* [65] formalised tree automata in Coq.

Closer to our own work, Minamide [48] has verified three decision procedures on context-free grammars in the context of analyzing XML documents. These procedures include inclusion between a context-free language and a regular language, well-formedness and validity of the XML documents. The formalisation is also used to generate executable code which is incorporated into an existing program analyser. They do not tackle any proofs for normalisation of CFGs.

Our work on the theory of CFGs naturally led us to tackle an application of such a mechanisation, a verified SLR parser generator. CFGs form the basis of compiler theory. The parsing phase of compiler generation heavily relies on the properties of context-free grammars.

In the area of verifying a compiler, most of the verification work starts past the parsing stage. The compiler phases can be roughly divided into front-end (lexing, parsing, etc. leading up to the creation of the abstract syntax tree) and the back-end (control and data flow analysis, register allocation, code emission, assembler, linker).

Leroy has done substantial work on verifying the C compiler but the verified stages do not including either lexing or parsing. In *Formal Verification of a C Compiler Front-end*, Leroy *et al.* [12] discusses the translation of a subset of the C language into Cminor intermediate language. Carrying on from that, the *Formal Certification of a Compiler Back-end* [42] by Leroy discusses the formal certification of a compiler from Cminor to PowerPC assembly code using Coq. In another report, *Compiler Verification for C0* [71], Strecker presents the correctness proof of a substantial fragment of C0 to DLX compiler. Here, C0 is a type-safe fragment of C. His work was carried out in Isabelle [56].

1.3 Where do we fit in?

Most of the formalisation in the area of language theory and compiler construction has been application oriented. This has left a niche to fill in the area of language theory: exploring the formalisation of widely known standard text proofs. This thesis makes a contribution towards theorem provers being used as a validation tool for textual proofs. This is not to say that the applications are left untouched. The verification of applications itself requires a theoretical framework to exist. In this thesis we provide such a framework for context-free grammars and explore the key issues of such a development. As an application, we present a verified SLR parser generator for CFGs. We have used the presentation in Hopcroft and Ullman [36], a standard textbook. This text is a starting point for any language theory course. The proofs are well known and well understood. Thus, it provides an ideal platform for exploring the various difficulties of translating widely known and at times simple proofs in a theorem prover. Hoproft and Ullman is a very old textbook. It first appeared in 1979. Almost all later texts are recapitulations especially with respect to the theory on which this thesis is based. In some cases, for example in the second edition, the detailed proofs of Chomsky and Greibach normal forms were omitted. They are only cursorily mentioned in the second edition as compared to the first one where the treatment is a bit more in-depth. Thus, using a more recent edition would not have had any impact on our work. Obviously, other undergraduate texts cover similar material but we found the treatment in Hopcroft and Ullman to be the most suitable for our purpose.

Throughout this thesis we use formalisation and mechanisation interchangeably to refer to the work that has been verified (and possibly implemented as well) using a proof assistant or a theorem prover.

1.3.1 The target domain – CFGs

Context-free grammars provide a nice way of expressing the recursive syntax common to programming languages. The earliest use of the use of recursive syntax dates back to somewhere between 4^{th} to 6^{th} century BC to Pãṇini (Ingerman [37]). He described how sentences in Classical Sanskrit can be built up from smaller clauses recursively. CFGs in their basic form have been found to be too restrictive to model natural languages apart from a few specific forms such as Venpa which is a form of Classical Tamil poetry. But they turned out perfect for use in the area of programming languages (Rozenberg and Salomaa [66]), which are a very constrained form of natural language. The use of block structure for describing programming languages started with Algol whose syntax is partially described using context-free grammar. CFGs as we know them now are attributed to Noam Chomsky. The three models for language description were published by Chomsky in 1956 [14]. Now CFGs are a standard feature of programming language descriptions.

The practical applications of CFGs are well known in the area of language representation, formalising the notion of parsing and in string-processing applications. They have had a huge impact on both defining such languages and the construction of compilers for them. With added extensions, CFGs have been able to move beyond this small niche. For example, stochastic context-free grammars (SCFGs) allow addition of a probability factor to each of the productions in the grammar. Thus, some derivations are more likely for a particular grammar. SCFGs are used in the area of natural language parsing to model the frequency of how the different linguistic blocks combine together in a sentence. They have also been used to predict the secondary structure of RNA (*R*ibo*n*ucleic *a*cid). Another form of CFGs is the multiple context-free grammar (MCFG) which is a specialisation of generalised context-free grammars (GCFGs) introduced by

Pollard in 1984 [61]. MCFGs were introduced in 1987 (T. Kasami, H. Seki and M. Fujii [72]). Using MCFGs it is possible to handle discontinuous constituents in language structures such as "respectively". Pullum *et al.* in [63] have analysed the correspondence between natural languages and context-free languages.

Phanindra, *et al.* [27] have presented a fast multiple pattern matching algorithm using CFGs to represent the queries for the strings to be searched. The resulting output are strings that match both the text data and the given context-free grammar. The CFG is first transformed into Greibach Normal Form before being used for pattern matching the strings. With its capacity to retrieve multiple patterns at the same time, the algorithm can be used in varied fields such as bio-informatics and information retrieval. The simple and clear specification of CFGs also comes in handy to reduce the complexities in modeling requirements [27], for example during the software development life cycle.

Close to our own work is the use of CFGs in the Isabelle theorem prover. Isabelle implements different logics such as HOL, LCF, ZF, Modal, etc. The syntax of each logic is presented using a CFG (Paulson [57]).

1.3.2 The assistant – HOL4

This work was done in the HOL4 theorem prover. HOL stands for *H*iger *O*rder *L*ogic. It is the name of the logic as well as the name of the theorem proving system. We have used the abbreviation to refer to the system. Gordon provides a nice introduction to the HOL theorem proving environment in [24]. The most recent version of HOL, *i.e.* HOL4 has been described by Slind and Norrish in [70]. Throughout the thesis, the use of HOL stands for this particular version. HOL is an LCF-style proof assistant. It is also a family of proof assistants with members including Isabelle/HOL and HOL Light. The members share the same logic and architecture to some extent (Gordon [26]). Again, LCF refers to a theorem prover and also stands for *L*ogic for *C*omputable *F*unctions. Logic for computable functions is Milner's name for a first order logic of domain theory devised by Dana Scott. The terms in LCF are based on typed λ-calculus, and formulae are based on predicate calculus. The higher order logic *à la Church* employed by HOL allows one to quantify over predicates. New concepts are introduced by definitions, the so called *definitional style*. HOL has a small trusted kernel (on which the soundness depends). The deductive system of the HOL logic has eight rules of inference: assumption introduction, reflexivity, beta-conversion, substitution, abstraction, type instantiation, discharging an assumption and modus ponens. In addition there are also five axioms which are defined using logical constants. An important consequence of this is that it preserves the soundness of the whole environment. New theorems are accepted only if they are accompanied by a fixed set of inference rules. Users are free to add onto the system without compromising it. An exception to this rule is allowing external proof tools to provide HOL theorems *without a proof*. In such cases, the theorems are tagged, which allows one to ascertain whether a proof is based on a HOL theorem or an external oracle (Gordon and Melham [24], Slind and Norrish [70]). HOL also provides various automated reasoners to help in proof search. External SAT tools can be used to verify propositional logic formulas. The proofs of such formulas are then translated back into HOL proofs. Similar procedures exist for arithmetic formulas. The most commonly used tool is the simplifier. It provides conditional and contextual ordered rewriting and can be extended with context-aware decision procedures.

HOL notation The presentation of proofs in HOL uses various shorthand notations. The inbuilt boolean operators in HOL are listed in Table 1.1.

$\neg$	Not
$\wedge$	And
$\vee$	Or
$\forall$	For all
$\exists$	There exists
$\Rightarrow$	Implies
$\Longleftrightarrow$	If and only if

Table 1.1 – Boolean operators in HOL

Table 1.2 presents list, set and relation operators that have been commonly used throughout our work.

$++$	List append
$::$	List cons
$\in$	List/set membership
$\notin$	List/set non-membership
*	Reflexive transitive closure
λ	Lambda abstraction

Table 1.2 – Other commonly used operators in HOL

We have introduced several other operators to make the proof text more readable in HOL and as a short hand for our own definitions. The new operators resulting from our formalisation process are introduced when the corresponding definitions are presented. The theorems presented in this thesis are directly pulled from the HOL sources compared to being typeset manually in LaTeX. As a result the premises of some theorems use conjunctions while those of others use implications. For example, $A \land B => C$ versus $A => B => C$. This difference in expression of what are logically the same statement arises when using implications proves an easier form to prove in HOL. This is particularly relevant when applying existing induction principles.

We use the term *text* proof or *textual* proof to refer to a pen-and-paper proof. The text proofs are the result of multiple attempts at both establishing a proof and making it structurally clear and simple. The latter is usually done by abstracting away the details, *i.e.* not spelling out every little step. If one were to enumerate all the inferences, the proof would be too complicated and tedious to comprehend. As Julie Rehmeyer writes: “When Bertrand Russell and Alfred North Whitehead tried to do so for just the most elementary parts of mathematics, they produced a 2,500-page tome” [38]. The result was so difficult to understand that Russell admitted to a friend, “I imagine no human being will ever read through it”.

On the contrary, the proofs done in a theorem prover have to give up on the desire for elegance (so dear to mathematicians) and explicitly tackle all the numerous details. This is the primary reason that the HOL proofs are so complicated and lengthy. To assist in understanding the overall formalisation process, when presenting the proofs we gloss over many of the details that make the proof work in a theorem prover. Even then the proofs require a significant effort to follow.

1.4 Extending the story

The main contribution of this thesis is the infrastructure: definitions and proofs relating to context-free grammars and pushdown automata. These well-established fields are traditionally ignored as a possible subject for mechanisation. In some sense the proofs are seen as *too* well-understood to justify the large effort formalisation requires. On the contrary, the formalisation of these theories is vital because they form the basis for further works such as compiler proofs. The importance of CFGs in their many forms goes beyond their use in compiler theory. They are increasingly being used in problems such as pattern matching and natural language processing.

The other area our work addresses is the goal of proof certification. In order for theorem provers to be used more extensively for guaranteeing proof correctness in various fields, we need to provide an extensive background of theories so that further automation efforts can start at a higher level of abstraction.

As previously mentioned, Hopcroft and Ullman have been our primary text for formalisation. We have provided corresponding references within Hopcroft and Ullman (*H&U*) where applicable. From here on the thesis is organised into four themes: theory, application, concerns and conclusions. These cover mechanisation of the theory of CFGs and PDA, an application of this theory in the form of a parser generator, the various concerns that make the mechanisation both a long and a complex process and finally we conclude and present areas for improvement and extensions.

Theory Chapter 2 is based on Chapter 4 in *H&U* and covers the mechanisation of context-free grammars. In Chapter 3 we cover pushdown automata, the acceptors of CFGs. This corresponds to Chapter 5 in *H&U*. The properties of context-free languages are presented in Chapter 4. This is Chapter 6 in *H&U*. These include the pumping lemma and closure properties. The concrete contribution for each chapter are listed below.

Context-free grammars (Chapter 2)

◇ a formalisation of the theory of context-free grammars (Section 2.1) including the mechanised proofs for termination and correctness for

- simplification of CFGs (Sections 2.2 to 2.4);
- Chomsky Normal Form (Section 2.5) and
- Greibach Normal Form (Section 2.6).

Pushdown automata (Chapter 3)

◇ a formalisation of the theory of pushdown automata (Section 3.1) including the mechanised proofs for termination and correctness for

- language equivalence between the two criterion whereby a PDA recognises input (final state acceptance and empty stack) (Section 3.2) and
- language equivalence between CFGs and PDA (Section 3.3).

Properties of context-free languages (Chapter 4)

◇ Closure properties for Kleene star, substitution (subsuming homomorphism), inverse homomorphism, concatenation, union and intersection (Section 4.3)

◇ Pumping lemma (Section 6.3.1)

Application As an application of the theory mechanised above, we provide an implementation of a verified SLR parser generator in Chapter 5.

Concerns We present the common underlying issues that recur throughout the mechanisation process. In Chapter 6 we present a broad categorisation of the problems we encountered during the mechanisation. We present a discussion of the ways in which well-known, "classic" proofs can require considerable "reworking" when fully mechanised. This reworking can result in considerable divergence from the original proofs in terms of both size and complexity. This discussion forms the central theme of the thesis. The divergence from text proofs is pointed out throughout the discussion of the mechanisation process and is discussed in depth in Chapter 6. These issues emphasise the need for extensive human input required during formalisation.

Conclusions As with most endeavours, there is always room for further work. We conclude by summarising our work and providing possible extensions based on our mechanisation. Our main impetus with this work was to investigate the difficulties of formalisation and verification. Thus executability has mostly been ignored and the executable parser generator is not optimised for runtime. This is presented in Chapter 7.

The HOL formalisations are introduced as we go along. The notations are presented at the outset in each of the chapters. But for a person familiar with the underlying theory, (if required) a quick reference to the notation should suffice to be able to read each chapter on its own. We use HOL's inbuilt typesetting for all the theorems, definitions and terms. Font Sans-serif is used to typeset ML level terms. HOL theorems and definitions are presented and numbered separately from their text counterparts. Where a definition is duplicated for easy access, we omit the numbering. At the end of relevant chapters, we have provided a summary of the lines of code (LOC), the number of definitions and the number of proofs. The lines of code include blank lines as well as comments. Some of our work has already published. In such cases, the summary numbers may be different from the published ones as the formalised code has been refactored since then.

To a person who is not from a theorem proving background, it may seem that the ratio of number of supporting lemmas to proofs is very high. This is typical of the automation process. The interesting bits are the theorems. The proofs themselves consist of HOL code which is hard to understand and usually obfuscates the bigger picture. At a high-level, the proof of a theorem can be viewed as a composition of all the sub-theorems that are tied together to achieve the final result.

The majority of the background mechanisation has been been presented in Chapter 2, the basis of both the theories and mechanisation in the latter chapters. Chapter 6, on automation concerns, draws on material presented in earlier chapters but by no means is dependent on knowledge of the earlier material. All the assumptions and assertions in this thesis have been mechanised and the HOL4 sources for the work are available online at [35].

Chapter 2

Context-Free Grammars

Theorem proving and
sanity; Oh, my! What a
delicate balance

Victor Carreno

Contents

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 “recursive structure” of sentences in a natural way.

The Chomsky and Greibach normal form results were first presented in 1965. The first was presented by Noam Chomsky [15] and the second by Sheila Greibach [28]. There is also another classical normal form called the Operator Normal Form (Miller [47]). In

Operator Normal Form, the right-hand side (RHS) of a rule cannot have two consecutive nonterminals.

Most of the applications of CFGs are based on the grammar being simplified in some manner. These are usually restrictions on the format of the productions in the grammar without affecting the language of the grammar.

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.

In this chapter we discuss mechanisation of Chomsky Normal Form and Greibach Normal Form. In Section 2.1, we provide the background mechanisation for CFGs and introduce some of the notation that will be used in the remainder of the thesis. Parts of this chapter have been published in [7].

Contributions

- ◇ The first mechanised proofs of termination and correctness for a methods that simplify a CFG without changing the language. These methods deal with removing useless symbols, removing ϵ-productions and unit productions (Sections 2.2 to 2.4).
- ◇ The first mechanised proofs of termination and correctness for a method that converts a CFG to Chomsky Normal Form (Section 2.5).
- ◇ The first mechanised proofs of termination and correctness for a method that converts a CFG to Greibach Normal Form (Section 2.6).

Interestingly, though the proofs we mechanise here both expand dramatically from their length in Hopcroft and Ullman [36], the GNF proof expands a great deal more than the CNF proof.

We briefly present the results of the less interesting transformations corresponding to our first set of contributions. Except for the tediousness of rendering them in HOL and some interesting “finite-ness” proofs (covered in Chapter 6), the algorithm for such transformations in itself was straightforward to implement. Therefore we devote much of the discussion to CNF and GNF transformations.

2.1 The theory of CFGs

A CFG is denoted as $G = (V, T, P, S)$, where V is a finite set of variables or nonterminals, T is a finite set of terminals, P is a finite set of rules or productions of the form $A \rightarrow \alpha$ where A is a variable and $\alpha \in (V \cup T)^*$ and S is a special variable called the start symbol.

We say a string $\alpha A \gamma$ *derives* $\alpha\beta\gamma$ if there is production $A \rightarrow \beta$ in G. The one step derive relation for grammar G is written as $\Rightarrow_G$. The subscript G may be omitted if it is clear which grammar is involved. The derivation of string β starting from string α in zero or more steps is written as $\alpha \Rightarrow^* \beta$. Similarly, the derivation of string β starting from string α in zero or more i steps is written as $\alpha \Rightarrow^i \beta$.

A string of terminals and variables α is called a *sentential form* if $S \Rightarrow^* \alpha$. If α contains only terminal symbols then it is referred to as a *word*. The *language* of a grammar is all the words that can be derived from the start symbol in zero or more steps. We refer to a string of terminals and variables as a *sentence* irrespective of where the derivation begins. In the context of strings being recognised by a grammar, an *input* is a string of terminals.

The implementation of the above infrastructure turns out be a swift and straightforward task in HOL.

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 type `'nts` is used for nonterminals and the type `'ts` is used for terminal symbols. The instantiation of the basic type variables for terminals and nonterminals may be the same. The constructors `NTS` and `TS` are used to turn types `'nts` and `'ts` into symbols thus allowing the distinction between a nonterminal symbol and a terminal symbol.

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`. 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. It should be noted that there is no predicate of the form "is_grammar" singling out the grammars from some larger set. Instead, any term having the `('nts, 'ts) grammar = G of ('nts, 'ts) rule list => 'nts` represents a grammar.

HOL Definition 2.1.1 (`rules`)

$$\texttt{rules (G } p\ s\texttt{) = } p$$

Traditional presentations of grammars often include separate sets corresponding to the grammar's terminals and nonterminals. 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 nonterminals or variables, 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 $A \to \beta$ is one of the rules in the grammar. In HOL:

HOL Definition 2.1.2 (`derives`)

$$\begin{array}{l}
\texttt{derives}\ g\ \mathit{lsl}\ \mathit{rsl} \iff \\
\quad \exists\, s_1\ s_2\ \mathit{rhs}\ \mathit{lhs}. \\
\qquad s_1 \texttt{ ++ [NTS } \mathit{lhs}\texttt{] ++ } s_2 = \mathit{lsl} \land s_1 \texttt{ ++ } \mathit{rhs} \texttt{ ++ } s_2 = \mathit{rsl} \land \\
\qquad \texttt{rule}\ \mathit{lhs}\ \mathit{rhs} \in \texttt{rules}\ g
\end{array}$$

(The infix ++ denotes list concatenation. The ϵ denotes membership.)

We write $\texttt{(derives } g\texttt{)}^*\ \mathit{sf}_1\ \mathit{sf}_2$ to indicate that sf_2 is derived from sf_1 in zero or more steps, also written $\mathit{sf}_1 \Rightarrow^* \mathit{sf}_2$ (where the grammar g is assumed). This is concretely represented using what we call derivation lists. If an arbitrary binary relation R holds on adjacent elements of ℓ which has x as its first element and y as its last element, then this is written $R \vdash \ell \lhd x \to y$. In the context of grammars, R relates sentential forms. Later we will use the same notation to relate derivations in a PDA. Using the concrete notation has simplified automating the proofs of many theorems.

We will also use the leftmost derivation relation (`lderives`) and the rightmost derivation relation (`rderives`) and their closures respectively. A leftmost derivation is obtained when at each step of the derivation it is the leftmost nonterminal that is expanded.

HOL Definition 2.1.3 (`lderives`)

$$\begin{array}{l}
\texttt{lderives}\ g\ \mathit{lsl}\ \mathit{rsl} \iff \\
\quad \exists\, s_1\ s_2\ \mathit{rhs}\ \mathit{lhs}. \\
\qquad s_1 \texttt{ ++ [NTS } \mathit{lhs}\texttt{] ++ } s_2 = \mathit{lsl} \land \texttt{isWord}\ s_1 \land \\
\qquad s_1 \texttt{ ++ } \mathit{rhs} \texttt{ ++ } s_2 = \mathit{rsl} \land \texttt{rule}\ \mathit{lhs}\ \mathit{rhs} \in \texttt{rules}\ g
\end{array}$$

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

If the derivation is obtained by expanding the rightmost nonterminal at each stage of the derivation, then the derivation is referred to as a rightmost derivation.

HOL Definition 2.1.4 (`rderives`)

$$
\begin{array}{l}
\texttt{rderives}\ g\ \mathit{lsl}\ \mathit{rsl} \iff \\
\quad \exists\, s_1\ s_2\ \mathit{rhs}\ \mathit{lhs}. \\
\qquad s_1\ \texttt{++}\ [\texttt{NTS}\ \mathit{lhs}]\ \texttt{++}\ s_2 = \mathit{lsl} \wedge \texttt{isWord}\ s_2 \wedge \\
\qquad s_1\ \texttt{++}\ \mathit{rhs}\ \texttt{++}\ s_2 = \mathit{rsl} \wedge \texttt{rule}\ \mathit{lhs}\ \mathit{rhs} \in \texttt{rules}\ g
\end{array}
$$

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 2.1.5 (L)

$$
\begin{array}{l}
L\ g = \\
\quad \{\mathit{tsl} \mid (\texttt{derives}\ g)^*\ [\texttt{NTS}\ (\texttt{startSym}\ g)]\ \mathit{tsl} \wedge \texttt{isWord}\ \mathit{tsl}\}
\end{array}
$$

The choice of the derivation relation (`derives`, `lderives` or `rderives`) does not affect the language of the grammar. Thus, we have:

HOL Theorem 2.1.1

$$L\ g = \texttt{llanguage}\ g$$

Function `llanguage` g returns words derived from the start symbol using only the leftmost derivation.

HOL Theorem 2.1.2

$$L\ g = \texttt{rlanguage}\ g$$

Function `rlanguage` g returns words derived from the start symbol using only the rightmost derivation.

2.2 Elimination of useless symbols

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 $A \to B$ 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 above transformations are formalised using relations in HOL. For each of the above, we describe a relation, say R. Relation R holds over the old grammar (g) and the new grammar (g') if and only if g' is obtained by restricting g in one of the above ways. We then go on to prove that if such a relation holds over g and g' then their languages must be equivalent.

For a grammar, $G = (V, T, P, S)$, a symbol X is useful if there is a derivation $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ for some α, β, w, where w is in T^*. There are two aspects to usefulness.

The first is that some terminal string must be derivable from X. Secondly, X must occur in some string derivable from S.

Removal of non-generating nonterminals We first prove that restricting a grammar G to satisfy the first condition results in a grammar G' such that $L(G) = L(G')$.

HOL Theorem 2.2.1 (H&U Lemma 4.2)

```
usefulnts g g′ ⇒ L g = L g′
```

The predicate `usefulnts g g′` holds if and only if g' contains only those symbols from g that are used in derivation of a word belonging in the language of g. Instead of using a predicate over symbols in the grammar, we use a predicate over the rules in the grammar. This is to avoid scenarios where some symbol X may occur in sentential forms that contain a variable which does derive a terminal string. `usefulntsRules` only includes rules of the form $A \rightarrow A_1 A_2 ... A_n$ where each of the symbols from $A_1..A_n$ derive a terminal string.

HOL Definition 2.2.1 (`usefulnts`)

```
usefulntsRules g =
  {rule ℓ r |
   rule ℓ r ∈ rules g ∧ gaw g (NTS ℓ) ∧ EVERY (gaw g) r}

usefulnts g g′ ⟺
  set (rules g′) = usefulntsRules g ∧
  startSym g′ = startSym g
```

(Function `set l` returns the corresponding set for list l.)

Predicate `gaw g nt` is true if the symbol nt derives some terminal string in grammar g, *i.e.* *g*enerates *a* *w*ord.

HOL Definition 2.2.2 (gaw)

```
gaw g nt ⟺ ∃w. (derives g)* [nt] w ∧ isWord w
```

An important thing to note here is the construction of the useful rules as a set rather than list. This choice is one of several constantly occurring issues in our mechanisation. As one can see, for this definition, set comprehension provides a neat way of expressing the concept. If lists are used instead, the resulting definition will lose its clarity and conciseness. In most cases, unless executability was of concern, we have stuck to using the clearer set comprehension notation.

Removal of non-reachable symbols For a symbol to occur as part of string that can be derived from the start symbol, it must be a part of the RHS of a production such that the nonterminal that forms the left-hand side (LHS) is derivable from the start symbol. Thus removing non-reachable symbols is just a matter of removing productions where the LHS nonterminal cannot be reached from the start symbol.

For this we define a function `rgr` that gives back a grammar which only includes productions where the LHS of the production can be reached from the start symbol of the grammar.

HOL Definition 2.2.3 (rgr)

```
rgrRules g =
  {rule ℓ r |
   rule ℓ r ∈ rules g ∧
   ∃a b. (derives g)* [NTS (startSym g)] (a ++ [NTS ℓ] ++ b)}

rgr g g′ ⟺
  set (rules g′) = rgrRules g ∧ startSym g′ = startSym g
```

We then prove that removing productions that are not reachable from the start symbol of a grammar does not affect the language.

HOL Theorem 2.2.2 (H&U Lemma 4.2)

```
rgr g g′ ⇒ L g = L g′
```

We can now chain Theorem 2.2.1 and Theorem 2.2.2 to achieve Theorem 2.2.3 that *removing non-generating and non-reachable symbols from a grammar does not affect the language generated by the grammar*.

Theorem 2.2.1 (H&U Theorem 4.2) *Every nonempty CFL is generated by a CFG with no useless symbols.*

This is reflected in the following HOL theorem.

HOL Theorem 2.2.3

```
L g ≠ ∅ ⇒ ∃g′ g″. rgr g g′ ∧ usefulnts g′ g″ ∧ L g = L g″
```

2.3 Elimination of ϵ-productions

Productions of the form, $A \rightarrow \epsilon$ are called ϵ-productions. If ϵ does not belong in the language of a grammar, then such productions can be eliminated without affecting the language of the grammar.

Theorem 2.3.1 (H&U Theorem 4.3) *If $L = L(G)$ for some CFG $G = (V, T, P, S)$, then $L - \epsilon$ is $L(G')$ for a CFG G' with no useless symbols or ϵ-productions.*

To do this, we determine for each variable A whether $A \Rightarrow^* \epsilon$. If so, we call A nullable. Each production of the form $A \rightarrow X_1 ... X_n$ can be replaced by a production (munge) where the nullable $X_i s$ have been stricken off (munge1).

HOL Definition 2.3.1 (munge)

```
munge1 g [] = [[]]
munge1 g (s::sl) =
  if nullable g [s] then
    MAP (CONS s) (munge1 g sl) ++ munge1 g sl
  else
    MAP (CONS s) (munge1 g sl)

munge g p =
  {rule ℓ r′ | ∃r. rule ℓ r ∈ p ∧ r′ ∈ munge1 g r ∧ r′ ≠ []}
```

Relation negr g g' holds if and only if grammar g' is derived from g such that g' does not have any ϵ-productions.

HOL Definition 2.3.2 (negr)

```
negr g g′ ⟺
  set (rules g′) = munge g (rules g) ∧
  startSym g′ = startSym g
```

We can then prove in HOL, Theorem 2.3.1.

HOL Theorem 2.3.1

```
negr g g′ ⇒ [] ∉ L g ⇒ L g = L g′
```

2.4 Elimination of unit productions

A unit production is a production of the form $A \to B$ where both A and B are variables.

Theorem 2.4.1 (H&U Theorem 4.4) *Every CFL without $epsilon$ is defined by a grammar with no useless symbols, ϵ-productions, or unit productions.*

In HOL this translates as,

HOL Theorem 2.4.1

```
[] ∉ L g0 ∧ rgr g0 g1 ∧ negr g1 g2 ∧ upgr g2 g3 ⇒ L g0 = L g3
```

We have a series of predicates which assert that the original grammar g_0 is transformed to give g_3 such that g_3 has no useless symbols, ϵ-productions and unit productions. The predicate `rgr` g_0 g_1 ensures that g_1 does not have any useless symbols, `negr` g_1 g_2 ensures that g_2 has no ϵ-productions and `upgr` g_2 g_3 ensures that that g_3 has no unit productions.

The rules in the new grammar consists of all non-unit productions (`nonUnitProds`) from the old grammar and new rules that are constructed using `newProds`.

HOL Definition 2.4.1 (upgr)

```
upgr_rules g = nonUnitProds g ∪ newProds g (nonUnitProds g)

upgr g g′ ⇔
  set (rules g′) = upgr_rules g ∧ startSym g′ = startSym g
```

If $A \Rightarrow^* B$ (`allDeps`) and there exists some non-unit production $B \to \alpha$ then construct the new non-unit production $A \to \alpha$. Note that α is allowed to be a single terminal symbol or ϵ here. This is the function `newProds`.

HOL Definition 2.4.2 (newProds)

```
newProds g p′ =
  {rule a r |
   ∃b ru.
     rule b r ∈ p′ ∧
     (NTS a,NTS b) ∈ allDeps (G ru (startSym g)) ∧
     set ru = unitProds g}
```

Here, unit and non-unit productions are defined as,

HOL Definition 2.4.3 (unitProds)

```
unitProds g =
  {rule ℓ r | ∃nt. r = [NTS nt] ∧ rule ℓ r ∈ rules g}
```

HOL Definition 2.4.4 (nonUnitProds)

```
nonUnitProds g = set (rules g) DIFF unitProds g
```

The 'unit' dependencies or derivations, *i.e.* those of the form $A \Rightarrow^* B$, in the grammar are given by allDeps define below.

HOL Definition 2.4.5 (allDeps)

```
allDeps g =
  {(a,b) |
   (derives g)* [a] [b] ∧ a ∈ allSyms g ∧ b ∈ allSyms g}
```

(Function allSyms *returns all the symbols, terminals and nonterminals, for the input grammar g.)*

2.5 Chomsky Normal Form

In this section we present a formalisation of Chomsky Normal Form, assuming the grammar has already gone through the previously described simplifications.

Theorem 2.5.1 (H&U Theorem 4.5) *[**Chomsky Normal Form**] Any CFL without ϵ can be generated by a grammar in which all productions are of the form $A \to BC$ or $A \to a$. Here A, B, C are variables and a is a terminal.*

Broadly, the algorithm is divided into two stages. In the first stage, all rules in the grammar are converted so that they are of the form $A \to B_1 \ldots B_n$ or $A \to a$. This is the *weak Chomsky Normal Form* (Rozenberg and Salomaa [66]).

In the second stage, they are transformed to be of the form $A \to BC$. Algorithm 1 shows the steps in the transformation. This process is the same as described in Hopcroft and Ullman. The two 'for' loops represent the two stages respectively. What we need to prove is that the grammar g', result of this process, is in Chomsky Normal Form.

The way we proceed in HOL is as follows. First, we implement the two stages in HOL.

The Algorithm 1 shows the corresponding transformation relations in HOL. The boxes show the steps that form part of these two relations. The for-loops are replicated by taking the reflexive, transitive closure (RTC) of the relations.

```
input  : Grammar $g$ where $\epsilon \notin L(g)$
output: Grammar $g'$ such that $L(g) = L(g')$ and $g'$ is in CNF
begin
    transform all rules to be of the form $A \to a$ or $A \to X_1 X_2 \ldots X_m$, where $X_i$ is a nonterminal
    while $A \to X_1 X_2 \ldots X_m \in$ rules $g$ do
        if $m \geq 2$ then
            for isTerminal $X_i$ do
                trans1Tmnl
                introduce new variable $C_i$
                introduce rule $C_i \to X_i$
                replace $X_i$ by $C_i$
    transform all rules to be of the form $A \to a$ or $A \to X_1 X_2$, nonterminals $X_1, X_2$
    while $A \to X_1 X_2 \ldots X_m \in$ rules $g$ do
        trans2NT
        if $m \geq 3$ then
            introduce new variable $D_i$
            introduce rule $D_i \to X_2 X_3$
            replace $X_2 X_3$ by $D_i$
```

Algorithm 1: Algorithm for transforming a grammar into Chomsky Normal Form.

The next step is to ensure the transformations affect the grammar in the correct way. This is established by proving properties at different stages of the transformations. These are summarised in Algorithm 2.

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 $l \to pts$ (p or s is nonempty and t is a terminal) with productions $A \to t$ and $l \to pAs$.

HOL Definition 2.5.1 (trans1Tmnl)

```
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 2.5.1

```
(λx y. ∃nt t. trans1Tmnl nt 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 $A \to a$, where a is a terminal symbol or $A \to A_1 ... A_n$ 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`.

Idiosyncrasies of formalisation Algorithm 2 shows the two steps and the properties that need to be established at each stage.

The process for transformation of a grammar into Chomsky Normal Form requires introducing new variables in to the grammar. In a text proof one would just express it by simply stating '*pick a symbol not in the grammar*'. Mechanisation is too rigorous for one to be able to use such a statement straight away. Instead, we have to show that the nonterminals in a grammar are finite and if the universal set for the type is infinite *then* we can pick a nonterminal that does not belong in the grammar. The first condition (Property 0 in Algorithm 2) follows from the fact the number of rules in a grammar are finite. Since the nonterminals are derived from the rules, the number of nonterminals must be finite as well. The second condition needs to be asserted as a premise for any theorems asserting that the transformation relation holds between two grammars (*e.g.* HOL Theorem 2.5.2).

We have chosen to model the transformations as relations rather than functions in HOL. Step 1 in Algorithm 2 corresponds to the first for-loop in Algorithm 1. Since relations are partial in HOL (compared to functions which are total), we need to show that zero or more applications of `trans1Tmnl` will result in a grammar satisfying the properties 1.1 and 1.2. Hence the termination assertion in Step 1.

In addition, since we are interested in the grammar preserving certain properties, we also have to show that such multiple applications preserve the language (Property 1.1) and all the rules are of the form $A \rightarrow B_1 \ldots B_n$ and $A \rightarrow a$ (Property 1.2).

We establish similarly properties for Step 2. Step 2 corresponds to second for-loop over the rules of a grammar. At the end of the two transformations and if the stated properties are preserved then we have established that grammar g can be transformed into g' such that g' is in CNF and the language for g and g' is the same.

Another set of tedious proofs corresponds to showing that the transformation for CNF do not affect the original assumptions we made about useless symbols, unit and ϵ-productions.

```
input  : Grammar $g$ where $\epsilon \notin L(g)$
output: Grammar $g'$ such that $L(g) = L(g')$ and $g'$ is in CNF
transform all rules to be of the form $A \rightarrow a$ or $A \rightarrow X_1 X_2 \ldots X_m$, where $X_i$ is a nonterminal
Property 0 FINITE (nonTerminals $g$)
Step 1 $\exists g_2.\ (\lambda x\ y.\ \exists nt\ t.$ trans1Tmnl $nt\ t\ x\ y)^*\ g\ g_2$
Property 1.1 $L\ g = L\ g_2$
Property 1.2 badTmnlsCount $g_2$ = 0
transform all rules to be of the form $A \rightarrow a$ or $A \rightarrow X_1 X_2$, for nonterminals $X_1, X_2$
Step 2 $\exists g_3.\ (\lambda x\ y.\ \exists nt\ t.$ trans2NT $nt\ nt_1\ nt_2\ x\ y)^*\ g_2\ g_3$
Property 2.1 $L\ g_2 = L\ g_3$
Property 2.2 badTmnlsCount $g_3$ = 0
Property 2.3 badNtmsCount $g_3$ = 0
Property 2.4 isCnf $g_3$
```

Algorithm 2: Steps for transforming a grammar into CNF in HOL and the key properties to be established at each stage.

Coming back to the proof, we now want to show termination corresponding to Step 1. For this, we define the constant `badTmnlsCount`, which counts the terminals in the RHSs of all productions which have more than one terminal symbol present in their RHS.

HOL Definition 2.5.2 (`badTmnlsCount`)

```
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:

HOL Definition 2.5.3 (`ruleTmnls`)

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

(Notation $|r|$ *represents the size of 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 are finite.

HOL Theorem 2.5.2

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

(Note the use of the assumption `INFINITE 𝒰(:'nts)`. Here `𝒰(:'nts)` represents the universal set for the type of nonterminals (`nts`) in the grammar g and g'. As mentioned before, the transformation process involves introducing a new nonterminal symbol. To be able to pick a fresh symbol, the set of possible nonterminals has to be infinite.)

The above process gives us a simplified grammar g_2 such that `badTmnlsCount` g_2 `= 0`. By HOL Theorem 2.5.1 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 give us a grammar where all productions conform to the CNF criteria.

HOL Definition 2.5.4 (`trans2NT`)

```
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 (Property 2.1 in Algorithm 2).

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 $A \to A_1 A_2$ (Property 2.3 in Algorithm 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` (Property 2.2 in Algorithm 2). We can then apply the two transformations to obtain our grammar in CNF (Property 2.4 in Algorithm 2) where all rules are of the form $A \to a$ or $A \to A_1 A_2$. This is in HOL asserted by the `isCnf` predicate.

HOL Definition 2.5.5 (`isCnf`)

```
isCnf g ⟺
  ∀ℓ r.
    rule ℓ r ∈ rules g ⇒
    |r| = 2 ∧ EVERY isNonTmnlSym r ∨ |r| = 1 ∧ isWord r
```

(Predicate `isNonTmnlSym` *is true of a symbol if it is of the form* `NTS s` *for some string* `s`. `EVERY` *checks that every element of a list satisfies the given predicate.)*

The HOL theorem corresponding to Theorem 2.5.1 is:

HOL Theorem 2.5.3

```
INFINITE 𝒰(:'nts) ∧ [] ∉ L g ⇒ ∃g'. isCnf g' ∧ L g = L g'
```

2.6 Greibach Normal Form

We now move on to Greibach Normal Form where all productions start with a terminal symbol, followed by zero or more variables.

If ϵ does not belong in the language of a grammar then the grammar 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 nondeterministic pushdown automata. Productions in GNF are of the form $A \to a\alpha$ where a is a terminal symbol and α is list (possibly empty) of nonterminals.

Formally,

Theorem 2.6.1 (H&U Theorem 4.6) *[**Greibach Normal Form**] Any CFL without ϵ can be generated by a grammar in which all productions are of the form $A \rightarrow a\alpha$ or $A \rightarrow a$. Here A is a variable, α is a (possibly empty) string of variables and a is a terminal.*

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

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

Step 1 For each ordered nonterminal A_k do the following:

Step 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 2.6.1).

Step 1.2 Convert left recursive rules in the grammar to right recursive rules. This is based on `left2Right` lemma (Section 2.6.2).

Step 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 Step 1 which is implemented using relation `gnf_p1` in Section 2.6.3.1 describing the GNF algorithm. Step 1 depends on two crucial results, Step 1.1 and Step 1.2 which are established separately. Step 2 can be further subdivided into two parts covered in (Sections 2.6.3.2 and 2.6.3.3), the Step 2.1 and Step 2.2 of the algorithm, respectively. Algorithm 4 shows the different stages and the properties that need to be established at each of the stage.

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

2.6.1 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 nonterminals 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 2.6.1 (`aProdsRules`)

```
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 $x|vs|p$ denotes that variables vs that occur in x are treated as bound variables when evaluating predicate p.)

Lemma 2.6.2 (H&U Lemma 4.3) *[“**`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 $L(g) = L(g')$.*

2.6.2 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 2.6.3 (H&U Lemma 4.4) *[“**`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 2.6.1.

Relation `left2Right` A B g g' holds if and only if 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 2.6.2 (`left2Right`)

```
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 $A \to A\alpha_i$ will eventually end with a production $A \to \beta_j$. The sequence of replacements

$$A \Rightarrow A\alpha_{i_1} \Rightarrow A\alpha_{i_2}\alpha_{i_1} \Rightarrow \ldots \Rightarrow A\alpha_{i_p}\ldots\alpha_{i_1} \Rightarrow \beta_j\alpha_{i_p}\ldots\alpha_{i_1} \tag{2.1}$$

in g can be replaced in g' by

$$A \Rightarrow \beta_j B \Rightarrow \beta_j\alpha_{i_p}B \Rightarrow \ldots \Rightarrow \beta_j\alpha_{i_p}\ldots\alpha_{i_2}B \Rightarrow \beta_j\alpha_{i_p}\ldots\alpha_{i_2}\alpha_{i_1} \tag{2.2}$$

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 2.1.

HOL Theorem 2.6.1

```
left2Right A B g g' ⇒ L g = L g'
```

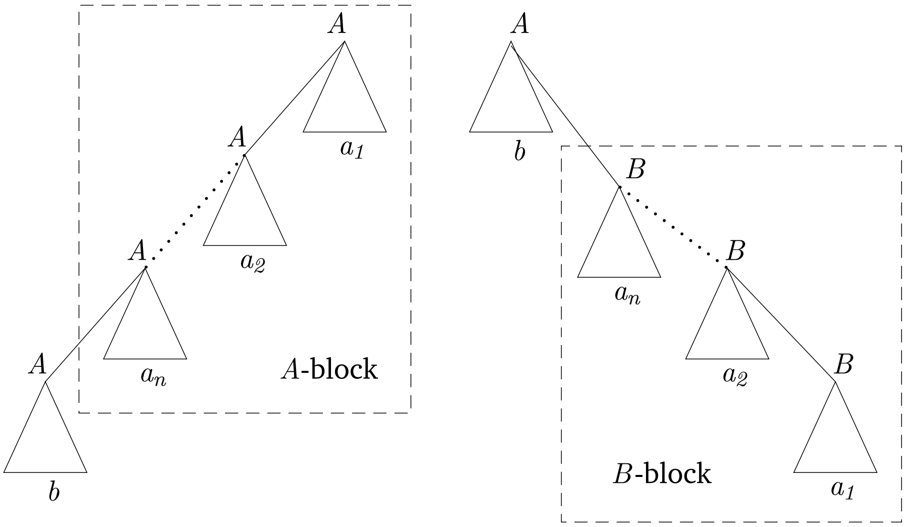


Figure 2.1 – A left recursive derivation $A \to Aa_1 \to Aa_2a_1 \to \cdots \to A_n\ldots a_2a_1 \to ba_n\ldots a_2a_1$ can be transformed into a right recursive derivation $A \to bB \to ba_n \to \cdots \to ba_n\ldots a_2 \to ba_n\ldots a_2a_1$. Here the RHS b does not start with an A.

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 (2.1) and (2.2), and a figure corresponding to our Figure 2.1. 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'.

2.6.2.1 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 2.6.2

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

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

Base Case If there are no As in the leftmost position (*i.e.* $\texttt{ldNumNt (NTS}\ A\texttt{)}\ dl = 0$) 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 2.1 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 2.2.

The division is given by HOL Theorem 2.6.3. The second $\exists$-clause of this theorem refers to the side conditions on the composition of the expansions shown in Figure 2.2.

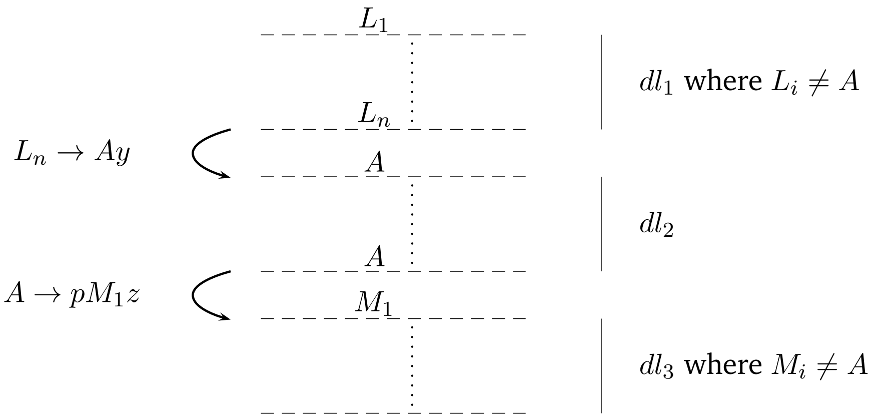


Figure 2.2 – We can split dl into dl_1, dl_2 and dl_3 such that dl_1 has no A-expansions, dl_2 consists of only A-expansions and dl_3 breaks the sequence of A-expansions so that the very first element in dl_3 is not a result of an A-expansion. The dashed lines are elements in the derivation list showing 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 2.6.3

$$
\begin{array}{l}
\texttt{lderives}\ g \vdash dl \lhd x \to y \wedge \texttt{ldNumNt (NTS}\ A\texttt{)}\ dl \neq \texttt{0} \wedge |dl| > \texttt{1} \Rightarrow \\
\exists\, dl_1\ dl_2\ dl_3. \\
\quad dl = dl_1 \texttt{ ++ } dl_2 \texttt{ ++ } dl_3 \wedge \texttt{ldNumNt (NTS}\ A\texttt{)}\ dl_1 = \texttt{0} \wedge \\
\quad (\forall\, e_1\ e_2\ p\ s.\ dl_2 = p \texttt{ ++ [}e_1\texttt{; }e_2\texttt{] ++ } s \Rightarrow |e_2| \geq |e_1|) \wedge \\
\quad \exists\, pfx. \\
\qquad \texttt{isWord}\ pfx \wedge \\
\qquad (\forall\, e.\ e \in dl_2 \Rightarrow \exists\, sfx.\ e = pfx \texttt{ ++ [NTS }A\texttt{] ++ } sfx) \wedge \\
\qquad dl_2 \neq \texttt{[]} \wedge \\
\qquad (dl_3 \neq \texttt{[]} \Rightarrow \\
\qquad\ |\texttt{LAST}\ dl_2| \leq |\texttt{HD}\ dl_3| \Rightarrow \\
\qquad\ \neg(pfx \texttt{ ++ [NTS }A\texttt{]} \preceq \texttt{HD}\ dl_3))
\end{array}
$$

(Here $x \preceq y$ holds if and only if x is a prefix of y.)

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 is easily finished if derivation dl_2 can somehow be shown to have an equivalent in g'. This is shown by proving HOL Theorem 2.6.4. The theorem states that for a derivations in g of the form given by Equation (2.1), there is an equivalent derivation in g' in the form of Equation (2.2).

HOL Theorem 2.6.4

$\texttt{left2Right}\ A\ B\ g\ g' \wedge$

```
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′
```

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

2.6.3 Stitching the pieces together

Using the `aProds` lemma and elimination 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 Step 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 Step 2.1 and Step 2.2. Each step 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 steps are applied in succession to the grammar and achieve the following results.

Step 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 nonterminals, 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.

Step 2.1 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.

Step 2.2 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.

2.6.3.1 Step 1—Ordering the A_i-productions

Step 1 is represented by the HOL relation `gnf_p1`. This corresponds to Fig 4.9 in Hopcroft and Ullman showing the first step in the GNF algorithm. The `gnf_p1` relation relates two states where the second state is the result of transforming rules for a single A_k.

Algorithm 3 shows the process for Step 1 as presented in Hopcroft and Algorithm (Fig 4.9). The extra annotations show the correspondence with the HOL implementation.

Following a similar pattern to CNF mechanisation, we have used relations to represent the transformations and RTC to imitate for-loops.

HOL Definition 2.6.3 (`gnf_p1`)

$$
\begin{array}{l}
\texttt{gnf\_p1}\ (bs_0, nts_0, g_0, seen_0, ubs_0)\ (bs, nts, g, seen, ubs) \iff \\
\quad \exists\, ntk\ b\ rules_0\ rules_1. \\
\qquad nts_0 = ntk::nts \land bs_0 = b::bs \land ubs = ubs_0 \mathbin{+\!+} [b] \land \\
\qquad seen = seen_0 \mathbin{+\!+} [ntk] \land nts = \texttt{TL}\ nts_0 \land \\
\qquad (\texttt{gnf\_p1Elem}\ ntk)^*\ (seen_0, \texttt{rules}\ g_0, [])\ ([], rules_0, seen_0) \land \\
\qquad rules_1 = \texttt{l2rRules}\ ntk\ b\ (\texttt{set}\ rules_0) \land \\
\qquad \texttt{startSym}\ g = \texttt{startSym}\ g_0 \land \texttt{set}\ (\texttt{rules}\ g) = rules_1
\end{array}
$$

(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, `gnf_p1`, holds if the rules of g are obtained by transforming rules for a single nonterminal (A_k) and using up a fresh nonterminal 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 `gnf_p1`. The first part is the relation `gnf_p1Elem`. 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 2.6.4 (`gnf_p1Elem`)

$$
\begin{array}{l}
\texttt{gnf\_p1Elem}\ ntk\ (seen_0, ru_0, sl_0)\ (seen, ru, sl) \iff \\
\quad \exists\, se. \\
\qquad seen_0 = se::seen \land sl = sl_0 \mathbin{+\!+} [se] \land \\
\qquad \texttt{set}\ ru = \texttt{aProdsRules}\ ntk\ [se]\ (\texttt{set}\ ru_0)
\end{array}
$$

```
input  : Grammar $g$ where $\epsilon \notin L(g)$
output: Grammar $g'$ such that $L(g) = L(g')$ and $g'$ is in GNF
begin
    Loop over k by using RTC over gnf_p1
    for $k := 1$ to $m$ do
        gnf_p1
        begin
            Loop over j by using RTC over gnf_p1Elem
            for $j = 1$ to $k-1$ do
                for each production of the form $A_k \rightarrow A_j\alpha$ do
                    gnf_p1Elem
                    begin
                        Calculation of aProdsRules
                        for all productions $A_j \rightarrow \beta$ do
                            add production $A_j \rightarrow \beta$
                        end
                        remove production $A_k \rightarrow A_j\alpha$
                    end
                end
            end
            rules$_1$
            Calculation of l2rRules
            for each production of the form $A_k \rightarrow A_k\alpha$ do
                begin
                    add production $B_k \rightarrow \alpha$ and $B_k \rightarrow \alpha B_k$
                    remove productions $A_k \rightarrow A_k\alpha$
                end
            end
            for each production $A_k \rightarrow \beta$ where $\beta$ does not begin with $A_k$ do
                add production $A_k \rightarrow \beta B_k$
            end
        end
    end
end
```

Algorithm 3: Step 1 of transforming a grammar into Greibach Normal Form.

Using the closure, `gnf_p1Elem`*, we can repeatedly do this transformation for all the elements in $seen_0$ to obtain the new set of rules $rules_0$ in the `gnf_p1` 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 `gnf_p1`. Thus, we show that some grammar g_1 exists such that predicate `gnf_p1`* $(B, V, g, [], [])$ $(B_1, [], g_1, V, B_2)$ holds. This means that g has successfully been transformed to g_1 such that g_1 satisfies the Step 1 conditions. More explicitly, as mentioned in Hopcroft and Ullman, the rules in g_1 should now be of the following form:

- **(C1) Ordered** A_i **rules** - if $A_i \rightarrow A_j\gamma$ is in rules of g_1, then $j > i$.
- **(C2)** A_i **rules in GNF** - $A_i \rightarrow a\gamma$, where a is a terminal symbol and γ is a string of nonterminals.
- **(C3)** B_i **rules** - $B_i \rightarrow \gamma$, where γ is in $(V \cup B_1, B_2, \ldots, B_n)^*$.

Automating an algorithm That Step 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 2.6.5 (`rhsTlNonTms`)

```
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 ∧
          ∃ nt₁ t₁.
            t = NTS nt₁::t₁ ∧ nt₁ ∈ set ntsl DIFF set bs
```

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

HOL Definition 2.6.6 (seenInv)

```
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 Step 2.2), but in HOL mechanisation needs to be proven at this stage.

HOL Definition 2.6.7 (rhsBNonTms)

```
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.)*

Note that between the invariants rhsTlNonTms and rhsBNonTms we have that the grammar at this stage is in *Greibach Intermediate Form* (GIF). A grammar is in Greibach Intermediate Form if each rule leads to a terminal, a terminal followed by some nonterminals, or a string of nonterminals. Notice that Chomsky Normal Form also satisfies GIF thus making GIF an invariant for the first step of the transformation to GNF.

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 may 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.

Each of the steps in the GNF algorithm results in a grammar. Both the original and the resulting grammars have to satisfy certain properties. Algorithm 4 shows the different properties that are established in the process of transforming the original grammar into Greibach Normal Form.

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

HOL Theorem 2.6.5

$$
\begin{array}{l}
\texttt{gnf\_p1}^{*}\ (bs_0, nts_0, g_0, seen_0, ubs_0)\ (bs, nts, g, seen, ubs)\ \wedge \\
|bs_0| \geq |nts_0| \wedge \texttt{ALL\_DISTINCT}\ bs_0 \wedge \texttt{ALL\_DISTINCT}\ nts_0 \wedge \\
\texttt{set}\ (\texttt{ntms}\ g_0) \cap \texttt{set}\ bs_0 = \emptyset \wedge \texttt{set}\ bs_0 \cap \texttt{set}\ ubs_0 = \emptyset \wedge \\
\texttt{set}\ nts_0 \cap \texttt{set}\ seen_0 = \emptyset \Rightarrow \\
L\ g_0 = L\ g
\end{array}
$$

(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 (from the original grammar) and the fresh B_is that have been used up as part of our states. Because of this, extra assertions about these seen lists have to provided. Once ‘seen’, any nonterminal from the original grammar 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 `gnf_p1`.

The above proof becomes trivially true if relation `gnf_p1` fails to hold. To counter this problem, we show that such a transformation does exist for any start state.

HOL Theorem 2.6.6

$$
\begin{array}{l}
|bs_0| \geq |nts_0| \wedge \texttt{ALL\_DISTINCT}\ bs_0 \wedge \texttt{ALL\_DISTINCT}\ nts_0 \wedge \\
\texttt{set}\ nts_0 \cap \texttt{set}\ seen_0 = \emptyset \wedge \texttt{set}\ (\texttt{ntms}\ g_0) \cap \texttt{set}\ bs_0 = \emptyset \wedge \\
\texttt{set}\ bs_0 \cap \texttt{set}\ ubs_0 = \emptyset \Rightarrow \\
\exists g. \\
\quad \texttt{gnf\_p1}^{*}\ (bs_0, nts_0, g_0, seen_0, ubs_0) \\
\quad\quad (\texttt{DROP}\ |nts_0|\ bs_0, [\,], g, seen_0 \ \texttt{++}\ nts_0, \\
\quad\quad\ \ ubs_0 \ \texttt{++}\ \texttt{TAKE}\ |nts_0|\ bs_0)
\end{array}
$$

(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.

```
input  : Grammar $g$ where $g$ is in CNF
output: Grammar $g'$ such that $L(g) = L(g')$ and $g'$ is in GNF
transform the rules in $g$ to give a new grammar $g_1 = (V_1, T, P_1, S)$ such that if
$A_i \to A_j\alpha$ is a rule of $g_1$, then $j > i$.
Property 0.1 FINITE (nonTerminals $g_0$)
Property 0.2 rhsTlNonTms (rules $g_0$) (ntms $g$) $ubs_0$
Property 0.3 seenInv (rules $g_0$) $seen_0$
Property 0.4 rhsBNonTms (rules $g_0$) $ubs_0$
Step 1
  $\exists\, g_1$.
    gnf_p1$^*$ ($bs_0$, $nts_0$, $g_0$, $seen_0$, $ubs_0$)
      (DROP $|nts_0|$ $bs_0$, [], $g_1$, $seen_0$ ++ $nts_0$,
       $ubs_0$ ++ TAKE $|nts_0|$ $bs_0$)
Property 1.1 $L$ $g_0$ = $L$ $g_1$
Property 1.2
  rhsTlNonTms (rules $g_1$) (ntms $g_1$) ($ubs_0$ ++ TAKE $|nts_0|$ $bs_0$)
Property 1.3 seenInv (rules $g_1$) ($seen_0$ ++ $nts_0$)
Property 1.4 rhsBNonTms (rules $g_1$) ($ubs_0$ ++ TAKE $|nts_0|$ $bs_0$)
Property 1.5 set (ntms $g_1$) $\subseteq$ set (ntms $g_0$)
all rules of the form $A_i \to A_j\beta$ in $g_1$ are replaced by $A_i \to a\alpha\beta$, where $A_j \to a\alpha$
Step 2.1 $\exists\, g_2$. fstNtm2Tm$^*$ ($ontms_1$, $g_1$, $s_1$) ([], $g_2$, $ontms_1$ ++ $s_1$)
Property 2.1.1 $L$ $g_1$ = $L$ $g_2$
Property 2.1.2 rhsBNonTms (rules $g_2$) ($ubs_0$ ++ TAKE $|nts_0|$ $bs_0$)
Property 2.1.3 set (ntms $g_2$) $\subseteq$ set (ntms $g_1$)
Property 2.1.4 gnfInv (rules $g$) ($ontms_1$ ++ $s_1$)
all rules of the form $B_k \to A_i\beta$ in $g_2$ are replaced with $B_k \to a\alpha\beta$, where $A_i \to a\alpha$
Step 2.2
  $\exists\, g_3$.
    fstNtm2TmBrules$^*$ ($ubs_0$, $ontms_0$, $g_0$, $seen_0$)
      ([], $ontms_0$, $g$, $seen_0$ ++ $ubs_0$)
Property 2.2.1 $L$ $g_2$ = $L$ $g_3$
Property 2.1.3 set (ntms $g_3$) $\subseteq$ set (ntms $g_2$)
Property 2.1.4 gnfInv (rules $g_3$) ($ontms_0$ ++ $seen_0$)
```

Algorithm 4: Steps for transforming a grammar into GNF in HOL and the key properties to be established at each stage.

Note: Everywhere a relation such as `gnf_p1`, `gnf_p1Elem` is used we have provided proofs that the relation always holds.

2.6.3.2 Step 2.1—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 CNF and only has useful nonterminals, rule $A_n \to a$ is in g_1, where a is in T. A_n is the highest ranked nonterminal and as such cannot expand to an RHS starting with a nonterminal.

Thus, if we transform nonterminals V using `aProds`, starting from the highest rank, we are bound to get rules of the form, $A_k \to a\alpha$, 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 2.6.8 (`fstNtm2Tm`)

$$
\begin{array}{l}
\texttt{fstNtm2Tm}\ (\mathit{ontms}_0, g_0, \mathit{seen}_0)\ (\mathit{ontms}, g, \mathit{seen}) \iff \\
\quad \exists\, \mathit{ntk}\ \mathit{rules}_0. \\
\quad\quad \mathit{ontms}_0 = \mathit{ontms} \mathbin{+\!+} [\mathit{ntk}] \land \mathit{seen} = \mathit{ntk}::\mathit{seen}_0 \land \\
\quad\quad (\texttt{gnf\_p1Elem}\ \mathit{ntk})^*\ (\mathit{seen}_0, \texttt{rules}\ g_0, [\,])\ ([\,], \mathit{rules}_0, \mathit{seen}_0) \land \\
\quad\quad \texttt{rules}\ g = \mathit{rules}_0 \land \texttt{startSym}\ g = \texttt{startSym}\ g_0
\end{array}
$$

(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 2.6.9 (`gnfInv`)

$$
\begin{array}{l}
\texttt{gnfInv}\ \mathit{ru}\ s \iff \\
\quad \forall\, i. \\
\quad\quad i < |s| \Rightarrow \\
\quad\quad \forall\, r.\ \texttt{rule}\ (\texttt{EL}\ i\ s)\ r \in \mathit{ru} \Rightarrow \texttt{validGnfProd}\ (\texttt{rule}\ (\texttt{EL}\ i\ s)\ r)
\end{array}
$$

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

Multiple applications of this process result in g_2 satisfying the Step 2.1 condition that all the rules for nonterminals in V are now in GNF.

2.6.3.3 Step 2.2—Changing B_i-productions to GNF

The final step 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 Step 2.1 to convert all B_i-productions to GNF.

At the end of Step 2.1, all rules involving nonterminals in V are of the form $A_i \to a\alpha$, 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 $B_i \to A_k\beta$ where β is a list (possibly empty) of nonterminals. The `aProds` lemma is now used to obtain rules of the form $B_i \to a\alpha\beta$. We then show that these rules satisfy the GNF criertia by establishing that `gnfInv` holds for seen B_is.

HOL Definition 2.6.10 (`fstNtm2TmBrules`)

$$
\begin{array}{l}
\texttt{fstNtm2TmBrules}\ (\mathit{ubs}_0, \mathit{ontms}_0, g_0, \mathit{seen}_0)\ (\mathit{ubs}, \mathit{ontms}, g, \mathit{seen}) \iff \\
\quad \exists\, b\ \ \mathit{rules}_0. \\
\qquad \mathit{ubs}_0 = b :: \mathit{ubs} \land \mathit{ontms}_0 = \mathit{ontms} \land \mathit{seen} = \mathit{seen}_0 \mathbin{++} [b] \land \\
\qquad \mathit{rules}_0 = \texttt{aProdsRules}\ b\ \mathit{ontms}\ (\texttt{set}\ (\texttt{rules}\ g_0)) \land \\
\qquad \texttt{set}\ (\texttt{rules}\ g) = \mathit{rules}_0 \land \texttt{startSym}\ g = \texttt{startSym}\ g_0
\end{array}
$$

($ubs0$ 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_is that have been handled.)

The above transformation results in grammar g_3 (from Algorithm 4) preserves the language of the grammar.

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 steps can be combined to show Theorem 2.6.1, that any grammar that can be transformed into Chomsky Normal Form can be subsequently transformed into a grammar in Greibach Normal Form.

HOL Theorem 2.6.7

$$\texttt{INFINITE}\ \mathcal{U}(\texttt{:'nts}) \land [] \notin L\ g \Rightarrow \exists g'.\ \texttt{isGnf}\ g' \land L\ g = L\ g'$$

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

2.7 Conclusions

As mentioned before, Nipkow has formalised some of the theory of regular languages, verifying a lexical analyzer generator obtained from a regular expression [52]. The executability of the definitions provided by Nipkow breaks down when dealing with the inductively defined transitive closure (*). Because of the manner in which we have implemented our grammar, all the items that result are finite. This makes it easier for us to present an executable $closure$ function in Section 5.5, even though we do end up proving numerous theorems about finiteness.

We have discussed the two most well-known normal forms for CFGs, the Chomsky Normal Form and the Greibach Normal Form. Our mechanisation follows the treatment as presented in Hopcroft and Ullman, a standard textbook for introducing language and automata theory at university level. Of course other algorithms for normalisation exist but we believe that their mechanisation should be an easier process from here on.

Our treatment of the proofs (apart from the extent of detail expected of any mechanisation) differs in only one aspect from Hopcroft and Ullman. We use a relational approach instead of a functional one. For example, we use closures to represent for-loops. We chose this approach so that we could leverage the already existing relational theory present in HOL. In the cases of the simplifications related to removing useless symbols, unit and ϵ-productions and Greibach Normal Form, this was a big additional advantage. Using relations allowed us to use set comprehension further simplifying our proofs. With most choices, there's always a competing concern. The readability gained by using set comprehension meant that the resulting functions were not executable in our mechanisation. In this case, this was not a major concern. As we will see later on in Chapter 5, executability for a parser is of higher value. For mechanisation in such a case, the focus shifts from readability. To be able to use 'nicer' definitions requires quite a bit of extra effort which usually goes into proving equivalence between the readable and the executable definitions.

Table 2.1 shows the proof effort for the mechanisation covered in this chapter.

The LOC count excludes proofs that are in the HOL library as well as the library maintained by us. 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

Mechanisation		LOC	#Definitions	#Proofs
ϵ-productions		299	5	20
Generating symbols		174	2	13
Reachable symbols		239	2	21
Unit productions		785	7	45
Chomsky Normal Form		1438	13	96
Greibach Normal Form	aProds lemma	219	2	8
	left2Right lemma	3001	23	84
	Final GNF algorithm	2723	16	91

Table 2.1 – Summary of the mechanisation effort for simplification of CFGs

case is very sparse. It is problematic to frame general techniques when the majority of the results require carefully combing the fine details 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.

A mechanised proof is more than the sum of the steps of a textual proof. It goes beyond filling in logical gaps and identifying the missing or obvious deductions that need to be made explicit in a theorem prover. New strategies such as concrete derivation lists and special structures for induction for proving equivalence between left and right recursion, that go beyond the textual treatment, were developed to tackle existing proofs. At times, these strategies are not clear from the original proofs.

A good example of such a case is the mechanisation of Greibach Normal Form. The GNF turned out to be much harder than anticipated especially compared to formalisation of CNF. When we started off with the mechanisation of the theory presented, we encountered difficulties with GNF. Our main focus at that time was parsing so after a few tries GNF was abandoned. In the process of developing the theory of PDAs we implemented what we have referred to as derivation lists (introduced in Section 2.1). Even though it has been presented upfront, in actual development it was implemented much later. With this in hand, we were able to swiftly knock off the lemmas required for the final GNF proof. This technique of representing derivations as concrete lists was not as obvious when dealing with GNF as it was when working with PDAs (Chapter 3). It has simplified many proofs which might have taken longer otherwise. On the other hand, as we will see later on in Chapter 4, the use of this construct made the proof for the pumping lemma highly complicated and almost incomprehensible!

Chapter 3

Pushdown Automata

> Roy: [*singing*] We don't need no education.
> Moss: Yes you do. You've just used a double negative.
>
> IT Crowd

Contents

This chapter covers the design of pushdown automata (PDAs) which are the accepting devices for context-free languages. PDAs are "predicting machines" that use knowledge about their stack contents to determine whether and how a given string can be generated by the grammar. Using context-free grammars and pushdown automata one can construct efficient parsing algorithms. For example, PDAs can be used to to build efficient parsers for LR grammars (see Chapter 5). PDAs have also been used to provide elegant proofs of properties such that context-free grammars are closed under inverse homomorphism (Chapter 4). PDAs are essentially transition devices and hence are useful for modeling systems (Bouajjani et al. [13]).

The relationship between context-free grammars and pushdown automata was first described by Chomsky in 1962 (Chomsky [16]). In 1960, before Chomsky's formal description, Yngve [81] used a machine closely related to a pushdown automaton for modeling transient memory required by the human processor to look at sentences generated by CFGs. Yngve's work is still applied in studies of short-term memory and sentence processing (Murata et al. [49]).

In this chapter we start off by establishing the background for PDAs in Section 3.1. We define a PDA and describe the two notions of acceptance of an input by a PDA. In Section 3.2 we discuss the mechanisation of a key property that the two modes of acceptance are equivalent. This will be relevant in the proofs to come thereafter. In Sections 3.3.1 and 3.3.2 we provide mechanisation of the proof that the class of languages accepted by PDAs is precisely the class of context-free languages.

Contributions

- ◇ A formalisation of PDAs in HOL (Section 3.1).
- ◇ Proof that the languages accepted by PDAs by final state are exactly the languages accepted by PDAs by empty stack (Section 3.2).
- ◇ Formalisation of the algorithm to construct a PDA for a context-free grammar (Section 3.3.1) and vice versa (Section 3.3.2) and the mechanisation of the result that the two formalisms are equivalent in power.

Parts of this chapter have been published in [8].

3.1 The theory of PDAs

The PDA is modeled as a record containing the start state (`start` or q_0), the starting stack symbol (`ssSym` or Z_0), list of final states (`final` or F) and the next state transitions (`next` or δ). Depending on how the next state transitions are defined, the PDA can either be deterministic or nondeterministic.

HOL Definition 3.1.1 (PDA)

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

The input alphabets (Σ), stack alphabets (Γ) and the states for the PDA (Q) can be easily extracted from the above information. In the proofs, we will refer to a PDA M

as the tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ for easy access to the components. We have used lists instead of sets to avoid unnecessary finiteness constraints in our proofs.

The `trans` type implements a single transition. 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 `trans` type describes a transition in the PDA’s state machine. The *next* field of the 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 $M \vdash (q, i :: \alpha, s) \rightarrow (q', i', s')$ 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` $M \vdash \ell \lhd x \rightarrow y$ 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 the state where the stack is empty and only ϵ moves are possible in the state 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 done reading the input, and the stack contents are irrelevant.

HOL Definition 3.1.2 (lafs)

```
lafs p =
  {w |
   ∃state stack.
     p ⊢ (p.start, w, [p.ssSym]) →* (state, [], stack) ∧
     state ∈ p.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. The two criteria for acceptance give us two different PDAs but having the same language.

HOL Definition 3.1.3 (laes)

```
laes p =
  {w | ∃state. p ⊢ (p.start, w, [p.ssSym]) →* (state, [], [])}
```

When the acceptance is by empty stack, the set of final states is irrelevant, so we usually let the list of final states be empty.

To be consistent with the notation in Hopcroft and Ullman, predicate `lafs` M is referred to as $L(M)$ and predicate `laes` M is referred to as $N(M)$ in the proofs to follow.

3.2 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 the following two claims.

- ◇ If a PDA accepts strings using final state acceptance then we can construct a corresponding equivalent PDA that accepts the same strings using empty stack acceptance criterion.
- ◇ If a PDA accepts strings using empty stack, then we can construct a corresponding PDA that accepts the strings using a final state.

3.2.1 PDA construction for acceptance by empty stack

Theorem 3.2.1 (H&U Theorem 5.1) *If L is $L(M_2)$ for some PDA M_2, then L is $N(M_1)$ for some PDA, M_1.*

Proof Let $M_2 = (Q, \Sigma, \Gamma, \delta, q_0, m, Z_0, F)$ be a PDA such that $L = L(M_2)$. We define M_1 as follows. Let $M_1 = (Q \cup q_e, q_0', \Sigma, \Gamma \cup X_0, \delta', q_0', X_0, \phi)$, where δ' is defined as follows.

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 in Q, a in Σ or $a = \epsilon$, and Z in Γ.

M_1 simulates M_2 by first putting a stack marker for M_2 (Z_0) on its stack (Rule 1). The stack for M_1 starts off with the bottom of stack marker X_0. This is to ensure 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 allows M_1 the choice of entering the state q_e and erasing the remaining stack contents or to continue simulating M_2 when M_2 has entered 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. One should note that M_2 may possibly erase its entire stack for some input x not in $L(M_2)$. This is the reason M_1 has its own special bottom-of-stack marker.

The corresponding construction of the new machine (newm) in HOL is:

HOL Definition 3.2.1 (newm)

```
newm p (q0,x0,qe) =
  (let d =
        [((NONE,x0,q0),p.start,[p.ssSym; x0])] ++ p.next ++
        finalStateTrans qe p.final (x0::stkSyms p p.next) ++
        newStateTrans qe (x0::stkSyms p p.next)
   in
     ⟨start := q0; ssSym := x0; next := d; final := []⟩)
```

where, finalStateTrans implements the Rule 2 of the construction, newStateTrans implements Rule 3. Rule 4 simply mimics the original machine transitions (p.next). Function stkSyms returns the stack alphabets Γ.

We first prove that $x \in L(M_2) \Rightarrow x \in N(M_1)$.
Let x be in $L(M_2)$. Then $(q_0, x, Z_0) \vdash^*_{M_2} (q'_0, \epsilon, \gamma)$ for some q in F. Consider M_1 with 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, thus

$(q_0, x, Z_0) \vdash^*_{M_1} (q, \epsilon, \gamma)$.

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. Thus we have,

$(q'_0, x, X_0) \vdash_{M_1} (q_0, x, Z_0X_0) \vdash^*_{M_1} (q, \epsilon, \gamma X_0)$.

As an aside the premise (italicised) is deemed sufficient for deducing the above equation. This is the case for not just the presentation in Hopcroft and Ullman. Such a self-explanatory statement suffices in all presentations of this proof. It is statements like these that need to be caught and further elaborated in a mechanised version of the proof. At times, such assumptions may not even be explicitly vocalised in the text itself.

In this case we have to prove this statement in HOL before we can make any further progress.

HOL Theorem 3.2.1

$m \vdash$ `(`q`,` x`,` stk`)` $\rightarrow^*$ `(`q'`,` x'`,` stk'`)` $\Rightarrow$
$\forall \ell.\ m \vdash$ `(`q`,` x`,` stk `++` ℓ`)` $\rightarrow^*$ `(`q'`,` x'`,` stk' `++` ℓ`)`

Coming back 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.

This is our HOL theorem:

HOL Theorem 3.2.2

$x_0 \notin$ `stkSyms` $m \wedge q'_0 \notin$ `states` $m \wedge qe \notin$ `states` $m \Rightarrow$
$x \in$ `lafs` $m \Rightarrow$
$x \in$ `laes (newm` m `(`q'_0`,`x_0`,`qe`))`

Conversely, if $x \in N(M_1) \Rightarrow x \in L(M_2)$. If M_1 accepts x by empty stack, M_2 can simulate M_1 by the following sequence of moves. The first move is by Rule 1, then a sequence of moves by Rule 2 in which M_1 simulates acceptance of x by M_2, followed by the erasure of M_1's stack using Rules 3 and 4. Thus x must be in $L(M_2)$. This is our HOL theorem:

HOL Theorem 3.2.3

$x_0 \notin$ `stkSyms` $m \wedge q'_0 \notin$ `states` $m \wedge qe \notin$ `states` $m \wedge q'_0 \neq qe \Rightarrow$
$x \in$ `laes (newm` m `(`q'_0`,`x_0`,`qe`))` $\Rightarrow$
$x \in$ `lafs` m

With HOL Theorems 3.2.2 and 3.2.3 in hand, we can now conclude:

HOL Theorem 3.2.4

`INFINITE` $\mathcal{U}$`(:'ssym)` $\wedge$ `INFINITE` $\mathcal{U}$`(:'state)` $\Rightarrow$
$\forall m.\ \exists m'.$ `lafs` m `=` `laes` m'

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

This is the HOL statement for Theorem 3.2.1.

Note that there are two extra conditions in the premise of the HOL statement. The proof works by constructing a new PDA M_1 according to the rules discussed and providing it as a witness. The extra assertions correspond to the construction of machine M_1. M_1's construction requires introducing two new states and a new symbol. With respect to HOL types, one can pick a fresh instance of a type if and only if the type is infinite and the set of values for that type in the PDA, *i.e.* the PDA states and the stack symbols are finite. The former gives rise to the two new conditions that form the part of the theorem statement in HOL. The finiteness of the states and the stack symbols had to be proven as part of the mechanisation process.

3.2.2 PDA construction for acceptance by final state

Now we present the construction of a PDA that accepts inputs via the final state criterion that is equivalent to the given PDA accepting input via the empty stack criterion.

Theorem 3.2.2 (H&U Theorem 5.2) *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 $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \phi)$ be a PDA such that $L = N(M_1)$. 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 as follows:

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 3.2.2 (newm')

```
newm' p (q0',x0,qf) =
  (let d =
        [((NONE,x0,q0'),p.start,[p.ssSym; x0])] ++ p.next ++
        MAP (toFinalStateTrans x0 qf) (states p)
   in
     ⟨start := q0'; ssSym := x0; next := d; final := [qf]⟩)
```

(Function `toFinalStateTrans` x_0 *qf st returns the next state transition* `((NONE,`x_0`,`*st*`),`*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 3.2.1 to establish $L(M_2) = N(M_1)$ by proving the following subgoals.

First, if $x \in N(M_1) \Rightarrow x \in L(M_2)$:

HOL Theorem 3.2.5

```
x0 ∉ stkSyms m ∧ q0' ∉ states m ∧ qf ∉ states m ⇒
x ∈ laes m ⇒
x ∈ lafs (newm' m (q0',x0,qf))
```

Then, if $x \in L(M_2) \Rightarrow x \in N(M_1)$:

HOL Theorem 3.2.6

$x_0 \notin$ `stkSyms` $m \wedge q_0' \notin$ `states` $m \wedge qf \notin$ `states` $m \wedge q_0' \neq qf \Rightarrow$
$x \in$ `lafs (newm'` m `(`q_0', x_0, qf`))` $\Rightarrow$
$x \in$ `laes` m

From HOL Theorems 3.2.5 and 3.2.6 we can deduce:

HOL Theorem 3.2.7

`INFINITE` $\mathcal{U}$`(:'ssym)` $\wedge$ `INFINITE` $\mathcal{U}$`(:'state)` $\Rightarrow$
$\forall m.\ \exists m'.$ `laes` m `=` `lafs` m'

Similar to HOL Theorem 3.2.4 we have to provide the assertion about the universe of the types of symbols and states in the PDA being infinite.

3.3 Equivalence of acceptance by PDAs and CFGs

Now that we have established that the two forms of acceptance of input by a PDA are equivalent, we can tackle the next set of proofs. This is the fundamental result that the languages accepted by a PDA are exactly the languages accepted by a CFG.

In the proofs to come we rely on acceptance by empty stack for a PDA. Following Hopcroft and Ullman, we first mechanise the construction of an equivalent PDA for a context-free grammar in Greibach Normal Form (Section 3.3.1). To finish off the proof we show that we can construct an equivalent CFG for a given PDA (Section 3.3.2).

3.3.1 Constructing a PDA from a CFG

Theorem 3.3.1 (H&U Theorem 5.3) *Let L be a context-free language. Then there exists a PDA M such that $L = N(M)$.*

Proof Let $G = (V, T, P, S)$ be a context-free grammar in Greibach Normal Form generating L[1]. 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 nonterminal symbols (`isGnf`). The automaton for the grammar is constructed by creating transitions from the grammar productions, $A \ \rightarrow \ a\alpha$

[1] Note that we already have the mechanised result that any grammar without an empty word can be transformed into Greibach Normal Form (Section 2.6).

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 nonterminals are the stack symbols for the PDA.

HOL Definition 3.3.1 (grammar2pda)

```
trans q (rule 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.)*

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

$$S \overset{l}{\Rightarrow}{}^{*}\ x\alpha \text{ by a leftmost derivation if and only if } (q, x, A)\ \rightarrow^{*}_{M}\ (q, \epsilon, \alpha) \tag{3.1}$$

First we suppose that $(q, x, S) \rightarrow^{i} (q, \epsilon, \alpha)$ and show by induction that $S \Rightarrow^{*} x\alpha$. The basis, $i = 0$, is trivial since $x = \epsilon$ and $\alpha = S$. For the induction, suppose $i \geq 1$, and let $x = \gamma a$. Looking at the next-to-last step,

$$(q, ya, S) \rightarrow^{i-1} (q, a, \beta) \rightarrow (q, \epsilon, \alpha). \tag{3.2}$$

This turns out to be straightforward process in HOL and is done by representing the both the machine derivations and grammar derivations using derivation lists. Let *dl* represent the derivation from (q, x, A) to (q, ϵ, α) in the machine and *dl′* represent the grammar derivation from S to $x\alpha$. Then an induction on *dl* gives us the "if" portion of (3.1) and induction on *dl′* gives us the "only if" portion of (3.1). Thus we are able to conclude Theorem 3.3.1 which in HOL is:

HOL Theorem 3.3.1

```
∀g. isGnf g ⇒ ∃m. x ∈ L g ⟺ x ∈ laes m
```

The "if" direction: Hopcroft and Ullman prove the "if" direction by showing

$$(q, x, S) \vdash^{*} (q, \epsilon, \epsilon) \Rightarrow S \Rightarrow^{*}_{g} x \tag{3.3}$$

We instead prove a more general theorem based on the structure of the sentential forms derived from a grammar in GNF. Each sentential form can be written as $pfx\ {+}{+}\ sfx$ such that *pfx* consists of terminals and *sfx* consists of nonterminals, giving the following result.

HOL Theorem 3.3.2

```
ID (grammar2pda g q) ⊢ dl ◁
    (q,pfx ++ sfx,stk) → (q,sfx,stk′) ∧ isGnf g ⇒
(lderives g)* stk (pfx ++ stk′)
```

If we remove a from the end of the input string in the first i IDs of the sequence (3.2), we discover that $(q, \gamma, S) \rightarrow^{i-1} (q, \epsilon, \beta)$, since a can have no affect on the moves of M until it is actually consumed from the input. By the induction hypothesis we have $S \Rightarrow y\beta$. The move $(q, a, \beta) \rightarrow (q, \epsilon, \alpha)$ gives us that $\beta = A\gamma$ for some A in V, $A \rightarrow a\eta$ is a production of G and $\alpha = \eta\gamma$. Hence,

$$S \overset{l}{\Rightarrow}{}^{*} \gamma\beta \overset{l}{\Rightarrow} ya\eta\gamma = x\alpha,$$

and we conclude the "if" portion of (3.1). By providing *pfx* as empty and *sfx* as the start symbol, we arrive at the following conclusion.

HOL Theorem 3.3.3

```
x ∈ laes (grammar2pda g q) ∧ isGnf g ⇒ x ∈ L g
```

The "only if" direction: Now suppose that $S \overset{l}{\Rightarrow}{}^{i} x\alpha$ by a leftmost derivation. We show by induction on *dl* that $(q, x, S) \rightarrow^{*} (q, \epsilon, \alpha)$. We use a similar strategy to the above proof to prove the following statement.

HOL Theorem 3.3.4

```
lderives g ⊢ dl ◁ [NTS (startSym g)] → (x ++ y) ∧ isGnf g ∧
isWord x ∧ EVERY isNonTmnlSym y ⇒
grammar2pda g q ⊢ (q,x,[NTS (startSym g)]) →* (q,[],y)
```

The basis, $i = 0$, is again trivial. Let $i \geq 1$ and suppose

$$S \overset{l}{\Rightarrow}{}^{i-1} yA\gamma \overset{l}{\Rightarrow} ya\eta\gamma,$$

where $x = ya$ and $\alpha = \eta\gamma$. By the inductive hypothesis, $(q, y, S) \rightarrow^{*} (q, \epsilon, A\gamma)$ and thus $(q, ya, S) \rightarrow^{*} (q, a, A\gamma)$. Since $A \rightarrow a\eta$ is a production, it follows that $\delta(q, a, A)$ contains (q, η). Thus

$$(q, x, S) \rightarrow^{*} (q, a, A\gamma) \rightarrow (q, \epsilon, \alpha),$$

and the "only if" portion of (3.1) follows. Again by providing x as the terminal string that gets derived and y as empty, we get the following.

HOL Theorem 3.3.5

```
x ∈ L g ∧ isGnf g ⇒ x ∈ laes (grammar2pda g q)
```

Note that (3.1) with $\alpha = \epsilon$ says $S \Rightarrow x$ if and only if $(q, x, S) \rightarrow^{*} (q, \epsilon, \epsilon)$. That is, x is in $L(G)$ if and only if x is in $N(M)$.

3.3.2 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 state of the transition, the top stack symbol and the 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 $[q, A, q_{m+1}] \rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q_{m+1}]$ for each $q, q_1, q_2, \ldots, q_{m+1}$ in Q, each a in $\Sigma \cup \{\epsilon\}$, and $A, B_1, B_2, \ldots, B_m$ in Γ, such that $\delta(q, a, A)$ contains $(q_1, B_1 B_2 \ldots B_m)$ (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. In other words, $[q, A, p]$ derives x if and only if x causes M to erase an A from its stack by some sequence of moves beginning in state q and ending in state p. The input string form the set T^* while the nonterminals are used to represent the stack content.

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 nonterminals 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 $B_1 \ldots B_m$;

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 nonterminal exists on the RHS of the rule then

- **C2.4.1** α is composed of only nonterminals;

C2.4.2 a nonterminal is an object of the form $[q, A, p]$ for the 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 nonterminals must be the same;

C2.4.5 the states encoded in the nonterminals 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 3.3.2 (pda2grammar)

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

The nonterminals are a tuple of a from-state, a stack symbol and a to-state where 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 HOL Definition 3.3.3) enforces the conditions C2.1, C2.2, C2.3 (capturing the four possibilities for a rule, $A \rightarrow \epsilon$; $A \rightarrow a$, $A \rightarrow \alpha$, $A \rightarrow a\alpha$, where a is a terminal symbol and $A \rightarrow \alpha$ for nonterminals α).

HOL Definition 3.3.3 (`p2gRules`)

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

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 nonterminals). The component $[q, A, p]$ is interpreted as a nonterminal 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 3.3.4 (`ntslCond`)

```
ntslCond M (q',ql) ntsl ⟺
  EVERY isNonTmnlSym ntsl ∧
  (∀e₁ e₂ p s. ntsl = p ++ [e₁; e₂] ++ s ⇒ adj e₁ e₂) ∧
  frmState (HD ntsl) = q' ∧ toState (LAST ntsl) = ql ∧
  (∀e. e ∈ ntsl ⇒ toState e ∈ states M) ∧
  ∀e. e ∈ ntsl ⇒ frmState e ∈ states M
```

(The `;` is used to separate elements in a list and `LAST` 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 3.3.2 (H&U Theorem 5.4) *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

$$(q, x, A) \;\rightarrow_M^*\; (p, \epsilon, \epsilon) \text{ iff } [q, A, p] \overset{l}{\Rightarrow}_G^* x\ . \tag{3.4}$$

3.3.2.1 Proof of the "if" portion

First we present the proof of the "if" portion of Equation (3.4). We show by induction on i that if $(q, x, A) \rightarrow^i (p, \epsilon, \epsilon)$, then $[q, A, p] \Rightarrow^* x$.

HOL Theorem 3.3.6

```
ID M ⊢ dl ◁ (q,x,[A]) → (p,[],[]) ∧ isWord x ∧
pda2grammar M g ⇒
(derives g)* [NTS (q,A,p)] 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 $(q, a\gamma, A) \;\rightarrow\; (q_1, \gamma, B_1B_2 \ldots B_n) \;\rightarrow^{i-1}\; (p, \epsilon, \epsilon)$. 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 $\gamma = \gamma_1\gamma_2 \ldots \gamma_n$ 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 since B_1 may be changed if it is at the top of the stack and is replaced by one or more symbols. However, none of $B_2, B_3, \ldots, B_n$ are ever at the top while γ_1 is being read and so cannot be changed or influence the computation.

In general, B_j remains on the stack unchanged while $\gamma_1, \gamma_2, \ldots, \gamma_{j-1}$ is read. There exist states $q_2, q_3, \ldots, q_{n+1}$, 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, $(q_1, \gamma, B_1B_2 \ldots B_n) \;\rightarrow^i\; (p, \epsilon, \epsilon)$, this is asserted in HOL by constructing a list of objects or tuples $(q_0, \gamma_j, B_j, q_n)$ (combination of the object's from-state, input, stack symbols and to-state), such that

$(q_0, \gamma_j, B_j) \ \rightarrow^i \ (q_n, \epsilon, \epsilon)$, 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 3.3.7.

HOL Theorem 3.3.7

```
ID M ⊢ dl ◁ (q,inp,stk) → (qf,[],[]) ⇒
∃ℓ.
  inp = FLAT (MAP tupinp ℓ) ∧ stk = MAP tupstk ℓ ∧
  (∀e. e ∈ MAP tuptost ℓ ⇒ e ∈ states M) ∧
  (∀e. e ∈ MAP tupfrmst ℓ ⇒ e ∈ states M) ∧
  (∀h t.
     ℓ = h::t ⇒
     tupfrmst h = q ∧ tupstk h = HD stk ∧
     tuptost (LAST ℓ) = qf) ∧
  ∀e1 e2 pfx sfx.
    ℓ = pfx ++ [e1; e2] ++ sfx ⇒
    tupfrmst e2 = tuptost e1 ∧
    ∀e.
      e ∈ ℓ ⇒
      ∃m.
        m < |dl| ∧
        NRC (ID M) m (tupfrmst e,tupinp e,[tupstk e])
          (tuptost e,[],[])
```

(Relation NRC R m x y *is the RTC closure of* R *from* x *to* y *in* m *steps. For tuple* $(q_0, \gamma_j, B_j, q_n)$*,* tupinp $(q_0, \gamma_j, B_j, q_n) = \gamma_j$*,* tupstk $(q_0, \gamma_j, B_j, q_n) = B_j$*,* tupfrmst $(q_0, \gamma_j, B_j, q_n) = q_0$ *and* tuptost $(q_0, \gamma_j, B_j, q_n) = q_n$*).*

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 3.3.8.

HOL Theorem 3.3.8

```
ID p ⊢ dl ◁ (q,inp,stk) → (qf,[],[]) ⇒
∃dl0 q0 i0 s0 spfx.
  ID p ⊢ dl0 ◁ (q,inp,stk) → (q0,i0,s0) ∧ |s0| = |stk| − 1 ∧
  (∀q′ i′ s′. (q′,i′,s′) ∈ FRONT dl0 ⇒ |stk| ≤ |s′|) ∧
  ((∃dl1.
      ID p ⊢ dl1 ◁ (q0,i0,s0) → (qf,[],[]) ∧ |dl1| < |dl| ∧
      |dl0| < |dl|) ∨
   (q0,i0,s0) = (qf,[],[]))
```

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

The proof of HOL Theorem 3.3.8 is based on another HOL theorem that if $(q, \gamma\eta, \alpha\beta) \rightarrow^i (q', \eta, \beta)$ under some conditions then we can conclude $(q, \gamma, \alpha) \rightarrow^i (q', \epsilon, \epsilon)$. In HOL:

HOL Theorem 3.3.9

```
ID p ⊢ dl ◁ (q, ipfx ++ isfx, spfx ++ ssfx) → (q′, isfx, ssfx) ⇒
(∀ e. e ∈ FRONT dl ⇒ ∃ s. s ≠ [] ∧ pdastk e = s ++ ssfx) ⇒
p ⊢ (q, ipfx, spfx) →* (q′, [], [])
```

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.

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

$$[q_j, B_j, q_{j+1}] \overset{l}{\Rightarrow}^* \gamma_j \text{ for } 1 \leq j \leq n. \tag{3.5}$$

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

Because of the the list structure that has to be used to represent the individual (3.5) derivations, we also have to provide the following proof.

HOL Theorem 3.3.10

```
  (∀ e.
    e ∈ ℓ ⇒
    (derives g)* [NTS (tupfrmst e,tupstk e,tuptost e)]
      (MAP toTmnlSym (tupinp e))) ⇒
(derives g)* (MAP toRhs ℓ)
  (MAP toTmnlSym (FLAT (MAP tupinp ℓ)))
```

Since $(q, a\gamma, A) \rightarrow (q_1, \gamma, B_1 B_2 \ldots B_n)$, we know that

$[q, A, p] \overset{l}{\Rightarrow} a[q_1, B, q_2][q_2, B_2, q_3] \ldots [q_n, B_n, q_{n+1}]$, so finally we can conclude that $[q, A, p] \overset{l}{\Rightarrow}^* a\gamma_1\gamma_2 \ldots \gamma_n = x$.

The overall structure of the proof follows Hopcroft and Ullman but for the extent of the details. These proofs were quite involved, only a small subset of which has been shown above so as not to obscure the overall proof process.

3.3.2.2 Proof of the “only if” portion

Now we prove the other direction of (3.4). Suppose $[q, A, p] \Rightarrow^i x$. We show by induction on i that $(q, x, A) \rightarrow^* (p, \epsilon, \epsilon)$.

HOL Theorem 3.3.11

```
derives g ⊢ dl ◁ [NTS (q,A,p)] → x ∧ q ∈ states M ⇒
isWord x ⇒
pda2grammar M g ⇒
M ⊢ (q,x,[A]) →* (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, $[q, A, p] \Rightarrow a[q_1, B_1, q_2] \ldots [q_n, B_n, q_{n+1}] \Rightarrow^{i-1} x$, where $q_{n+1} = p$ or $[q, A, p] \Rightarrow [q_1, B_1, q_2] \ldots [q_n, B_n, q_{n+1}] \Rightarrow^{i-1} x$, 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 $x = ax_1x_2 \ldots x_n$, 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, $\alpha = \alpha_1\alpha_2 \ldots \alpha_n$ and $\beta = \beta_1\beta_2 \ldots \beta_n$, such that $\alpha_i \Rightarrow^i \beta_i$ in $i \leq m$ steps.

HOL Theorem 3.3.12

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

$$(q_j, x_j, B_jB_{j+1} \ldots B_n) \rightarrow^* (q_{j+1}, \epsilon, B_{j+1} \ldots B_n). \tag{3.6}$$

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

$$(q, x, A) \rightarrow (q_1, x_1x_2 \ldots x_n, B_1B_2 \ldots B_n) \tag{3.7}$$

is a legal move of M. From this move and (3.6) for $j = 1, 2, \ldots, n$, $(q, x, A) \rightarrow^* (p, \epsilon, \epsilon)$ 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 HOL 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[2]. By our induction hypothesis,

$$(q_j, x_j, B_j) \ \rightarrow^* \ (q_{j+1}, \epsilon, \epsilon). \tag{3.8}$$

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

$$(q_1, x_1 \ldots x_n, B_1 \ldots B_n) \ \rightarrow^* \ (p, \epsilon, \epsilon). \tag{3.9}$$

Another fact that needs to be asserted explicitly is reasoning for (3.9) which expressed in HOL as follows.

HOL Theorem 3.3.13

```
(∀ e₁  e₂.
   (e₁,e₂) ∈ ℓ ⇒
   m ⊢
       (frmState e₁,e₂,[transSym e₁]) →*
           (toState e₁,[],[])) ∧
(∀ e₁  e₂  p  s. MAP FST ℓ = p ++ [e₁; e₂] ++ s ⇒ adj e₁ e₂) ⇒
EVERY isNonTmnlSym (MAP FST ℓ) ⇒
∀ h₁  h₂  i₁  i₂  t.
  ℓ = (h₁,h₂)::t ∧ LAST ℓ = (i₁,i₂) ⇒
  m ⊢
      (frmState h₁,h₂ ++ FLAT (MAP SND t),
       [transSym h₁] ++ MAP transSym (MAP FST t)) →*
          (toState i₁,[],[])
```

(Here `frmState (NTS (q,A,p))` *=* q*,* `toState (NTS (q,A,p))` *=* p *and* `transSym (NTS (q,A,p))` *=* A*.)*

This is done by proving the affect of inserting input/stack symbols on the PDA transitions.

HOL Theorem 3.3.14

```
m ⊢ (q,x,stk) →* (q′,x′,stk′) ⇒
∀ℓ. m ⊢ (q,x ++ ℓ,stk) →* (q′,x′ ++ ℓ,stk′)
```

Now from the first step, (3.7) and (3.9), $(q, x, A) \ \rightarrow^* \ (p, \epsilon, \epsilon)$ follows.

Equation (3.4) 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 for the construction of G, says that $S \ \Rightarrow^* \ x$ if

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

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 3.3.15

$\texttt{pda2grammar}\ M\ g \wedge \texttt{isWord}\ x \Rightarrow$
$\texttt{((derives}\ g\texttt{)}^*\ \texttt{[NTS (startSym}\ g\texttt{)]}\ x \iff$
$\quad \exists p.\ M \vdash \texttt{(}M\texttt{.start,}x\texttt{,[}M\texttt{.ssSym])} \rightarrow^* \texttt{(}p\texttt{,[],[]))}$

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

HOL Theorem 3.3.16

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

The INFINITE 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.

3.4 Conclusions

The crux of the effort in formalising the PDA has been the proof that every pushdown automaton can be represented by a context-free grammar. The increased effort is due to the numerous facts hidden behind simple statements, discussed in Section 3.3.2. There are also logical gaps in the proof, from a theorem prover's point of view. All this implicit knowledge embedded in the proof increases the complexity when translating the proof into an automated proof. Thankfully, the remaining results, though tedious, were relatively straightforward to formalise. Our relational style of presenting definitions has definitely simplified some aspects of the proof. Unfortunately, we had to forego executability in favour of relations. An interesting exercise would be to see how the definitions and proofs scale with executable definitions, especially for deriving a CFG for a PDA where we have calculate a lot of possibilities of transitioning between states, most of which never even get used. One can obviously only compute valid combinations, but this may have an impact on the readability and possibly the runtime as well.

Similar to our work, Courant and Filliâtre have formalised some of the theory of finite automata and rational languages, context-free grammars and pushdown automata in Coq and is available at [18]. This is available as part of the Coq library though there have been no publications discussing this work. They also do not seem to have a proof for the statement that every pushdown automaton can be associated with a context-free grammar. In a similar formalisation of theory, Rival *et al.* [65] have formalised tree automata in Coq.

Table 3.1 shows the mechanisation effort. The high number of proofs for *Final state to empty stack* conversion is not because of the complexity of the process but rather its tedious nature. The definitions and the process themselves are very simple. They just required a lot of small proofs. In contrary, the number of proofs for *PDA to CFG* are not that many but are larger and more complex.

Mechanisation	LOC	#Definitions	#Proofs
Final state to empty stack	1259	4	53
Empty stack to final state	553	2	22
CFG to PDA	666	2	19
PDA to CFG	1951	14	31

Table 3.1 – Summary of the mechanisation effort for PDAs

Chapter 4

Properties of Context-Free Languages

> Jen: How can you two live like this?
> Moss: [*typing*] How can you two...
> Roy: Don't google the question, Moss!
>
> IT Crowd

Contents

In this chapter we discuss the mechanisation of the proof of the pumping lemma for context-free languages (Section 6.3.1) and the proofs for the various closure properties (Section 4.3).

We first provide a formalisation of parse trees in Section 4.1. Parse trees are used to represent derivations in a grammar. This alternative notation for expressing derivations is used to prove the pumping lemma and closure under substitution. Later on in Section 6.3, we present another attempt at the proof of the pumping lemma by using derivation lists. Parse trees are also used in mechanisation of the SLR parsing algorithm described in Chapter 5.

The closure properties covered in this chapter are as following. We show closure under union in Section 4.3.1, closure under concatenation in Section 4.3.2, closure under substitution in Section 4.3.4 and closure under inverse homomorphism in Section 4.3.5. Closure under homomorphism follows directly from closure under substitution.

Contributions

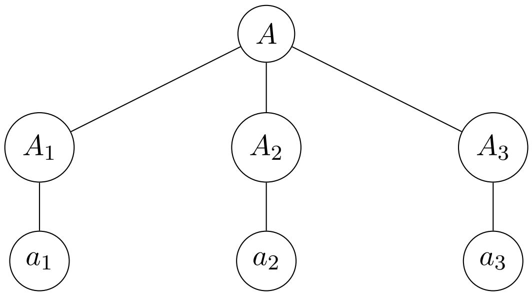
- ◇ Mechanisation of a proof of pumping lemma.
- ◇ Mechanisation of proofs for closure properties of context-free languages.

4.1 Derivation (or parse) trees

In Section 2.1 we introduced derivation lists for modeling derivations in a grammar. They have been immensely helpful in stating goals in a way such that a simple induction suffices for a proof. Unfortunately, they have one major drawback: with derivation lists each derivation step corresponds to expanding a single nonterminal. This adds to the complexity of the mechanisation, more of which is discussed in Section 6.3. 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 nonterminals. The derivation

$$A \Rightarrow A_1 A_2 A_3 \Rightarrow a_1 A_2 A_3 \Rightarrow a_1 a_2 A_3 \Rightarrow a_1 a_2 a_3$$

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



Here, nonterminal 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 (nonterminal) node which can expand to a sequence of 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 the definition used in Hopcroft and Ullman. They define a leaf node as a node with an empty (ϵ) node as its only child.

HOL Definition 4.1.1 (`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. Function `validptree` is only called on the nonterminal nodes. Thus, a leaf node on its own is not considered a valid parse tree.

HOL Definition 4.1.2 (`validptree`)

```
validptree g (Node n ptl) ⟺
  rule 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 nonterminal.)*

This is another place where 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 has to derive a word. We instead define a looser version where 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 it still has to be some nonterminal symbol. Also, 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 root nonterminal to the yield.

HOL Theorem 4.1.1

```
validptree g t ⇒ (derives g)* [root t] (MAP TS (fringe t))
```

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

HOL Theorem 4.1.2

```
derives g ⊢ dl ◁ [NTS A] → y ⇒
isWord y ⇒
∃t. validptree g t ∧ MAP TS (fringe t) = y ∧ root t = NTS A
```

(`fringe t` is a list of terminals. We use `MAP TS` to convert the bare terminals into terminal symbols.)

Theorem 4.1.1 (H&U Theorem 4.1) *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.*

The corresponding HOL theorem is:

HOL Theorem 4.1.3

```
w ∈ L g ⇔
  ∃t.
    validptree g t ∧ MAP TS (fringe t) = w ∧
    root t = NTS (startSym g)
```

Given any grammar g, one can construct a corresponding parse tree for the strings derivable from the start symbol in g *i.e.* in the language of g. Each level of the parse tree (the parse tree list) will then represent the expansions using one of the grammar rules. A valid parse tree is created using only the rules of g. Accessing a subtree (subderivation) is now simply a case of traversing down the parse tree to a particular node.

Function `subtree` returns the subtree at the end of a given path p. The path is a list of indices to the root nodes as one traverses down a parse tree.

HOL Definition 4.1.3 (subtree)

```
subtree (Leaf tm) (p::ps) = NONE
subtree (Leaf v₂) [] = SOME (Leaf v₂)
subtree (Node v₃ v₄) [] = SOME (Node v₃ v₄)
subtree (Node n ptl) (p::ps) =
  if p < |ptl| then subtree (EL p ptl) ps else NONE
```

Using the above definition we can define what it means to be a proper subtree. Tree st is a proper subtree of t if the path p is nonempty.

HOL Definition 4.1.4 (isSubtree)

```
isSubtree st t ⟺ ∃p. p ≠ [] ∧ subtree t p = SOME st
```

Similarly we can define predicate isSubtreeEq which also allows the subtree to be exactly the same as the parent tree. This is easily achieved by omitting the condition which asserts that the path to the subtree has to be nonempty ($p \neq$ []).

4.2 Pumping lemma

The pumping lemma for context-free languages is a standard, well-known result in language theory. It provides a sufficient condition for showing that a language is *not* context-free. This is done by showing that for all sufficiently large n, where $n > 0$ and n is larger than the pumping length, there is a string s in the language with length at least n which cannot be "pumped" without producing strings that are not in the language. Here we provide what we believe to be the first mechanised proof of the pumping lemma. We first provide a broad overview of the proof. This is followed by an explanation of the lemmas required for the final proof (Sections 4.2.1 to 4.2.3). We put these together to obtain the final proof of the pumping lemma in Section 4.2.4. In Section 4.2.5, we talk about a frequently occurring issue in theorem proving that increases the complexity of the mechanised proofs.

Theorem 4.2.1 (H&U Lemma 6.1) *Let L be a context-free language. Then there exists some integer $n > 0$ such that any string z in L with $|z| \geq n$ can be written as $z = uvwxy$, with substrings u, v, w, x and y, such that*

Clause 1 $|vx| \geq 1$, *either v or x has to be nonempty since they are the pieces that get "pumped";*

Clause 2 $|vwx| \leq n$, *i.e. the middle portion is not too long;*

Clause 3 uv^iwx^iy *is in L for every integer $i \geq 0$, i.e. v and x may be "pumped" zero or more times and the resulting string will still be a member of L.*

The value n is a constant for any given language, and is called the *pumping length*. Informally, any string that is at least n symbols long can be pumped to obtain even longer strings that still belong in the language. Note that finite languages (which are regular and hence context-free) obey the pumping lemma trivially by having n equal to the maximum string length in L plus one.

This property can be stated succinctly as HOL Theorem 4.2.1 given below.

HOL Theorem 4.2.1

```
isCnf g ⇒
∃n.
  n > 0 ∧
  ∀z.
    z ∈ L g ∧ |z| ≥ n ⇒
    ∃u v w x y.
      z = u ++ v ++ w ++ x ++ y ∧ |v| + |x| ≥ 1 ∧
      |v| + |w| + |x| ≤ n ∧
      ∀i.
        u ++ FLAT (lpow v i) ++ w ++ FLAT (lpow x i) ++ y ∈
        L g
```

(The `isCnf` *predicate is true of grammars in Chomsky Normal Form. The fact that all context-free languages not including ϵ can be captured by CNF grammars is presented in Section 2.5.)*

Overview of proof

The goal is to prove that for all grammars, there is an n whose value depends on the grammar such that for each terminal string derivable in this grammar, the string satisfies the given properties. We show that such an n exists; if k is the number of nonterminals in the grammar, then $n = 2^k$.

In what is to come, we will show that it is possible to find the witnesses for u, v, w, x and y. When we have these witnesses, the rest of the proof divides into three obligations corresponding to the three clauses mentioned earlier.

4.2.1 A nonterminal must recur in the derivation

We use parse trees to represent derivations in a grammar. When using parse trees the pumping lemma can be rephrased in the following manner. If t is the parse tree for a given a string which is "sufficiently" long, then there must exist two nested subtrees such that the root nonterminal for the trees is the same.

The existence of repeated nodes is given by the rootRep property.

HOL Definition 4.2.1 (rootRep)

```
rootRep st t ⟺ root st = root t ∧ isSubtree st t
```

In Figure 4.1 the expression rootRep t_0 t_1 holds. The first important property we prove is about the existence of a recurring nonterminal within a subtree.

Let k equal the number of nonterminals in the grammar.

Lemma 4.2.2 (Size of the parse tree) *If $|z| \geq 2^k$ then there must be a nonterminal which expands to itself within the derivation of z.*

We begin the mechanisation by modeling the notion of a derivation containing a repeated symbol, symRepProp:

HOL Definition 4.2.2 (symRepProp)

```
symRepProp t0 t1 t ⟺ isSubtreeEq t1 t ∧ rootRep t0 t1
```

We say that symRepProp holds over t if a subtree of t has the same root node as t. In Figure 4.1, expression symRepProp t_0 t_1 t is true.

If none of the subtrees of t have nonterminals that are repeated, the length of the final terminal string is constrained by the number of distinct nonterminals in t.

HOL Lemma 4.2.1

```
validptree g t ⇒
isCnf g ⇒
¬(∃t0 t1. symRepProp t0 t1 t) ⇒
∀k. k = |distinctNtms t| ⇒ |leaves t| ≤ 2 ** (k − 1)
```

*(The ** operator represents exponentiation. Function distinctldNtms t returns the list of nonterminals occurring in tree t.)*

Proof This property directly follows from the grammar being in CNF. The proof is by induction on the height of the tree.

Because the total number of nonterminals in the grammar is at least as large as the number occurring distinctly in any given derivation, if $|z|$ is larger than 2^k, then we know that the derivation tree of z has a repeated nonterminal symbol.

4.2.2 Isolating the last repetition of a nonterminal

The next step is to pick the very last instance when the repetition of a nonterminal occurs, *i.e.*, as close as possible to the final terminal string. This corresponds to witnessing the tree structure represented in Figure 4.1.

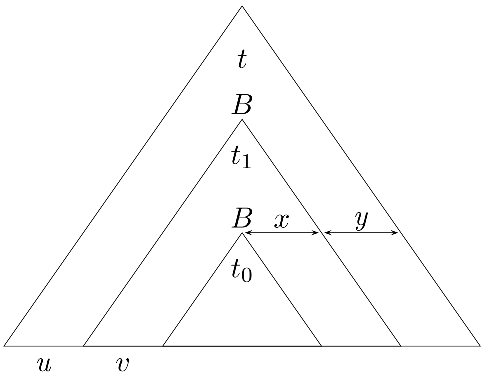


Figure 4.1 – Tree representation of property `lastExpProp`. The outermost tree t has the two nested subtrees (t_1 and t_0) with the same node B. Here the subtree t_1 may possibly be the same tree t. t_0 is a proper subtree of t_1. `symRepProp` does not hold of any proper subtree of t_1.

The property corresponding to the tree structure in HOL is `lastExpProp`.

HOL Definition 4.2.3

```
lastExpProp t ⟺
  ∃ t₀ t₁.
    symRepProp t₀ t₁ t ∧
    ∃ n ptl.
      t₁ = Node n ptl ∧
      ∀ e. e ∈ ptl ⇒ ¬∃ st₀ st₁. symRepProp st₀ st₁ e
```

We can now prove that if `symRepProp` holds of the parse tree t, then we should be able to deduce that `lastExpProp` also holds:

HOL Lemma 4.2.2

```
symRepProp t₀ t₁ t ⇒ lastExpProp t
```

Proof By induction on the height of t.

4.2.3 Deriving the relevant subtrees from the derivation

To figure out the individual components of the pumping lemma (u, v, w, x and y) we have to derive the relevant subtrees from the original derivation.

Let $z \in L(G)$. From this we can deduce the tree version of the language equivalence property.

$$\begin{array}{l}
\exists\, t.\\
\quad \mathtt{validptree}\ g\ t\ \land\ \mathtt{root}\ t\ \mathtt{=}\ \mathtt{NTS}\ \mathtt{(startSym}\ g\mathtt{)}\ \land\\
\quad z\ \mathtt{=}\ \mathtt{MAP}\ \mathtt{TS}\ \mathtt{(leaves}\ t\mathtt{)}
\end{array}$$

Since we pick z such that it is sufficiently long, *i.e.* $|z| \geq 2^{k-1}$ for k distinct nonterminals in tree t, we have a repeated nonterminal in one of the subtrees of t. Thus by HOL Lemma 4.2.1 we have

$$\exists\, t_0\ \ t_1.\ \mathtt{symRepProp}\ t_0\ \ t_1\ \ t$$

We now pick the subtree t_0, using HOL Lemma 4.2.2, under which there are no repeated nonterminals.

$$\exists\, t_0\ \ t_1.\ \mathtt{symRepProp}\ t_0\ \ t_1\ \ t$$

Using the definition of `lastExpProp` we have

$$\begin{array}{l}
\exists\, x\ \ y.\\
\quad \mathtt{symRepProp}\ x\ \ y\ \ t_0\ \land\\
\quad \exists\, n\ \ ptl.\\
\qquad y\ \mathtt{=}\ \mathtt{Node}\ n\ \ ptl\ \land\\
\qquad \forall\, e.\ e \in ptl \Rightarrow \neg\exists\, st_0\ \ st_1.\ \mathtt{symRepProp}\ st_0\ \ st_1\ \ e
\end{array}$$

Since the tree corresponds to a grammar in Chomsky Normal Form, ptl must have exactly two subtrees and they must both correspond to nonterminal nodes.

$$\exists\, N_1\ \ N_2\ \ s_1\ \ s_2.\ ptl\ \mathtt{=}\ \mathtt{[Node}\ N_1\ \ s_1\mathtt{;}\ \mathtt{Node}\ N_2\ \ s_2\mathtt{]}$$

These two nodes are responsible for the derivation of string vwx in Clause 2. We can reason about the length of the strings mentioned in the clauses based on the `symRepProp` property.

4.2.4 Putting it all together

We can now bring together the proof for the pumping lemma using the elements described above. Figure 4.2 shows what each of the quantifiers in the pumping lemma proof get instantiated to.

The figure shows the derivation of terminal string z from the start symbol S of some grammar g. B is the symbol that gets repeated in its own derivation closest to z. The

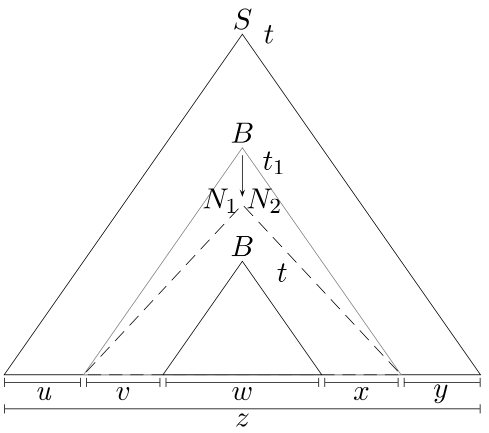


Figure 4.2 – Tree showing the witnesses for the existential quantifiers in the pumping lemma.

triangles for B show the very last occurrence of any nonterminal that expands to itself. B takes a single step to expand to the two nonterminals N_1 and N_2 which eventually expand out to the terminal string vwx. There are no repeated nonterminals in the expansion that follows from N_1N_2. The components u, v, w, x and y split up z so that they satisfy the pumping lemma clauses.

The description of the proof follows.

Clause 1 $|vx| \geq 1$.

Proof We have $B \Rightarrow^* vBt \Rightarrow^* vwx$, where v, w, x are terminal strings. Since the grammar is in CNF, it follows that either v or t must be nonempty. If v is nonempty then we are done.

From the two facts, $t \neq \epsilon$, and $t \Rightarrow^* x$, again we use the properties of CNF to deduce that x must be nonempty since CNF excludes ϵ-productions.

Clause 2 $|vwx| \leq 2^k$, where k is the number of nonterminals in the grammar.

Proof With $|z| \geq 2^k$, we use HOL Lemma 4.2.1 to get `symRepProp` t_0 t_1 t.

From HOL Lemma 4.2.2, we have `lastExpProp` t. This gives us a handle on the structure of the tree as shown in Figure 4.2.

From this structure we are able to deduce, $B \Rightarrow N_1N_2 \Rightarrow^* vwx$.

There must exist strings m_1 and m_2 such that $vwx = m_1m_2$ and $N_1 \Rightarrow^* m_1$ and $N_2 \Rightarrow^* m_2$ such that no nonterminal symbol is repeated in the derivations of m_1 and m_2. This corresponds to the area of the dashed triangle, which covers all the expansion under the top B except the one step derivation to N_1N_2.

Since no symbols are repeated, from HOL Lemma 4.2.1, we get the bound on the sizes of m_1 and m_2. Let d_1 be the number of unique nonterminals in the derivation of m_1 and d_2 be the number of unique nonterminals in the derivation of m_2. Then, $|m_1| \leq 2^{d_1-1}$ and $|m_2| \leq 2^{d_2-1}$. We also know that, $d_1 \leq k$ and $d_2 \leq k$.

Therefore $|vwx| \leq 2^k$.

Clause 3 uv^iwx^iy is in L for every natural number i.

Proof By the application of HOL Lemma 4.2.2 we have the shape of the tree as shown in Figure 4.2, and the following derivations: $S \Rightarrow^* uBy$, $B \Rightarrow^* vwx$, and $B \Rightarrow^* vBx$.

The applications of the above expansions give us the following two possibilities to satisfying this clause.

- ◇ $S \Rightarrow^* uBy \Rightarrow^* uwy$, for $i = 0$.
- ◇ $S \Rightarrow^* uBy \Rightarrow^* uvBxy \cdots \Rightarrow^* uv^iwx^iy$, for $i \geq 1$.

4.2.5 Filling in the gaps

Throughout the mechanisation, there has been one constant concern. The process of mechanising a proof in HOL never seems to follow as simple a pattern as described in the textbooks. Even though formal in nature, the textbook proof steps are very coarse-grained when compared to what needs to be covered in the theorem prover. This is a well-known cause of both the complexity and the tediousness of mechanised proofs.

As an example, consider the following property, essential in establishing Clause 3 of the pumping lemma. Let $B \overset{l}{\Rightarrow}^* pBs$, $B \overset{l}{\Rightarrow}^* z$ and $s \overset{l}{\Rightarrow}^* z'$, where p, z and z' consist of only terminals. It is easily deduced that $B \overset{l}{\Rightarrow}^* p^i z\, z'^i$. This lemma is quite reasonably considered too trivial to require any proof in a textbook presentation. On the other hand, mechanisation requires this fact to be proven explicitly. It is stated in HOL as follows.

HOL Lemma 4.2.3

```
(lderives g)* [NTS B] (p ++ [NTS B] ++ s) ∧ isWord p ∧
(lderives g)* [NTS B] z ∧ isWord z ∧ (lderives g)* s z′ ∧
isWord z′ ⇒
(lderives g)* [NTS B]
  (FLAT (lpow p i) ++ z ++ FLAT (lpow z′ i))
```

Function `lpow p n` ***returns*** p^i.

Most of the work of automating a theory goes into recognising and proving such gap proofs: HOL proofs that fill in the gaps in the textbook. It is quite hard to anticipate how many gap proofs will be needed at the outset. To add to the issue, these proofs may in turn require further properties that also have to be stated explicitly.

The proof of the above example is done by induction on i. Among other sub-proofs, we end up proving the following (even more trivial) three properties about replicating lists.

Property 1 If a list p is replicated $i+1$ times, then it is the same as a copy of p followed by i copies of p. Here, `SUC` n returns $n+1$.

```
FLAT (lpow p (SUC i)) = p ++ FLAT (lpow p i)
```

Property 2 Adding a copy of p to either side of i copies of p results in the same list.

```
p ++ FLAT (lpow p i) = FLAT (lpow p i) ++ p
```

Property 3 Replicating a list of terminals results in a string of terminals.

```
isWord ℓ ⇒ isWord (FLAT (lpow ℓ i))
```

After establishing our basic framework, mechanising a proof rapidly brings us to a state where most of the work goes beyond what a textbook covers. Each problem area tackled has its own unique challenges, which mostly have to be covered on a case-by-case basis. Using existing libraries is usually advantageous, but can also be a burden if, as in the choice of derivation lists, the existing library is too well-developed to ignore, but not quite a good fit for the task at hand.

4.3 Closure properties

In their Chapter 6, Hopcroft and Ullman prove that context-free languages are closed under the following operations. That is, if L and P are context-free languages, the following languages are context-free as well:

- the union $L \cup P$ of L and P
- the concatenation $L \circ P$ of L and P
- the Kleene star L^* of L
- language obtained by substitution operation
- the image $\varphi(L)$ of L under a homomorphism φ

In this section we go through the HOL formalisation for proving closure of CFGs under union, concatenation, Kleene star operation, substitution and inverse homomorphism. 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. The only additional theorem we had to establish is the following:

HOL Theorem 4.3.1

```
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. Of course, this is intuitively clear. The closure properties that follow are based on this very assumption. Hence, to be able to proceed with the remainder of the work we had to prove this *disjoint* property in HOL. Note that since we are renaming variables by picking new ones, we need the premise that the type universe of the variables is infinite.

Closure properties 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 $S \to S_1|S_2$. 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 nonterminals of G_1 and G_2 do not overlap. Hence, the need for the disjoint property.

Formally, for the disjoint property we show:

Theorem 4.3.1 *For any grammar $G = (V, T, P, S)$, we can find a new grammar $G' = (V', T, P, S')$ such that $L(G) = L(G')$ and $V \cap V' = \phi$.*

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 4.3.1 (`rename`)

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

Using `rename`, we can rename the nonterminal nt to nt' for a particular rule.

HOL Definition 4.3.2 (`ruleNt2Nt′`)

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

Now given a new replacement value (nt') for a nonterminal nt, we systematically rename all nts to nt' in our old grammar $G\ p\ s$. Note that we need to rename the start symbol as well. This is our function `grNt2Nt′`.

HOL Definition 4.3.3 (`grNt2Nt′`)

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

We can now define the notion of renaming a *single* nonterminal in the grammar. Relation `rename1_R` holds if and only if renaming nonterminal nt to nt' (a new nonterminal, not in g) in grammar g gives us grammar g'.

HOL Definition 4.3.4 (`rename1_R`)

```
rename1_R (nt,nt′) g g′ ⇔
  NTS nt′ ∉ nonTerminals g ∧ grNt2Nt′ nt nt′ g = g′
```

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

HOL Theorem 4.3.2

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

In order to get a new grammar g' starting from the old grammar g such that the nonterminals are disjoint, all we need to do is rename all the nonterminals in g such that the new names introduced are not part of g. This is achieved by taking the reflexive, transitive closure of the relation $\lambda x\ y.\ \exists nt\ nt'.$ `rename1_R` $(nt, nt')\ x\ y$. We show that such a grammar (g'), which does not have any overlapping nonterminals with g and the language of g' is the same as of g, exists.

HOL Theorem 4.3.3

```
FINITE s ⇒
∀g g″.
  s = nonTerminals g ∩ nonTerminals g″ ⇒
  INFINITE 𝒰(:α) ⇒
  ∃g′.
    (λx y. ∃nt nt′. rename1_R (nt,nt′) x y)* g g′ ∧
    L g = L g′ ∧
    DISJOINT (nonTerminals g′) (nonTerminals g″)
```

The proof of the theorem is by induction on the finiteness of the set s. When the above theorem is invoked as part of proving the disjoint property, both g and g'' variables get instantiated with the original grammar for which the variables are being renamed.

After establishing this property we can now start having a look at the closure properties.

4.3.1 Union

Theorem 4.3.2 (H&U Theorem 6.1) *Context-free languages are closed under union.*

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 (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 $S_3 \rightarrow S_1 | S_2$. Given grammars G_1 and G_2, function `grUnion` constructs such a grammar G_3.

HOL Definition 4.3.5 (`grUnion`)

```
grUnion s0 g1 g2 =
  G
    (rules g1 ++ rules g2 ++ [rule s0 [NTS (startSym g1)]] ++
     [rule 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 corresponding statement for Theorem 4.3.2 in HOL is:

HOL Theorem 4.3.4

```
DISJOINT (nonTerminals g1) (nonTerminals g2) ∧
INFINITE 𝒰(:α) ⇒
∃ s0.
  NTS s0 ∉ nonTerminals g1 ∪ nonTerminals g2 ∧
  (w ∈ L g1 ∪ L g2 ⟺ w ∈ L (grUnion s0 g1 g2))
```

4.3.2 Concatenation

Theorem 4.3.3 (H&U Theorem 6.1) *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 $S_3 \to S_1S_2$.

In HOL this is expressed using function `grConcat`.

HOL Definition 4.3.6 (`grConcat`)

```
grConcat $s_0$ $g_1$ $g_2$ =
  G
    (rules $g_1$ ++ rules $g_2$ ++
     [rule $s_0$ [NTS (startSym $g_1$); NTS (startSym $g_2$)]]) $s_0$
```

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 4.3.3 in HOL is:

HOL Theorem 4.3.5

```
INFINITE $\mathcal{U}$(:$\alpha$) $\wedge$
DISJOINT (nonTerminals $g_1$) (nonTerminals $g_2$) $\Rightarrow$
$\exists$ $s_0$.
  NTS $s_0$ $\notin$ nonTerminals $g_1$ $\cup$ nonTerminals $g_2$ $\wedge$
  ($w$ $\in$ conc ($L$ $g_1$) ($L$ $g_2$) $\iff$ $w$ $\in$ $L$ (grConcat $s_0$ $g_1$ $g_2$))
```

(Element s is in `conc` *as bs if and only if there exists $u \in as$ and $v \in as$ such that $s = u$ ++ v.)*

4.3.3 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 4.3.4 (H&U Theorem 6.1) *Context free languages are closed under Kleene closure.*

In HOL, the `star` relation represents the Kleene closure.

HOL Definition 4.3.7 (`star`)

```
star A []
s ∈ A ⇒ star A s
s₁ ∈ A ∧ star A s₂ ⇒ star A (s₁ ++ s₂)
```

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 $S_1 \to SS_1$ and $S_1 \to \epsilon$.

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

HOL Definition 4.3.8 (`grClosure`)

```
grClosure s₀ g =
  G
    (rules g ++ [rule s₀ [NTS (startSym g); NTS s₀]] ++
     [rule s₀ []]) s₀
```

A proof that $L(G)^* = L(G_1)$ follows a similar methodology as used in proofs above. Theorem 4.3.4 translates into HOL as following:

HOL Theorem 4.3.6

```
INFINITE 𝒰(:α) ⇒
∃s₀.
  NTS s₀ ∉ nonTerminals g ∧ star (L g) w ⇐⇒
    w ∈ L (grClosure s₀ g)
```

4.3.4 Substitution

A more interesting closure proof is that of the substitution operation.

Theorem 4.3.5 (H&U Theorem 6.2) *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 $A \rightarrow \alpha$ 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 $L(G^{'}) = \mathit{replace}\ a\ w_a\ s_G$, where `replace` substitutes the word w_a for terminal a in sentence s_G, $w_a \in L(G_a)$ and $s_G \in (V \cup T)^*$. Figure 4.3 shows derivations in G'.

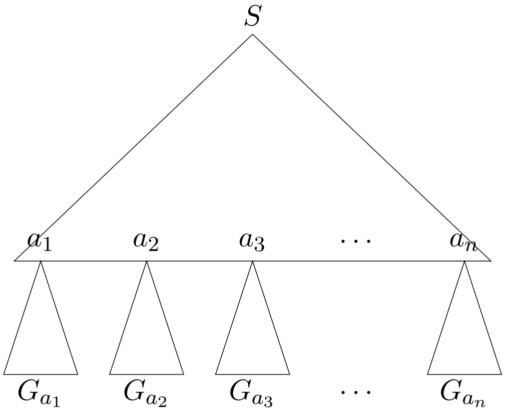


Figure 4.3 – Given grammar G with start symbol S and derivation $S \Rightarrow^* a_1a_2 \ldots a_n$, terminals $a_1, a_2, \ldots, a_n$ are substituted by words from grammars $G_{a_1}, G_{a_1}, \ldots, G_{a_n}$.

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 nonterminals in g and $gsub$ are disjoint.

HOL Definition 4.3.9 (`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 4.3.10 (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 4.3.7

```
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 4.3.8

```
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 4.3.9

```
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 4.3.10

```
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.

The closure under union, concatenation and Kleene operation can also be shown as a corollary of closure under substitution. This is done by setting up a proper substitution function. As an example let us have a quick look at the union property. Let L_1 and L_2 be two context-free languages. We can define $s(L)$ such that L is the language $\{1, 2\}$ and s is the substitution defined by $s(1) = L_1$ and $s(2) = L_2$. We can provide substitutions for proofs of closure under concatenation and Kleene operation in a similar manner.

Another property that follows from closure under substitution is that of homomorphism. A string *homomorphism* is a function on strings that works by substituting a particular string for each symbol. Formally, for homomorphism h on alphabet Σ, and $w = a_1a_2 \ldots a_n$ is a string of symbols Σ, then $h(w) = h(a_1)h(a_2) \ldots h(a_n)$. We can also apply a homomorphism to a language by applying to each of the strings in the language. For language L over alphabet Σ and homomorphism h on Σ, $h(L) = \{h(w)|w \in L\}$

Corollary 4.3.6 (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.*

4.3.5 Inverse homomorphism

Homomorphisms can also be applied in reverse, *i.e.* taking the inverse of the homomorphism function h. This inverse application is also closed for context-free languages. The inverse homomorphism of h is written as $h^{-1}(L)$ and is the set of strings w in Σ^* such that $h(w)$ is in L.

Theorem 4.3.7 (H&U Theorem 6.2) *Context-free languages are closed under inverse homomorphism.*

The proof for closure under inverse homomorphism uses pushdown automata. Let $h : \Sigma \rightarrow \delta$ be a homomorphism and L be a CFL. Let $L = L(M)$, where M is the PDA $(Q, \delta, \Gamma, \delta, q_0, Z_0, F)$.

The construction of PDA M' that accepts $h^{-1}(L)$ is as follows. Let a be a single element of the alphabet Σ. On input a, M' generates $h(a)$ and simulates M on $h(a)$. If M'

were a finite automaton M' could simulate such a composite move in one of its moves. However for a nondeterministic PDA M, M could pop many symbols, or make moves that push an arbitrary number of symbols on the stack. Thus M' cannot necessarily simulate M's moves on $h(a)$ with one (or any finite number of) moves of its own.

To handle this, we give M' a buffer, in which it may store $h(a)$. Then M' may simulate any ϵ moves of M it likes and consume the symbols of $h(a)$ one at a time, as if they were M's input. As the buffer is part of M's finite control, it cannot be allowed to grow arbitrarily long. We ensure this by permitting M' to read an input symbol only when the buffer is empty. Thus the buffer holds a suffix of $h(a)$ for some a at all times. M' accepts its input w if the buffer is empty and M is in a final state. That is, M has accepted $h(w)$. Thus L(M') = $\{w \mid h(w)$ is in $L\}$, that is $L(M') = h^{-1}(L(M))$.

Formally, let $M' = (Q', \Sigma, \Gamma, \delta', [q_0, \epsilon], Z_0, F \times \epsilon)$, where Q' consists of pairs $[q, x]$ such that q in Q and x is a (not necessarily proper) suffix of some $h(a)$ for a in Σ.

δ' is defined as follows.

Rule 1 $\delta'([q, x], \epsilon, Y)$ contains all (p, γ) such that $\delta(q, \epsilon, Y)$ contains (p, γ). This rule simulates ϵ-moves of M independent of the buffer contents.

Rule 2 $\delta'([q, ax], \epsilon, Y)$ contains all $([p, x], \gamma)$ such that $\delta(q, a, \gamma)$ contains (p, γ). This rule simulates moves of M on input a in Δ and removes a from the front of the buffer.

Rule 3 $\delta'([q, \epsilon], a, Y)$ contains $([q, h(a)], Y)$ for all a in Σ and Y in Γ. This move loads the buffer with $h(a)$ reading a from M''s input.

We model the construction of M' as a relation. Relation `hInvpda` M M' h holds if and only if PDA M' simulates the inverse of homomorphic function `h`. PDA M' starts off in a new start state (`(`q_0`, [])`) with a new stack symbol (z_0). The states of M' have an associated buffer.

HOL Definition 4.3.11 (`hInvpda`)

```
hInvpda m m' h ⟺
  ∃z₀.
    z₀ ∉ stkSyms m ∧ m'.ssSym = z₀ ∧
    ∃q₀.
      q₀ ∉ states m ∧ m'.start = (q₀,[]) ∧
      (∀q r. (q,r) ∈ m'.final ⟺ q ∈ m.final ∧ r = []) ∧
      ∀r.
        r ∈ m'.next ⟺
          r ∈
          rule1 m h ∪ rule2 m h ∪ rule3 m h m'.ssSym ∪
          {((NONE,m'.ssSym,m'.start),(m.start,[]),
            [m.ssSym; m'.ssSym])}
```

In HOL, rule1, rule2 and rule3 correspond to the three different ways (described above) of constructing the transition rules of the machine accepting the inverse of the homomorphic function.

Function rule1 simulates ϵ-moves of M independent of the buffer content. Function rule1 takes a PDA m and a homomorphism h and returns transitions of the form ((NONE, $ssym$, q, x), (p, x), $ssyms$). Thus, starting in state (q, x) on reading input $ssym$ the PDA transitions to state (p, x) and the stack gets increased with $ssyms$ symbols. Here the buffer symbols x are a suffix of $h\ \ a$ for some a and a corresponding transition (((NONE, $ssym$, q), p, $ssyms$)) to the one above belongs in the PDA m.

HOL Definition 4.3.12 (rule1)

```
rule1 m h =
  {((NONE,ssym,q,x),(p,x),ssyms) |
   ∃a. isSuffix x (h a) ∧ ((NONE,ssym,q),p,ssyms) ∈ m.next}
```

Similarly, we can define rule2 and rule3. rule2 simulate moves of M on input a in δ, removing a from the front of the buffer

HOL Definition 4.3.13 (rule2)

```
rule2 m h =
  {((NONE,ssym,q,isym::x),(p,x),ssyms) |
   ∃a.
     isSuffix (isym::x) (h a) ∧
     ((SOME isym,ssym,q),p,ssyms) ∈ m.next}
```

rule3 loads the buffer with $h(a)$, reading a from M's input; the state and stack of M remain unchanged.

HOL Definition 4.3.14 (rule3)

```
rule3 m h newssym =
  {((SOME a,ssym,q,[]),(q,h a),[ssym]) |
   (a,ssym,q) |
   q ∈ states m ∧ ssym ∈ stkSyms m ∪ {newssym} ∧
   h a ∈ IMAGE h {a}}
```

To show that $L(M') = h^{-1}(L(M))$ we first show that $s \in L(M') \Rightarrow s \in h^{-1}(L(M))$, for some word s. This amounts to proving the following theorem in HOL.

HOL Theorem 4.3.11

```
ID m ⊢ dl ◁ (q,FLAT (MAP h x),ssyms) → (p,[],ssyms′) ∧
hInvpda m m′ h ∧ |dl| > 1 ∧ stkSymsInPda m′ ssyms ∧
q ∈ states m ⇒
```

$m' \vdash$
```
    ((q,[]),x,ssyms ++ [m'.ssSym]) →*
        ((p,[]),[],ssyms' ++ [m'.ssSym])
```

(Here `FLAT (MAP h x)` *gives the words in* $h^{-1}(L(M))$*.)*

By one application of Rule 3, followed by applications of Rules 1 and 2, if $(q, h(a), \alpha) \vdash^*_M (p, \epsilon, \beta)$, then

$$([q, \epsilon], a, \alpha) \vdash_M ([q, h(a)], \epsilon, \alpha) \vdash^*_M ([p, \epsilon], \epsilon, \beta).$$

If M accepts $h(w)$ we have

$(q_0, h(w), Z_0) \vdash^*_M (p, \epsilon, \beta)$, for some p in F and β in $\Gamma *$.

From this we can derive,

$$([q_0, \epsilon], w, Z_0) \vdash^*_M ([p, \epsilon], \epsilon, \beta).$$

So M' accepts w (HOL Theorem 4.3.11). Thus $L(M') \supseteq h^{-1}(L(M))$.

HOL Theorem 4.3.12
```
ID m' ⊢ dl ◁
   ((q,[]),x,pfx ++ [m'.ssSym])
     → ((p,[]),[],ssyms' ++ [m'.ssSym]) ∧
stkSymsInPda m pfx ∧ q ∈ states m ∧ hInvpda m m' h ⇒
∃dl'. ID m ⊢ dl' ◁ (q,FLAT (MAP h x),pfx) → (p,[],ssyms')
```

Conversely, we show that $s \in h^{-1}(L(M)) \Rightarrow s \in L(M')$, for some string s.

Following Hopcroft and Ullman, suppose M' accepts $w = a_1a_2...a_n$. Since Rule 3 can be applied only with the buffer empty, the sequence of the moves of M' leading to acceptance can be written as

$$\begin{aligned}
&([q_0, \epsilon], a_1a_2 \ldots a_n, Z_0) \vdash^*_{M'} ([p_1, \epsilon], a_1a_2...a_n, \alpha_1),\\
&\vdash_{M'} ([p_1, h(a_1)], a_2a_3 \ldots a_n, \alpha_1),\\
&\vdash^*_{M'} ([p_2, \epsilon], a_2a_3 \ldots a_n, \alpha_2),\\
&\vdash_{M'} ([p_1, h(a_2)], a_3a_4 \ldots a_n, \alpha_2),\\
&\vdots\\
&\vdash^*_{M'} ([p_{n-1}, \epsilon], a_n, \alpha_n),\\
&\vdash_{M'} ([p_{n_1}, h(a_n)], \epsilon, \alpha_n),\\
&\vdash^*_{M'} ([p_n, \epsilon], \epsilon, \alpha_n),
\end{aligned}$$

where p_n is in F. The transitions from state $[p_i, \epsilon]$ to $[p_i, h(a_i)]$ are by Rule 3, the other transitions are by Rule 1 and Rule 2. Thus, $(q_0, \epsilon, Z_0) \vdash^*_M (p_1, \epsilon, \alpha)$, and for all i,

$$(p_i, h(a_i), \alpha_i) \vdash^*_M (p_{i+1}, \epsilon, \alpha_{i+1}).$$

From these moves, we have

$$(q0, h(a_1a_2...a_n), Z_0) \vdash_M^* (p_n, \epsilon, \alpha_{n+1}).$$

Therefore $h(a_1a_2...a_n)$ is in $L(M)$ and $L(M') \subseteq h^{-1}(L(M))$ (HOL Theorem 4.3.12).

Thus, $L(M') = h^{-1}(L(M))$, *i.e.* in HOL:

HOL Theorem 4.3.13

```
hInvpda m m′ h ⇒
(x ∈ lafs m′ ⟺ x ∈ {w | FLAT (MAP h w) ∈ lafs m})
```

Apart from a couple of minor additions, the mechanisation of closure under inverse homomorphism follows Hopcroft and Ullman quite closely.

The first additional predicate is the property `stkSymsInPda` which forms a part of the premise of most of the proofs.

HOL Definition 4.3.15 (`stkSymsInPda`)

```
stkSymsInPda m ssyms ⟺ ∀e. e ∈ ssyms ⇒ e ∈ stkSyms m
```

When establishing the correspondence of derivations in PDAs m and m', the symbols on the stack have to be valid for both m and m'. Invariant `stkSymsInPda` ensures that the symbols in PDA m are also in PDA m' and vice versa.

The second additional predicate is the property of the form q ∈ `states` m which explicitly states that the start state is valid for the given PDA.

Both these properties are derived from the premise of the final proof goal (HOL Theorem 4.3.13) and therefore only affect the individual statements for the "if" (HOL Theorem 4.3.11) and "only if" (HOL Theorem 4.3.12) direction.

4.4 Conclusions

The work presented in this thesis is on mechanising classical proofs in language theory, those relating to context-free languages. Mechanisation in this area is scarce. As mentioned throughout, proofs of some well-known results do exist, but they are rare and usually presented as an example case or as part of a more application-oriented project.

We have tried to come up with a body of work that presents a coherent library of results pertaining to context-free languages. This is along the lines of work by Courant and Filliâtre [18]. They have formalised some theory of regular and context-free languages.

In light of the work presented in this chapter, they have also mechanised the result that CFLs are closed under union. We are not aware of any other existing formalisation for either pumping lemma or any of the remaining closure properties covered in this chapter.

As previously mentioned, we initially decided to use lists instead of trees to represent derivations in the mechanisation of pumping lemma. This naturally led to the proofs looking rather different to the way they appear in Hopcroft and Ullman. The top-level proof statement is the same and its proof is fairly readable. However, the complexity of the intermediate proofs, in terms of their size, turned out to be much greater than expected. We re-implemented pumping lemma using trees (see Section 6.3) which turned out to be a better choice. This highlights the inherent problem when automating a proof. At the outset, it can be hard to anticipate the scale of a problem. Decisions such as choice of data structures, and the form of definitions (relations *vs.* functions, for example) have a huge impact on the size of the proof, as well as ease of automation. As with the examples in Section 4.2.5, these gaps include both the deductive steps that get omitted in a textbook proof, and also the intermediate results needed because of the particular mechanisation technique.

Table 4.1 shows the proof effort for the mechanisation covered in this chapter.

Mechanisation	LOC	#Definitions	#Proofs
Pumping lemma	2500	2	26
Disjoint-ness	448	5	29
Union	278	1	15
Concatenation	169	1	7
Kleene closure	218	1	11
Substitution	539	4	29
Inverse homomorphism	1706	5	46

Table 4.1 – Summary of the mechanisation effort for properties of CFLs

The above numbers exclude library proofs and the general framework for CFGs. Proof of closure under inverse homomorphism requires the definition of the algorithm for the construction of the pushdown automata and hence is more involved. Similarly compared to other closure properties, showing closure under substitution involves reasoning with the parse trees. What is interesting is the fact that the easily assumed disjoint-ness property turned out to be not so trivial when formalised.

Chapter

5

SLR Parsing

This chip is correct?
Well no, but it's verified
Which means what? (you know)

Evan Cohn

Contents

A Simple LR (SLR) parser reads input from Left to right and produces a Rightmost derivation. It is a bottom-up parser. It traces the derivation of a string by deducing the productions in the reverse order to the one used when the string is produced from the start symbol of the given grammar.

In this chapter we describe the verification of the SLR parser generating process: the construction of a provably correct SLR automaton[6]. Among the various properties proved, we show in particular, *"soundness"*: if the parser results in a parse tree on a given input, then the parse tree is valid with respect to the grammar, and the leaves of the parse tree match the input; *"completeness"*: if the input is in the language of the grammar, then the parser constructs the correct parse tree for the input with respect to the grammar; and *non-ambiguity*: SLR grammars are unambiguous.

In addition, we develop a version of the parser generator algorithm that is executable by automatic translation from HOL to SML. This alternative version of the algorithm requires some interesting termination proofs.

The (context-free) parsing problem is one of determining whether or not a string of terminal symbols belongs to a language that has been specified by means of a context-free grammar. In addition, we imagine that the input is to be processed by some later form of analysis, *e.g.* a compiler. Therefore, we wish to generate the parse tree that demonstrates this membership when the string is in the language, rather than just a yes/no verdict.

The parsing problem can be solved in a general way for large classes of grammars through the construction of deterministic pushdown automata. Given any grammar in the acceptable class, the application of one function produces an automaton embodying the grammar. This automaton then analyses its input, producing an appropriate verdict. The particular function we have chosen to formally characterise and verify produces what is known as an SLR automaton.

Thus, at a high level, our task is to specify and verify two functions in HOL:

```
slrmac : grammar -> automaton option
parser : automaton -> token list -> stack -> ptree option
```

The `slrmac` function returns `SOME` m if the grammar is in the SLR class, and `NONE` otherwise. The `parse` function uses the machine m to consume the input and produce a parse tree for the input string and return `NONE` in case of a failure.

In the rest of this chapter, we will describe the types and functions that appear above. We first provide a brief overview of the parsing algorithm in Section 5.1. This establishes the context for the formalisation to come. In Section 5.2 we describe the important properties of context-free grammars that will be used later on, such as the "follow set" of a sentential form. In Section 5.3.1 we describe the type of SLR automata, and the type of their results. In Section 5.3.2.1 we describe the construction of automata from input grammars. We are then in a position to verify important properties about these functions. Our theorems are presented in Section 5.4.

Finally, we also wish to be able to turn our verified HOL functions into functions that can be executed in SML. To do this, a number of definitions that have rather abstract or "semantic" characterisations need to be shown to have executable equivalents. Executable versions of functions like `followSet` don't use, for example, set comprehension or existential quantifiers. The derivation of executable forms is described in Section 5.5.

Parsers as External Proof Oracles If an external, potentially untrusted, tool were to generate the parse tree for a given word, it would be easy to verify that this parse tree was indeed valid for the given grammar. The parse tree would serve as a proof that the input was indeed in the grammar's language, and the trusted infrastructure need only check that proof. It is natural then to ask what additional value a verified parser-generator might provide. Apart from the intellectual interest in mechanising interesting mathematics, we believe there is at least one pragmatic benefit: if the (verified) construction of an SLR automaton succeeds, one has a proof that the grammar in question is unambiguous. When a parse is produced, one also knows that no other parse is possible.

Contributions

◇ A verified SLR parser generator in HOL.

◇ An executable version of the parser generator in SML.

5.1 Overview of SLR parsing algorithm

As mentioned previously, there are three important parameters to the parsing function `parse`: an automaton, a stack and the input. Before we construct the parser generator, we first we need to understand how these three fit together in the parsing process. The parser reads the input and takes some action based on the automaton and the input. The automata consists of states which are reachable based on the particular sequence of input symbols. As the input is parsed, a corresponding parse tree is constructed. The stack is used to keep track of the *parse history*: input that has been parsed, the state in the automata and the parse tree for the parse of each input symbol.

Initially, the parser starts off with a state, stack and input symbols to be parsed. The first token of the input along with the state and top of the stack is used to decide what action has to be taken. The parser can perform two kinds of actions.

- ◇ **Shift** move the first input token to the top of the stack
- ◇ **Reduce** choose a grammar rule, *e.g.* $X \to ABC$; pop C, B, A from the top of the stack; push X onto the stack.

An SLR parser is also referred to as an LR(1) parser. The '1' in LR(1) refers to the fact that the parser looks ahead one symbol before deciding on the action to be performed. The reduction in the above description takes place only if the next symbol (s) on the input stream can follow X in a derivation of *some* word in the given grammar. In such a case, we say that s is in the *follow set* of X.

For both actions, the parser transitions to a new state. Each state reflects which rules in the grammar can still be used to parse the remaining input string. Algorithm 5 shows the major steps during parsing.

```
input  : Input symbols $syms$, Automaton $m$
output: parse tree for $syms$ if $syms \in L(G)$; otherwise invalid
begin
    Initialize the stack with start symbol $S$
    $newState = initialState_g$
    Read next input symbol, $sym$
    Note the input symbol after $sym$, $sym'$
    while $syms$ is nonempty do
        if Action($m$, top(stack), $sym$) = Shift then
            $newState \leftarrow$ Goto($m$, top(stack), $sym$)
            push $sym$
            Read next symbol
        else if Action($m$, top(stack), $sym$, $sym'$) = Reduce ($p$) then
            pop $|RHS|$ of production $p$ from stack
            $newState \leftarrow$ Goto($m$, top(stack), $LHS_p$)
            push $LHS_p$
        else if $syms$ is empty and top(stack) = $ then
            output parse tree, return
        else
            output invalid, return
```

Algorithm 5: SLR parsing algorithm.

Initially, the stack is empty and the parser is at the beginning of the input. The parser knows when to shift or reduce by using a deterministic finite automaton. The edges of the DFA are labeled by the symbols (terminals or nonterminals).

We will use the following simple grammar to explain the construction of the DFA.

$(1)\ E \rightarrow E1$
$(2)\ E \rightarrow 1$

The DFA is developed in the following manner. We first augment the grammar with a new start symbol and the end of file (or end of input) marker ($). The augmented grammar is:

$(0)\ S \rightarrow E\$$
$(1)\ E \rightarrow E1$
$(2)\ E \rightarrow 1$

The action of shifting the end of file marker $ is called *accepting* and causes the parser to stop successfully. Of course, there should be no remaining input symbols.

The creation of states and the corresponding transitions depends on the notion of an *item*. An item is a grammar rule combined with the dot (•) that indicates a position in the right-hand side of the rule. A set of items defines a state of our DFA. This is done as follows. Find the initial item set using the new start symbol. This includes all the rules satisfying the following condition. If there is an item of the form $A \rightarrow v{\bullet}Bw$ in an item set and in the grammar there is a rule of the form $B \rightarrow w'$ then the item $B \rightarrow \bullet\ w'$ should also be in the item set. This is called the *closure* of the item set.

In our example, doing this gives us our initial item set:

$S \rightarrow \bullet\ E\$$
$+E \rightarrow \bullet\ E1$
$+E \rightarrow \bullet\ 1$

The items with the + before them indicate that they were a result of taking the closure of the original items. The next step involves finding all the possible states one can get to from the initial state. Take the set, Z, of all items in the initial item set where there is a dot in front of some symbol x. For each item in Z, move the dot to the right of x. Close the resulting set of items. Figure 5.1 shows the DFA for the example grammar.

Note that the closure does not add new items in all cases. We continue this process until no more new item sets are found. This automaton can be used to deduce what action needs to be taken when an input symbol is read.

Figure 5.2 shows how the automaton described in Figure 5.1 is used to parse the input string 11$. The stack column contains pairs of state and input symbol. Elements to the stack are added at the front with ':' acting as the separator. The action column shows the reduction rule and the lookahead symbol when the reduce action takes place. At the start, the stack contains the initial state and the end of file marker on it. The input string from the un-augmented grammar gets accepted if the final state `F` is reached and

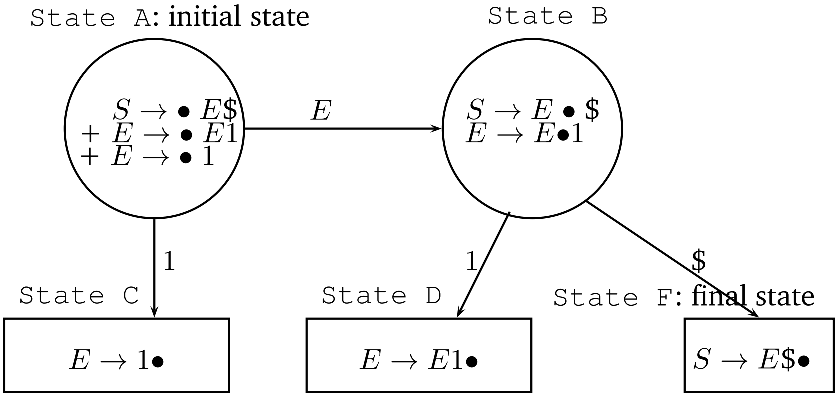


Figure 5.1 – DFA for example grammar G. Rectangles represent states where reduction occurs.

the only remaining input symbol is the end of file marker, *i.e.* \$.

Input	Stack	Action
11\$	(A,\$)	Start parse
1\$	(C,1):(A,\$)	Shift
1\$	(B,E):(A,\$)	Reduce using $E \to 1$ with lookahead 1
\$	(D,1):(B,E):(A,\$)	Shift
\$	(B,E):(A,\$)	Reduce using $E \to E1$ with lookahead \$
\$	(F,S)	Reduce using $S \to E\$$ with lookahead \$

Figure 5.2 – Parse of input $11\$$ using the DFA described earlier.

We now move on to formalising this theory in HOL.

5.2 Background mechanisation

We define the concept of nullability and functions for finding first sets and follow sets for a symbol as stated below. A list of symbols α is `nullable` if and only if $\alpha \Rightarrow^* \epsilon$.

HOL Definition 5.2.1 (`nullable`)

$\texttt{nullable}\ g\ sl \iff (\texttt{derives}\ g)^*\ sl\ \texttt{[]}$

The firstSet is a set of terminals that appear first in all sentential forms derivable from a symbol.

HOL Definition 5.2.2 (firstSet)

```
firstSet g ℓ =
  {TS fst | ∃rst. (derives g)* ℓ ([TS fst] ++ rst)}
```

The definition of followSet is more complicated. The followSet of a symbol *sym* is the set of terminals that can occur after *sym* in a sentential form derivable from any of the right-hand sides belonging to a rule in the grammar. This definition might be simplified by only considering derivations from the start symbol of the grammar. However, we choose to present it in the above way so it is compatible with our executable definition, which ignores reachability of nonterminals.

HOL Definition 5.2.3 (followSet)

```
followSet g sym =
  {TS ts |
   ∃s.
     s ∈ MAP ruleRhs (rules g) ∧
     ∃pfx sfx.
       (derives g)* s (pfx ++ [sym] ++ [TS ts] ++ sfx)}
```

The above notions are central when the actions for the SLR automaton are calculated (see Section 5.3.1). Executable versions of these functions (which do not need to scan all possible derivations) are described in Section 6.1.

5.3 SLR parser generator

We first describe the construction of the SLR automaton in HOL. We then describe the construction of the parser generator. For a detailed description of types refer to Section 2.1.

5.3.1 SLR automaton

An SLR machine is a pushdown automaton where each state corresponds to a set of *items*. An item $N \to \alpha \bullet b\beta$, is a grammar rule that has been split in two by the dot (•) marking the progress that has been made in recognising the given right-hand side ($\alpha\beta$). In HOL:

```
('nts, 'ts) item
  = item of 'nts =>
            ('nts, 'ts) symbol list × ('nts, 'ts) symbol list
```

We represent the item set using lists which makes the set finite by default since lists in HOL are finite. We use the state containing no items to represent an error state.

Based on the next symbol in the input, and the state the parser is in, the parser will perform one of the following actions:

- ◇ `REDUCE` the parser recognises a valid right-hand side on the stack and reduces it to the left-hand side of the rule
- ◇ `GOTO` the parser shifts an input symbol on to the stack and goes to the indicated state
- ◇ `NA`: the parser throws an error

In our framework, the automaton is a pair of functions: `sgoto` and `reduce`. The `sgoto` function encodes the links between the states, and so has type

```
(α, β) grammar →
(α, β) state → (α, β) symbol → (α, β) state
```

where the symbol is the label on the arrow. We have merged what is traditionally presented as two “tables”: the shift and goto tables, where the shift table encoded information for terminals and the goto table did the same for nonterminals.

The `reduce` function is of type:

```
(α, β) grammar → (α, β) state → β → (α, β) rule list
```

It returns a list of possible rules that can be reduced in the given state, when the next input symbol is also provided. When the machine has been constructed appropriately (from an SLR grammar), the list will always be empty or just one element long.

These functions are combined using a while combinator, called `mwhile`, of type

```
(α → bool) → (α → α option) → α → α option option
```

The type `'a` is the type of the loop-state. The first argument is a boolean condition on states specifying when the loop should continue. The second argument encodes the loop body, allowing for the possibility that the loop execution terminates abnormally (as happens in our case when the parser detects a string not in the grammar’s language). The third argument is the initial state. The result encodes normal termination, abnormal termination (`SOME NONE`) and failure to terminate (`NONE`).

5.3.2 Constructing the parser

The architecture of the parser-construction process is shown in Figure 5.3. The parser is a DFA that provides the transition for each of the states the stack may be in. Based on that, the stack is manipulated to result in a parse tree for the input (or alternatively throws an error).

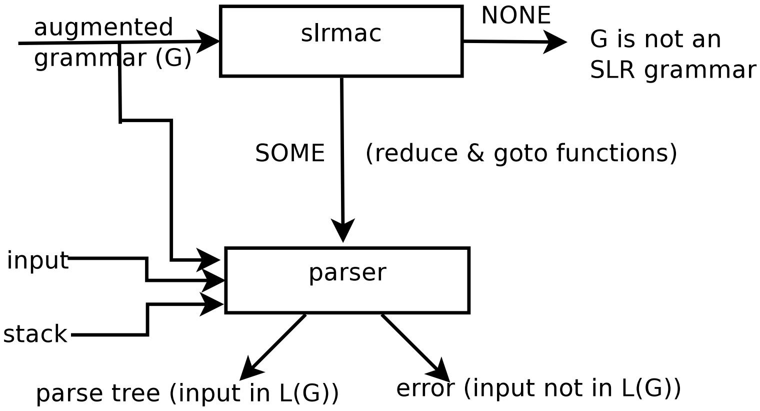

augmented
grammar (G)
slrmac
NONE
G is not an
SLR grammar
SOME
(reduce & goto functions)
input
stack
parser
parse tree (input in L(G))
error (input not in L(G))

Figure 5.3 – The structure of the SLR parser-construction process.

The first step towards parsing the grammar is to augment it as explained in Section 5.1. The augmentation adds an extra rule that introduces a new start symbol and a marker that appears at the end of all the words in the language of the grammar. The parser uses this rule for reduction exactly when it has accepted the input word. This ensures that the parser always 'spots' the end of input.

HOL Definition 5.3.1 (auggr)

```
auggr g s eof =
  if NTS s ∉ nonTerminals g ∧ TS eof ∉ terminals g then
    SOME
      (G ([rule s [NTS (startSym g); TS eof]] ++ rules g) s)
  else
    NONE
```

The failure point indicated here by NONE ensures that the symbols being introduced are 'fresh' and not currently part of the grammar.

5.3.2.1 Building the parsing table

The parsing table represents the DFA and maps the item sets to corresponding actions.

A single item cannot be used to define the state of the parser because it may not know in advance which rule is going to be used for reduction. Therefore, we need to use an item set that includes all the possible rules that may be valid at a particular state of the parser.

As previously mentioned, the item $A \to \alpha \bullet B\beta$ indicates that the parser expects to parse the nonterminal B next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must include all items describing how B itself may be parsed. This involves taking the closure of all the items in an item set until all nonterminals preceded by the dot are accounted for.

The DFA is comprised of the reachable items sets and the transitions between them. The starting state of the automaton is the closure of the item corresponding to the newly added rule in the augmented grammar. This involves finding all the rules that can be reached from the start symbol.

The full parsing table could be built upfront for reference. However, we compute the next state on the fly. This gives us simpler proof goals and also assists in reasoning about the program properties.

Given a state and a symbol, the `nextState` returns the possible items valid for that move.

HOL Definition 5.3.2 (`nextState`)

```
nextState g itl sym = closureML g (moveDot itl sym)
```

The function `closureML` (HOL Definition 5.5.3) computes the closure of item sets. We defer the internal details to the section where we discuss the exectability issue. Function `moveDot` is defined as:

HOL Definition 5.3.3 (`moveDot`)

```
moveDot [] sym = []
moveDot (item str (a,s::ss)::it) sym =
  if s = sym then
    item str (a ++ [s],ss)::moveDot it sym
  else
    moveDot it sym
moveDot (item v4 (v8,[])::it) sym = moveDot it sym
```

Based on the contents of the DFA state and the input, the parser performs two kinds of actions (excepting the error case).

The reduction action returns all the possible rules that can be used when performing a reduction in a state given some input symbol. Since we are implementing an SLR parser, a rule is considered valid for reduction if and only if the input symbol belongs in the follow set of the left-hand side of the rule.

The computation of the follow set for determining a reduction is the way in which an SLR parser differs from an LR(0) parser. An SLR parser can recognise more grammars by reducing the number of shift-reduce and reduce-reduce conflicts. This is done by using one symbol lookahead. Shift-reduce conflict occurs when a *handle* is recognised on the stack and there is also an incomplete item in the state. For a sentential form γ, a handle is a production $A \Rightarrow \beta$ and a position of β in γ, such that β may be replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. There must only be terminals to the right of a handle. If a grammar is unambiguous, every right sentential form has a unique handle.

Reduce-reduce conflict occurs when the current state provides more than one way of reducing the top of the stack. Function `reduce` g itl s returns a list of all the rules that are applicable given item list itl and input s. An item `item` ℓ `(`r_1`,`r_2`)` can be considered for reduction if and only if r_2 is empty and s belongs in the follow set of ℓ. If `reduce` returns a list containing more than one element then we have a reduce-reduce conflict.

HOL Definition 5.3.4 (reduce)

```
reduce g [] s = []
reduce g (item ℓ (r,[])::it) s =
  if TS s ∈ followSet g (NTS ℓ) then
    rule ℓ r::reduce g it s
  else
    reduce g it s
reduce g (item ℓ (r,v10::v11)::it) s = reduce g it s
```

The other scenario is when we have not read a full handle so we just need to shift the input onto the stack and compute the next state. This is the shift-goto action. The function `sgoto` returns the next state for a given symbol.

HOL Definition 5.3.5 (sgoto)

```
sgoto g itl sym = nextState g itl sym
```

5.3.2.2 Determining Conflicts in the Table

As explained before, while building the parsing table, one might encounter shift-reduce or reduce-reduce conflicts. The `slrmac` function predicts such conflicts for any state that is reachable in the DFA for the grammar. If no conflicts exist then the function returns the shift-goto and reduce functions for the given grammar. Otherwise `NONE` is returned to indicate failure when a conflict-free DFA cannot be created.

HOL Definition 5.3.6 (slrmac)

```
slrmac g initState =
  if okSlr g initState then SOME (sgoto g,reduce g) else NONE
```

Given a grammar and an initial state, auxiliary function `okSlr` determines conflicts in the DFA. When predicting conflicts, `okSlr` only looks at those states that are reachable by recognising viable prefixes *vp* for the grammar *g*.

HOL Definition 5.3.7 (`okSlr`)

```
okSlr g initState ⟺
  ∀vp state symlist.
    isWord symlist ∧ trans g (initState,vp) = SOME state ⇒
    noError (sgoto g,reduce g) symlist state
```

A reachable state corresponding to a viable prefix is determined using `trans`. Given a pair of a state and prefix, and a grammar, function `trans` gives back a state if the DFA can reach the state once all of the prefix has been consumed, *i.e.* prefix is viable with respect to the DFA, otherwise it returns `NONE`.

HOL Definition 5.3.8 (`trans`)

```
trans ag (s,[]) = SOME s
trans ag (s,sym::rst) =
  case moveDot s sym of
     [] → NONE
  || v2::v3 → trans ag (closureML ag (v2::v3),rst)
```

For such reachable states, `okSlr` determines whether reading an input string *symlist* gives a conflict or not. This is done using the `noError` function. Functions `sgoto` and `reduce` are used to figure out the shift-reduce or reduce-reduce conflicts. Function `noError` takes three arguments, a pair of shift (*sf*) and reduce (*red*) functions with respect to some grammar, a list of input symbols and a state.

HOL Definition 5.3.9 (`noError`)

```
noError (sf,red) [] st ⟺ T
noError (sf,red) (sym::rst) st ⟺
  st = [] ∨
  case red st (ts2str sym) of
     [] → noError (sf,red) rst (sf st sym)
  || [r] → sf st sym = []
  || r::v6::v7 → F
```

The process of creating an SLR machine (DFA) for a grammar indicates upfront whether the grammar can be handled by an SLR parser. If the construction of the DFA is successful, the parser will always be able to build a parse tree for a string in the language of the grammar.

5.3.2.3 Putting it all together

The output of the `parser` is a parse tree. A parse tree is represented as following:

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

Parse trees are discussed in detail in Section 4.1. The parser is encoded as:

HOL Definition 5.3.10 (`parser`)

```
parser (initConf, eof, oldS) m sl =
  (let out =
          mwhile (λs. ¬exitCond (eof,NTS oldS) s)
            (λ (sli, stli, csli). parse m (sli, stli, csli))
            (sl, [], [initConf])
   in
     case out of
        NONE → NONE
     || SOME NONE → SOME NONE
     || SOME (SOME (slo, [], cs)) → SOME NONE
     || SOME (SOME (slo, [(st_1, pt)], cs)) → SOME (SOME pt)
     || SOME (SOME (slo, (st_1, pt)::v_21::v_22, cs)) → SOME NONE)
```

The function `parser`, while consuming the input, determines the action to be performed: `GOTO`, `REDUCE`, `NA` (Section 5.3.1). At the end, it returns one of the above mentioned outputs.

The implementation of the `parser` ensures that if it does return a parse tree then the root node of the tree corresponds to the start symbol of the augmented grammar. The type of the `parser` is:

```
((α, β) symbol × (α, β) state) × β × α →
((α, β) state → (α, β) symbol → (α, β) state) ×
((α, β) state → β → (α, β) rule list) →
(α, β) symbol list → (α, β) ptree option option
```

The parser takes the following inputs. The first input is a tuple consisting of the initial configuration (new start symbol and the initial state), the end of input terminal and old start nonterminal. The second input is a pair consisting of the shift-goto function and the reduce function. The final input is the list of input symbols. The output is interpreted as discussed earlier.

The while combinator `mwhile` (see Section 5.3.1) performs the `parse` step for each of the input symbol resulting in a new 'state' for the parser. This encapsulates the input symbols still to be parsed (*sli*), the current stack view (*stli*) and the list of the states passed up to the current state *csli*. The `exitCond` for the loop is when the stack contains only the original start symbol (*oldS*) and the only remaining input symbol is the end of input marker *eof*. At the end of a successful parse, the current stack should have only the start symbol of the old grammar, which is then used to output the final parse tree.

The `parse` step is our HOL Definition 5.3.11. The input cannot be empty since the end of file marker is never shifted onto the stack. If only a single input symbol is left to parse then it must be *eof*. The only valid action at this stage has to be a reduction to the start symbol of the augmented grammar. For more than one input, function `getState` returns the appropriate action to be performed based on the DFA m.

HOL Definition 5.3.11 (parse)

```
parse m (inp,os,(s,itl)::rem) =
  case inp of
     [] → NONE
  ||  [e] →
       (let newState = getState m itl e in
          case newState of
             REDUCE ru → doReduce m ([e],os,(s,itl)::rem) ru
          ||  GOTO st → NONE
          ||  NA → NONE)
  ||  e::$v_4$::$v_5$ →
       (let newState = getState m itl e in
          case newState of
             REDUCE ru →
               doReduce m (e::$v_4$::$v_5$,os,(s,itl)::rem) ru
          ||  GOTO st →
               if isNonTmnlSym e then
                  NONE
               else
                  SOME
                    ($v_4$::$v_5$,((e,st),Leaf (ts2str e))::os,
                     push ((s,itl)::rem) (e,st))
          ||  NA → NONE)
```

(Function `ts2str (TS tm)` *returns tm. Function* `push` *adds an element to the front of a list.)*

5.4 Key proofs

With the above setup, we now have a parser generator in HOL. To formally verify that the procedure described previously is indeed correct, we would like to demonstrate that if an automaton is generated, then the language accepted by the automaton is the same as the language defined by the grammar. This goal is naturally split into two inclusion results: that everything accepted by the machine is in the language (“soundness”), and that everything in the language is accepted by the machine (“completeness”).

There are various invariants that are inherent to establishing these results because of the way the parser works and has been implemented. These invariants are needed to prove the properties described in the latter section and so are summarised below.

The `parser_inv` states implementation-specific properties about the stack. The clauses correspond to the following three properties. First, the `validptree_inv` invariant holds. Second, the initial start state is never popped off from the stack. Third, the items in each of the state on the stack correspond to some grammar rule (`validStates`).

HOL Definition 5.4.1 (`parser_inv`)

$$
\begin{array}{l}
\texttt{parser\_inv}\ g\ \mathit{stl}\ \mathit{csl}\ \iff \\
\quad \texttt{validptree\_inv}\ g\ \mathit{stl}\ \land\ \neg\texttt{NULL}\ \mathit{csl}\ \land\ \texttt{validStates}\ g\ \mathit{csl}
\end{array}
$$

Invariant `validptree_inv` stands for two conditions. First, that each symbol on the stack corresponds to the node of the associated parse tree (`validStkSymTree`). This is the first conjunct of `validptree_inv`. Second, for all the nonterminals on the stack, the associated parse trees are valid with respect to the given grammar. This is the second conjunct of `validptree_inv`. Proving this property as an invariant for the parser lets us derive that in the end, if the parser is able to reduce the stack symbols to the start symbol, then the corresponding parse tree must be valid as well.

HOL Definition 5.4.2 (`validptree_inv`)

$$
\begin{array}{l}
\texttt{validptree\_inv}\ g\ \mathit{stl}\ \iff \\
\quad \texttt{validStkSymTree}\ \mathit{stl}\ \land \\
\quad \forall s\ t. \\
\qquad (s,t)\ \in\ \texttt{stacklsymtree}\ \mathit{stl}\ \Rightarrow \\
\qquad \texttt{isNonTmnlSym}\ s\ \Rightarrow \\
\qquad \texttt{validptree}\ g\ t
\end{array}
$$

(Function `stacklsymtree` *returns a list of pairs of the form* $(\mathit{symbol}, \mathit{tree})$*, i.e. removes the state component from each of the stack elements.)*

The DFA for accepting $L(G)$ for LR(1) grammars works by computing valid items for each viable prefix. Invariant `validItem_inv` takes a grammar ag and the current stack (REVERSE *stl*) and asserts that for all nonempty stacks *stk* that are sub-stacks of the current stack, the current state of *stk* (stkItl (REVERSE *stk*)) can be reached from the initial state (initItems *ag* (rules *ag*)) by reading the symbols on the stack (stackSyms (REVERSE *stk*)). This is essentially stating that stackSyms (REVERSE *stk*) is the viable prefix for state stkItl (REVERSE *stk*). Since we always change the top *i.e.* the front of the stack, reversing the stack gives us access to the right order in which the input symbols were read.

HOL Definition 5.4.3 (validItem_inv)

```
validItem_inv (ag,stl) ⟺
  ∀stk.
    stk ≼ REVERSE stl ⇒
    ¬NULL stk ⇒
    trans ag
      (initItems ag (rules ag),stackSyms (REVERSE stk)) =
      SOME (stkItl (REVERSE stk))
```

(Here $x \preceq y$ holds if and only if x is a prefix of y. Function stkItl *returns the state i.e. the item list of the top of the stack. Function* stackSyms *returns the input symbols stored on the stack.)*

5.4.1 Validity of the parse tree generated

If the parser results in a parse tree, the tree is valid with respect to the grammar for which the parser was generated. This means that the parse tree was built using rules present in the given grammar.

HOL Theorem 5.4.1

```
auggr g s eof = SOME ag ⇒
slrmac ag = SOME m ⇒
parser_inv ag stl csl ⇒
parser
  ((NTS (startSym ag),initItems ag (rules ag)),eof,
   startSym g) m sl =
  SOME (SOME tree) ⇒
validptree ag tree
```

5.4.2 Equivalence of the output parse tree and the input string parsed

The main predicate of interest here is the leaves_eq_inv. This is defined in HOL as:

HOL Definition 5.4.4 (leaves_eq_inv)

```
leaves_eq_inv orig sl stl ⟺ stacktreeleaves stl ++ sl = orig
```

(Function `stacktreeleaves` *returns the concatenated leaves of all the parse trees stored on the stack, i.e. the input string that has already been parsed.)*

The term `leaves_eq_inv` *orig sl stl* is true when the original input *orig* is equal to the input still to be consumed (*sl*) appended to the concatenated leaves of the parse trees stored on the stack (`stacktreeleaves` *stl*). This ensures that the grammar rules being applied to form the parse tree correspond to the input string being parsed and the leaves of the resulting parse tree are equal to the original input string. Thus, when the parser returns a tree, the original input must equal the leaves of the tree plus the end of input marker. This is stated in HOL as:

HOL Theorem 5.4.2

```
auggr g s eof = SOME ag ⇒
¬NULL csl ⇒
validStates ag csl ⇒
slrmac ag = SOME m ⇒
inis = (NTS (startSym ag),initItems ag (rules ag)) ⇒
parser (inis,eof,startSym g) m sl = SOME (SOME tree) ⇒
sl = MAP TS (leaves tree) ++ [TS eof]
```

5.4.3 Soundness of the parser

To prove soundness, we have to show that the input for which a valid parse tree can be constructed is in the language of the grammar.

HOL Theorem 5.4.3

```
auggr g s eof = SOME ag ∧ slrmac ag = SOME m ∧
parser_inv ag stl csl ∧
parser
  ((NTS (startSym ag),initItems ag (rules ag)),eof,
   startSym g) m sl =
  SOME (SOME tree) ⇒
sl ∈ L ag
```

To achieve this result we proved that when a parse tree is valid with respect to a grammar, one can derive the leaves from the root node. This is HOL Theorem 4.1.1. Here it is once again:

```
validptree g t ⇒
(derives g)* [ptree2Sym t] (MAP TS (leaves t))
```

From the implementation of `parser` and `parse`, we know that if a parse tree is the output then the root node corresponds to the start symbol of the augmented grammar. This fact along with the above theorem, Theorem 5.4.1 and Theorem 5.4.2, allows us to derive membership in the language of the grammar.

5.4.4 Completeness of the parser

To show completeness, we have to prove that if the input is in $L(G)$ then the parser will terminate with a parse tree. The termination was established by quantifying the steps of the parser and showing that in a finite number of steps the parser reaches the end state.

The completeness proof involves assuming that the grammar is a 'generating grammar', *i.e.* all the nonterminal symbols generate some terminal string. Note that we proved that 'removing useless symbols, those that do not derive a word, does not affect the language of a grammar' in Section 2.2.

HOL Theorem 5.4.4

```
auggr g st eof = SOME ag ⇒
sl ∈ L ag ⇒
slrmac ag = SOME m ⇒
(∀nt. nt ∈ nonTerminals ag ⇒ gaw ag nt) ⇒
initState = (NTS st,initItems ag (rules ag)) ⇒
∃tree.
  parser (initState,eof,startSym g) m sl = SOME (SOME tree)
```

As previously mentioned, function `parser` depends on function `parse` to do the one-pass parse for a single input symbol. To establish termination we must to show that in a finite number of steps the parser can finish reading the input. This is done by defining the `takesSteps` function given below.

HOL Definition 5.4.5 (`takesSteps`)

```
takesSteps 0 f g s₀ s ⟺ s₀ = s
takesSteps (SUC n) f g s₀ s ⟺
  ¬g s₀ ∧ ∃s′. f s₀ = SOME s′ ∧ takesSteps n f g s′ s
```

Function `takesSteps` $n\ f\ g\ s_0\ s$ returns true if state s_0 can transition to state s using n applications of function f while the guard g does not hold. Function f will then be `parser`, the guard will be the exit condition `exitCond` while the states will be the tuple of the input symbols to be parsed and the stack. For the parser to terminate there *must* exist an n such that in the final state the input is empty and the stack only consists of the end of file marker. We prove that as long as the invariants (discussed earlier) hold then then there must be a `parse` step possible.

Finally, to establish completeness we must show that if the input string is in the language of the grammar then the parser should give back a correct parse tree. This is easily achieved using HOL Theorems 5.4.4 and 5.4.1.

5.4.5 SLR grammars are unambiguous

A grammar is unambiguous if for each input $w \in L(G)$, w has a unique rightmost derivation. We can also define ambiguous-ness with respect to leftmost derivation. In the case of an SLR parser, we are constructing a rightmost derivation and the former definition is more natural.

The `RTC` closure used to define derivations only asserts the existence of a derivation from one point to another. To be able to reason about unique derivations, we construct a concrete derivation by casting the `RTC` as a list using the derivation list notation (Section 2.1), where adjacent elements represent a single-step derivation.

The membership of w in $L(G)$ is represented by a derivation list starting from the start symbol of G and ending in w. A derivation for w is unique if and only if all possible derivation lists are identical. We define the function `isUnambiguous` is follows:

HOL Definition 5.4.6 (`isUnambiguous`)

```
isUnambiguous g ⟺
  ∀sl.
    sl ∈ L g ⇒
    ∀dl dl′.
      rderives g ⊢ dl ◁ [NTS (startSym g)] → sl ∧
      rderives g ⊢ dl′ ◁ [NTS (startSym g)] → sl ⇒
      dl = dl′
```

The augmented grammar ag, in which all nonterminals are generating, is unambiguous if `slrmac` returns machine m containing shift-goto and reduce functions for ag. That is, no shift-reduce or reduce-reduce conflicts were detected by `slrmac`.

HOL Theorem 5.4.5

```
auggr g st eof = SOME ag ⇒
(∀nt. nt ∈ nonTerminals ag ⇒ gaw ag nt) ⇒
slrmac ag = SOME m ⇒
isUnambiguous ag
```

5.5 An executable SLR parser

For the most part, the HOL definitions turn out to be executable. However, for the sake of simplicity and clarity, many of our definitions were written in a style that favoured mathematical ease of expression. The use of existential quantifiers, and the reflexive and transitive closure in such definitions make them unexecutable. Here we describe how the defined functions can be re-expressed in a way that makes them acceptable to HOL's emitML technology. Our general approach was to take an existing function f, and define a new f`ML` constant. After proving termination for the typically complicated recursion equations defining f`ML`, we then had to show that f`ML`'s behaviour encompassed f's.

In this section we describe our executable implementations of two of the non-executable HOL definitions. The remaining two are covered in more detail in Sections 6.1.1 and 6.1.2.

Even though the semantic versions of some definitions were more tractable for proving properties such as our language inclusion results, there have been places where it was decided to value executability over succinctness of presentation. For example, the closure function on item sets required to compute states (given below) is much clearer.

HOL Definition 5.5.1 (`closure`)

```
closure g itl =
  {item sym ([],r) |
   ∃ℓ r1 r2 nt sfx.
     item ℓ (r1,NTS nt::r2) ∈ itl ∧
     (derives g)* [NTS nt] (NTS sym::sfx) ∧
     rule sym r ∈ rules g}
```

However, it cannot be executed because of the use of transitive closure, set comprehension and quantifiers. For executability, we used an alternative version which explicitly defines the process of forming the closure of items. The auxiliary `closureML1` function computes the 'one-pass' closure by adding items corresponding to the nonterminal symbol after the dot (`getItems`). The `closureML` function returns a list containing no duplicate elements when no more new items can be added.

HOL Definition 5.5.2 (`closureML1`)

```
closureML1 g [] = []
closureML1 g (item s (l1,[])::il) =
  item s (l1,[])::closureML1 g il
closureML1 g (item s (l1,TS ts::l2)::il) =
  item s (l1,TS ts::l2)::closureML1 g il
closureML1 g (item s (l1,NTS nt::l2)::il) =
    getItems (rules g) nt ++ [item s (l1,NTS nt::l2)] ++
    closureML1 g il
```

HOL Definition 5.5.3 (closureML)

```
  closureML g [] = []
  closureML g (v2::v3) =
    (let ril = rmDupes (v2::v3) in
     let al = rmDupes (closureML1 g ril) in
       if set ril ≠ set al then closureML g al else al)
```

(Function `rmDupes` *removes duplicate elements.)*

The recursive call in the definition of `closureML` required a termination proof similar to the one for `slrmac` discussed below.

5.5.1 An executable slrmac

An interesting termination case is encountered when we try to make the definition of `slrmac` definition executable.

Function `slrmac` checks whether the resulting table for the grammar has any conflicts or not. It is not strictly a necessary component of the parser generator but does assist in stating some of the proofs. For example, with this function we can assert that if we can build a parse table for a grammar and the input belongs in the language of the grammar then the parser will output a parse tree.

Building the parse table involves traversing the state space to find the next state for each of the symbols in the grammar, starting from the initial state.

HOL Definition 5.5.4 (visit)

```
  visit g sn itl =
    if ¬ALL_DISTINCT itl ∨ ¬validItl g itl then
      []
    else
      (let s = asNeighbours g itl (SET_TO_LIST (allSyms g)) in
       let rem = diff s sn in
         rem ++ FLAT (MAP (λ a. visit g (sn ++ rem) a) rem))
```

The parse table builder here is the `visit` function. Starting in the initial state it follows the transitions for each of the symbols in the grammar until it can reach no more new states. The important thing here is to make sure states are not repeated otherwise we end up following the same path over and over again. The condition `ALL_DISTINCT` ensures that we don't loop forever by considering states where the same items might be repeated. Another check, `validItl` makes sure that the items in the state do correspond to some rule in the grammar.

Function `asNeighbours` takes a state and returns a state list. The state list contains states that can be reached by following each of the symbols in the input (*i.e.* transitions one-level deep).

HOL Definition 5.5.5 (`symNeighbour`)

```
symNeighbour g itl sym =
  rmDupes (closureML g (moveDot itl sym))
```

It uses `symNeighbour` to shift the dot past the current symbol and get the state corresponding to it. The resulting state contains no duplicates (`rmDupes`).

HOL Definition 5.5.6 (`asNeighbours`)

```
asNeighbours g itl [] = []
asNeighbours g itl (x::xs) =
  symNeighbour g itl x::asNeighbours g itl xs
```

For function `visit`, the number of states seen increases at each recursive call. We also know that the number of possible states (even though it might be large) is finite. This is because we have a finite number of symbols in our grammar and a finite number of rules as well. From this we can deduce that the number of states that have not been encountered decreases at each call. This forms our termination argument.

```
measure (λ(g,sn,itl). CARD (allGrammarItls g DIFF set sn))
```

With this to hand, we can implement an executable `slrmac`, called `slrmacML`, that checks the entire table for shift-reduce and reduce-reduce conflicts.

HOL Definition 5.5.7 (`slrmacML`)

```
slrmacML g itl [] = SOME (sgoto g,reduce g)
slrmacML g itl (sym::rst) =
  if slrML4Sym g itl sym = NONE then
    NONE
  else
    slrmacML g itl rst
```

The auxiliary function `slrML4Sym` checks for conflicts for one symbol with respect to a state.

HOL Definition 5.5.8 (`slrML4Sym`)

```
slrML4Sym g [] sym = SOME (sgoto g,reduce g)
slrML4Sym g (i::itl) sym =
  (let s = sgoto g i sym in
   let r = reduce g i (ts2str sym) in
     case (s,r) of
        ([],[]) → slrML4Sym g itl sym
     || ([],[v12]) → slrML4Sym g itl sym
     || ([],v12::v16::v17) → NONE
     || (v8::v9,[]) → slrML4Sym g itl sym
     || (v8::v9,[v20]) → NONE
     || (v8::v9,v20::v26::v27) → NONE)
```

5.6 Conclusions

To realise the ambition of fully verified translation from source to machine code, all phases in the compilation process should either be verified or subject to verification after the fact. These two strategies are implemented in what have been termed *verified* or *verifying* compilers respectively. As we have already commented, one might imagine that the appropriate strategy for parsing would be to verify the output of an external tool. This would be *verified parsing*. For example, a verifying parser would mesh with Blazy, Dargaye and Leroy's work on the formal verification of a compiler front-end for a subset of the C language [12], which otherwise ignores parsing as an issue.

In their formalisation of what they call *The beginning of formal language theory*, Courant and Filliâtre [18] have presented the extraction of a parser generator in Coq theorem prover.

Koprowski and Binsztok [40] presented a parser interpreter developed in Coq. They prove the parser to be correct and terminating. Their interpreter is based on parsing expression grammars (PEGs). PEGs were introduced by Ford [22] in 2004. Even though PEGs look similar to CFGs, the interpretation of the rules in PEGs is different from that of CFGs. In parsing expression grammars, if a string parses, it has exactly one parse tree. CFGs on the other hand can be ambiguous. Another way in which PEGs are different from CFGs is that they cannot handle left-recursive rules.

We have presented work towards the formal verification of an SLR parser generator. Most of the functions are directly executable. For those that we thought were better represented by set comprehensions and quantifiers, we have presented executable definitions of behaviourally equivalent alternatives. This conversion also illustrated the gap between simple textbook definitions and a verifiable executable implementation in a theorem prover. The process of conversion might be straightforward, but issues like termination which can be ignored when dealing with semantic definitions become necessary when executability comes into play. It also highlights how eminently suitable HOL is for developments of this kind, especially with its facility of emitting verified HOL definitions as ML code.

Table 5.1 shows the proof effort for the mechanisation covered in this chapter.

Mechanisation		LOC	#Definitions	#Proofs
Background		5000	69	228
Completeness		7168	1	58
Non-ambiguity		1434	1	11
Invariants	`validItem_inv`	2452	7	51
	`validptree_inv`	478	5	23
	`leaves_eq_inv`	182	2	10
	`parser_inv`	241	1	4

Table 5.1 – Summary of the mechanisation effort for SLR parser generator

The biggest part of the SLR parser generator work is the proof of completeness which uses the definition of `takesSteps` to establish termination. The background mechanisation refers to the common definitions such as those for closure, parser, etc. and the common proofs based on these definitions. As expected, the hardest and most tedious invariant to establish turned out to be `validItem_inv`. The proof of soundness depends on the invariants `validptree_inv` and `leaves_eq_inv`, and some of the background work on parse trees. As such as we have provided the summary for the invariants separately which aptly reflects the effort gone in to the proof for establishing soundness. The effort for invariant `parser_inv` is small since the properties in its definition have already been established as part of other invariants.

Chapter 6

Issues in Automation

It's blatantly clear
You stupid machine, that what
I tell you is true

Michael Norrish

Contents

By sheer coincidence, Michael Norrish has provided the perfect HOL *haiku*[1] to sum up the central theme of this chapter. Yes, *we* know what we tell the machine is blatantly

[1] a form of Japanese poetry

clear, but the machine needs a bit more coaxing and cajoling before it will accept our view.

There is a common thread running through all the automation we have undertaken. These are the issues that occur in mechanising a theory which at least at this stage are independent of the theorem prover itself (more in Section 6.6).

The leading chapters have discussed the dilemma of choosing between relations and functions, identifying and providing mechanised proofs for both explicit and implicit gaps present in the textual treatment, the problems of reconciling a readable definition *vs.* an executable one and so on. This chapter presents a broad categorisation of these issues, which of course at times deal with overlapping concerns, and how they impact the complexity of mechanisation.

In our experience, going from a text proof to a formalised proof in a theorem prover inevitably resulted in locating the gaps mentioned above. We have categorised them as:

◇ explicit inferences that are too big a leap for a theorem prover;

◇ implicit assumptions that are too "obvious" for a human reader; and

◇ inherent invariants when implementing a text proof in a theorem prover.

Again, these categories are not totally distinct and therefore have overlapping elements with the others. These 'assumptions' include statements that suffice to convince a human reader of the validity of the proof but are too coarse to be translated exactly as they are in a theorem prover. Such assumptions are also statements that are left for the reader to prove for themselves. In the latter case, strategies for mechanisation, which are in the majority of the cases based on the textbook treatment, have to be inferred from scratch. Another aspect of these assumptions, is the detail that has to be set in concrete in a theorem prover when defining a relation or a function. This in turn is accompanied by the inherent conditions that need to be carried around in a proof because of the way the implementation has been done.

For example, if $N_1N_2 \ldots N_n \Rightarrow^* \alpha$ then each nonterminal N_i must derive a (possibly empty) portion of α. A statement such as this falls in the first two categories. It does get stated explicitly in Hopcroft and Ullman when such a breakdown of a derivation is required but a proof is assumed for such an obvious goal.

Inherent invariants are common when implementing complex definitions where a lot of state information gets passed around. We have already seen the explosion of invariants in the implementation of an algorithm for Greibach Normal Form (Section 2.6) and that of SLR parser generator in Section 5.3.2.1.

Each of these issues has been further elaborated in the sections to come. These issues by and large are not the only ones but were the most common ones in our mechanisation process.

6.1 The cost of comprehension: executability *vs.* readability

The most obvious of such issues is the pull between making a definition readable versus making it executable. A verified parser is not of much use unless one can run it as well (Chapter 5). Clearly in such a case all the definitions for a parser need to be executable. In Section 5.5 we presented two executable definitions, the executable counterpart of `closure` over items and `slrmac` function.

In this section we present a detailed description of the effort required to have both a readable and an executable definition. We illustrate the process using the definitions of `nullable` and `firstSet` that are part of the SLR implementation.

6.1.1 Executable calculation of nullable nonterminals

The set of nullable sentential forms is defined as:

HOL Definition 6.1.1 (`nullable`)

$\texttt{nullable}\ g\ sl \iff (\texttt{derives}\ g)^*\ sl\ \texttt{[]}$

Observe how close the expression is to a textbook definition even though this one is a HOL definition.

The calculation of `nullable` is central to the calculation of the first sets and the follow sets required for the state change operations (reduce and shift-goto discussed in Section 5.3.1) in an SLR parser. Unfortunately, the above definition is not executable due to the use of `RTC`.

Instead we have to provide an alternate executable definition that can be exported to SML. This counterpart of the `nullable` function is given below.

HOL Definition 6.1.2 (`nullableML`)

```
nullableML g sn [] ⟺ T
nullableML g sn (TS x::t) ⟺ F
nullableML g sn (NTS n::t) ⟺
  if NTS n ∈ sn then
    F
  else
    EXISTS (nullableML g (NTS n::sn)) (getRhs n (rules g)) ∧
    nullableML g sn t
```

(`EXISTS` is an executable function over lists and is different from the logical ∃.)

The `nullableML` function determines whether or not a list of symbols (a sentential form) can derive the empty string. When the sentential form includes a terminal symbol, the result is false. When a nonterminal is encountered, we must recursively determine if any of the nonterminal's RHSs might derive the empty string.

Clearly, in terms of comprehension the executable definition is nowhere close to `nullable`. Even adding comments, annotations, etc. does not change the fact that `nullable` reflects the process more neatly and succinctly.

There is a problem associated with using either of the two definitions. First, `nullableML` is too complicated; HOL cannot automatically deduce the termination condition for this function. If we want an 'executable nullable' then we need to show that `nullableML` terminates. Second, if we want to use a more readable definition without loss of executability then we need to prove the equivalence between the two so that they can be used interchangeably in the theorems.

First, we show termination.

Termination of `nullableML` In order to ensure that the recursion terminates, we have already introduce a 'seen' list (*sn*) in `nullableML` definition. This gets updated with the nonterminal that is being visited when we expand the nonterminal to its RHSs and recurse over the expansions. This corresponds to the `getRhs` *n* `(rules` *g*`)` clause in the definition of `nullableML`. To then convince HOL that this function terminates, we must find a well-founded relation on the arguments of `nullableML`. Because a list containing a nonterminal may expand into a list of symbols of arbitrary length, we cannot simply use the length of the sentential form as a measure. Instead we use the lexicographic combination to deal with the scenario where some arguments are reduced in some calls and others are reduced in different recursive calls (see Section 6.2 for more detail):

```
measure (λ(g,sn). CARD (nonTerminals g DIFF set sn)) LEX
measure LENGTH
```

We assert that either the number of symbols except the ones in the seen list decreases, or that the length of the sentential form decreases. The former corresponds to the first conjunct in the third clause in the definition while the latter takes care of the second conjunct.

Equivalence between `nullable` and `nullableML` Now we deal with second issue and show the equivalence between the new HOL constants and the originals. Then we will know that execution of SML code will provide a behaviour corresponding to that of the formal HOL entity.

The proof requires showing, first,

HOL Theorem 6.1.1

```
nullableML g sn ℓ ⇒ nullable g ℓ
```

and second,

HOL Theorem 6.1.2

```
nullable g ℓ ⇒ sn = [] ⇒ nullableML g sn ℓ
```

to conclude the equivalence

HOL Theorem 6.1.3

```
nullable g ℓ ⟺ nullableML g [] ℓ
```

The first implication (HOL Theorem 6.1.1) is easy to show since `nullableML` traces a specific derivation to ϵ. To prove the second implication, we need to show that for any derivation, we can construct an equivalent derivation that will get accepted by `nullableML`.

This may seem easy and straightforward on surface but the actual proof turns out to be complicated. As previously outlined, for a sentential form to be nullable, it cannot have a terminal symbol. We look at the non-trivial case, *i.e.* when the sentential form itself is not empty. A sentential form $N_1N_2...N_n$ is nullable if and only if the individual derivations for the N_is itself are nullable.

$$
\begin{array}{l}
\mathtt{N}_1 \Rightarrow^* \epsilon \\
\mathtt{N}_2 \Rightarrow^* \epsilon \\
\vdots \\
\mathtt{N}_n \Rightarrow^* \epsilon
\end{array}
$$

`nullable` g sf asserts the existence of *some* derivation from sentential form sf to ϵ. On the other hand, `nullableML` can be seen to be constructing a concrete derivation with a specific property, *i.e.* in each individual derivation, the symbols cannot be repeated. This property corresponds to the check `NTS` $A \in sn$ in `nullableML`. The function goes on to compute the 'nullables' for `NTS` A if and only if it has not been seen before. This check gives us termination but it also makes the equivalence proof harder.

To prove the latter implication (HOL Theorem 6.1.2), we need to show that each derivation without any constraints on its form, can be recast into a derivation where the individual derivations of ϵ do not have repeated symbols. To prove this we need to show that any derivation of the form $N \Rightarrow^* \epsilon$ can be recast into a new derivation (possibly smaller), that gets accepted by `nullableML`.

We first define:

HOL Definition 6.1.3 (derivNts)

$\texttt{derivNts}\ d\ =\ \texttt{set}\ (\texttt{FLAT}\ d)$

Function `derivNts` gives us all the symbols involved in a derivation. Using this notion, we then assert that for a derivation dl, $\alpha N \beta \Rightarrow^* \alpha^{'} \gamma \beta'$, symbol N has to derive some (possibly empty) portion of the final string $\alpha^{'} \gamma \beta^{'}$. Since the N-derivation is part of the bigger derivation dl, the size of the N-derivation and the number of symbols in the N-derivation cannot exceed that of dl. Again, an assumption such as this is always deemed too obvious for a proof of its own in textbooks. HOL, of course, requires an explicit proof of such properties:

HOL Theorem 6.1.4

$$\begin{array}{l}
\texttt{derives}\ g \vdash dl \lhd pfx\ \texttt{++}\ [\texttt{NTS}\ N]\ \texttt{++}\ sfx \to \texttt{LAST}\ dl \Rightarrow \\
\exists pfx'\ rhs\ sfx'. \\
\quad \texttt{LAST}\ dl\ =\ pfx'\ \texttt{++}\ rhs\ \texttt{++}\ sfx' \wedge \\
\quad \exists dl'. \\
\quad\quad \texttt{derives}\ g \vdash dl' \lhd [\texttt{NTS}\ N] \to rhs \wedge |dl'| \leq |dl| \wedge \\
\quad\quad \texttt{derivNts}\ dl' \subseteq \texttt{derivNts}\ dl
\end{array}$$

With the help of the theorem above, we can now prove that a nullable derivation (d_0) for a nonterminal N can be recast into another nullable derivation (clause about existence of derivation d) such that the symbols of d are a subset of those of d_0 and N does not repeat in any of the subsequent expansions in d. This is the last conjunct of the exists clause in the theorem below.

HOL Theorem 6.1.5

$$\begin{array}{l}
\texttt{derives}\ g \vdash d_0 \lhd [\texttt{NTS}\ N] \to [\,] \Rightarrow \\
\exists d. \\
\quad \texttt{derives}\ g \vdash d \lhd [\texttt{NTS}\ N] \to [\,] \wedge \texttt{derivNts}\ d \subseteq \texttt{derivNts}\ d_0 \wedge \\
\quad |d| \leq |d_0| \wedge \forall sf.\ sf \in \texttt{TL}\ d \Rightarrow \texttt{NTS}\ N \notin sf
\end{array}$$

Now that we have established the conditions needed for `nullableML` to work, *i.e.* the existence of the recast derivation, we can prove the following:

HOL Theorem 6.1.6

```
derives g ⊢ dl ◁ [NTS N] → [] ⇒
dl ≠ [] ⇒
derivNts dl ∩ set sn = ∅ ⇒
(∀sf. sf ∈ TL dl ⇒ NTS N ∉ sf) ⇒
nullableML g sn [NTS N]
```

The theorem states that if nonterminal N is nullable and nonterminals in dl and sn are disjoint and N does not occur in its own derivation stream then N is also nullable using the executable definition `nullableML`. The proof of the above theorem follows from induction of the length of dl. From this we can prove:

HOL Theorem 6.1.7

```
nullable g ℓ ⇒
∀sn. (∀sym. sym ∈ sn ⇒ ¬nullable g [sym]) ⇒ nullableML g sn ℓ
```

which gives us the proof for the “only if” direction (HOL Theorem 6.1.2) direction. The proof is by induction on sentence ℓ. Note the broader condition on the members of the seen list, `∀sn sym. sym ∈ sn ⇒ ¬nullable g [sym]`. The induction does not work without this particular condition on sn. In the proof for HOL Theorem 6.1.2, sn gets instantiated as the empty list.

This ‘obvious’ property of nullable derivations is usually ‘assumed’ in textbook proofs, but plays a central role when proving the equivalence between a mathematical definition and an executable one.

With this equivalence we know now that execution of SML code will provide a behaviour corresponding to that of the formal HOL entity.

6.1.2 Executable calculation of first set

In a similar vein, we tackle the problem of calculation of the symbols in the first set of a sentential form.

In HOL we define the first set of a sentential form as follows:

HOL Definition 6.1.4 (`firstSet`)

```
firstSet g ℓ =
  {TS fst | ∃rst. (derives g)* ℓ ([TS fst] ++ rst)}
```

Again, the expression resembles closely what one might find in a textbook. On the other hand, it is definitely not executable, because of the use of set comprehension, the existential quantifier and the reflexive transitive closure (*).

So here is an executable version:

HOL Definition 6.1.5 (`firstSetML`)

```
firstSetML g sn [] = []
firstSetML g sn (TS ts::rest) = [TS ts]
firstSetML g sn (NTS nt::rest) =
  rmDupes
    (if NTS nt ∈ sn then
       []
     else
       (let r = getRhs nt (rules g) in
          FLAT (MAP (λa. firstSetML g (NTS nt::sn) a) r))) ++
  if nullableML g [] [NTS nt] then
    firstSetML g sn rest
  else
    []
```

The executable version follows a similar pattern to `nullableML` with the use of a seen list to keep track of the symbols encountered. This is helpful when providing a termination argument along the same lines as `nullableML`.

We will jump to the more difficult problem of proving the equivalence between the two versions.

HOL Theorem 6.1.8

```
s ∈ firstSetML g [] ℓ ⟺ s ∈ firstSet g ℓ
```

The easier "only if" direction follows from the induction principle generated by HOL when proving the termination for `firstSetML`.

HOL Theorem 6.1.9

```
TS t ∈ firstSet g sf ⇒ TS t ∈ set (firstSetML g [] sf)
```

The hard part is the "if" direction. The complication is going from a general derivation to a more specific one. In order to find out the symbols in the first set `firstSetML` traces the derivation corresponding to given symbols, first symbol being sym and remaining symbols being $syms$, in the following manner. If the symbol sym is terminal then we have a member for the first set being calculated. If sym is a nonterminal then the function evaluates the right-hand sides of the corresponding grammar rules and tries to recursively find the first set for each of the RHSs. If sym is nullable `firstSetML` also finds the first set of $syms$. To avoid looping indefinitely, first set are only calculated for symbols that have not already been encountered.

HOL Theorem 6.1.10

$$s \in \texttt{firstSetML}\ g\ sn\ \ell \Rightarrow s \in \texttt{firstSet}\ g\ \ell$$

Proof Function $\texttt{ntderive}\ g\ tok\ \ell$ returns true if and only if tok is in the first set of nonterminals l. List l consists of all the symbols that are encountered in the derivation of $\alpha\ \ tok\ \ \beta$ starting from sentence l such that α is nullable.

HOL Definition 6.1.6 (`ntderive`)

$\texttt{ntderive}\ g\ tok\ \texttt{[]} \iff \texttt{F}$
$\texttt{ntderive}\ g\ tok\ \texttt{[}N\texttt{]} \iff$
$\quad \exists\, pfx\ sfx\ rhs.$
$\qquad \texttt{rule}\ N\ rhs \in \texttt{rules}\ g \land rhs = pfx \mathbin{\texttt{++}} \texttt{[TS}\ tok\texttt{]} \mathbin{\texttt{++}} sfx \land$
$\qquad \texttt{nullable}\ g\ pfx$
$\texttt{ntderive}\ g\ tok\ (N_1\texttt{::}N_2\texttt{::}Ns) \iff$
$\quad \exists\, pfx\ sfx\ rhs.$
$\qquad \texttt{rule}\ N_1\ rhs \in \texttt{rules}\ g \land rhs = pfx \mathbin{\texttt{++}} \texttt{[NTS}\ N_2\texttt{]} \mathbin{\texttt{++}} sfx \land$
$\qquad \texttt{nullable}\ g\ pfx \land \texttt{ntderive}\ g\ tok\ (N_2\texttt{::}Ns)$

The above property serves as an intermediate point between the structure of derivations in `firstSet` and `firstSetML`. Each of the two definitions for first set can be related to `ntderive`, thus establishing a connection between the two.

In the first part, the theorem stated below establishes that in the absence of repeated symbols, tok belongs in the executable first set of the first nonterminal in l.

HOL Theorem 6.1.11

$\texttt{ntderive}\ g\ tok\ ns \land \texttt{ALL\_DISTINCT}\ ns \land$
$\texttt{IMAGE NTS (set}\ ns\texttt{)} \cap \texttt{set}\ sn = \emptyset \Rightarrow$
$\texttt{TS}\ tok \in \texttt{firstSetML}\ g\ sn\ \texttt{[NTS (HD}\ ns\texttt{)]}$

Proof By induction on ns.

In the second part, we establish the following:

HOL Theorem 6.1.12

$(\texttt{derives}\ g)^*\ sf_1\ (\texttt{TS}\ tok\texttt{::}rest) \Rightarrow$
$(\forall\, pfx\ sfx.\ \texttt{nullable}\ g\ pfx \Rightarrow sf_1 \neq pfx \mathbin{\texttt{++}} \texttt{[TS}\ tok\texttt{]} \mathbin{\texttt{++}} sfx) \Rightarrow$
$\exists\, nlist\ pfx\ sfx.$
$\quad sf_1 = pfx \mathbin{\texttt{++}} \texttt{[NTS (HD}\ nlist\texttt{)]} \mathbin{\texttt{++}} sfx \land \texttt{nullable}\ g\ pfx \land$
$\quad \texttt{ntderive}\ g\ tok\ nlist \land \texttt{ALL\_DISTINCT}\ nlist$

Proof By induction on the derivation steps from sf_1 to sf_2.

With the above two theorems we can finally prove HOL Theorem 6.1.10.

6.2 Termination

Authors tend not to discuss termination of functions being used. Similarly, in the case of relations, a statement such as *applying f in succession will result in a state s...* suffices when asserting that one can reach state s eventually. In this section we discuss how the above two cases translate into a theorem prover.

Functions in HOL are defined using the Define (Slind [69]) construct. This has been used for the majority of the definitions in this thesis. When dealing with recursive functions, Define automatically tries to determine the termination conditions for the function and tries to prove it using a termination prover. On the occasion that the termination proof fails one cannot use Define. Instead a different construct called HOL_defn is used to create the requested definition. In this case the termination proof is left to the user. Until such a proof is provided the definition is not 'usable'. The termination of the function is shown by providing a well-founded relation on the arguments of the function. On establishing termination of the function, one can access its definition and the induction principle.

For functions with complex termination arguments, one has to always provide an explicit termination proof. For example, functions such as `nullableML` (in Section 6.1.1) and `firstSetML` (in Section 6.1.2) require a termination proof.

6.2.1 Functions

Removing duplicates from a list is a trivial task. A function for this job will always terminate because lists are finite in HOL. Below is a HOL definition for such a task:

HOL Definition 6.2.1 (rmDupes)

```
rmDupes [] = []
rmDupes (h::t) = h::rmDupes (delete h t)
```

Apart from the knowledge of how `rmDupes` works, just looking at the definition does not tell us that the size of the input list in each recursive call gets smaller. This is because the termination prover does not know that `delete` function *does not* make its arguments longer. Thus, we have to use HOL_defn to make this definition and provide the termination argument.

Clearly in this case the termination is based on the length of the input list in the recursive `rmDupes` call. Hence, the well-founded relation is `measure` $(\lambda\,\ell.\ |\ell|)$.

If multiple arguments are affected differently in a recursive call, it complicates the termination argument required. Let us look at the well-founded relation for `firstSetML` described in Section 6.1.2.

```
measure (λ(g,sn). CARD (nonTerminals g DIFF set sn)) LEX
measure (λsyms. |syms|)
```

`firstSetML` takes three arguments, a grammar g, a seen list of symbols sn and a list of symbols $syms$ in order to calculate the first set of $syms$. The recursive calls to `firstSetML` either decrease, increase or leave the size of $syms$ untouched and for the sn list the recursive calls either leave sn untouched or increase its size by one. In this case it is not immediately obvious what the termination argument should be. A situation like this calls for a lexicographic ordering (`LEX`); some arguments to the function are reduced in some calls and others are reduced in different recursive calls. What we measure for `firstSetML` is a combination of the number of symbols we have encountered (sn) that belong to g and the length of $syms$. At least one of these should decrease in each of the recursive calls.

One aspect of having to show termination is that it may change the 'look' of the definition. In case of both `nullableML` and `firstSetML` we had to add an extra argument (the sn list) to be able to handle the termination argument.

6.2.2 Relations

Another notion of termination is in the context of defining relations in HOL. Let R be a relation such that $R\ s_1\ s_2$ relates the two entities or states such that s_2 is a result of some transformation that s_1 undergoes to result in s_2. If we want to derive a particular state s', starting from state s, such that some predicate P is true of s' by multiple applications of R, then we have to show that $\exists s'.\ R^*\ s\ s' \wedge P\ s'$. This asserts that the function R may eventually terminate to give state s' that satisfies our predicate P. We typically need to show is that there exists a terminating path for the function; not that it terminates on all paths.

For example, the first phase of the mechanisation of Chomsky Normal Form (Section 2.5) involves relation `trans1Tmnl`. Relation `trans1Tmnl` $nt\ \ t\ \ g\ \ g'$ holds if grammar g' is a result of some transformation based on grammar g. To prove that multiple applications of this relation can terminate with a grammar that satisfies the predicate `badTmnlsCount`, we show:

HOL Theorem 6.2.1

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

6.3 The trappings of history

Data structures can have a huge impact on how easy a proof implementation can be in a theorem prover.

For example, we first modeled grammar rules as sets. When we decided to tackle the executable SLR parser generator, we changed them to lists in order to conform more closely to theme of executability underlying our parser. On one hand, using lists meant we now did not have to concern ourselves with providing statements about finiteness. On the other hand, it meant that a lot of definitions, for normal forms, that were modeled as functions had to be changed to relations. The original functions returned a set of rules using set comprehension. In order to still be able to use the handy set comprehension notation, relations had to be used. All in all it turned out to be a fairly even exchange between the proofs we ended up avoiding and the ones we had to recast in terms of lists.

Another example where the impact of choice of data structure is obvious is the proof of the pumping lemma.

6.3.1 Pumping lemma using derivation lists

The nice and compact proof of the pumping lemma presented in Section 4.2 was the result of a second attempt. When the pumping lemma was tackled for the first time, we used derivation lists to represent derivations in the grammar. As we will see in this section that our first approach was fraught with convoluted and complex definitions.

The fundamental problem is that when we use derivation lists getting a handle on a particular subderivation (associated with a nonterminal) becomes a complicated process. The derivation list models derivation one expansion at a time. In such a case, to access a subderivation one needs to provide clauses that ensure that only a particular symbol gets expanded at each step and the remaining symbols in the list remain unchanged. This leads to additional assertions and quantifiers resulting in both complex definitions and proofs.

As we saw with the parse tree approach, using trees can easily subvert such problems. Derivation lists have the advantage that the order in which nonterminals are expanded is apparent. This was particularly important in earlier work on SLR parsing 5, and our theory of such lists is well-developed. Unfortunately, as we shall see in the course of the proof, the proof statements look less intuitive due to the barrage of quantifiers. In particular, the way in which a derivation is actually a concatenation of subtrees (as per Figure 6.1) becomes quite obscured. Derivation streams are to derivation lists what subtrees are to derivation trees.

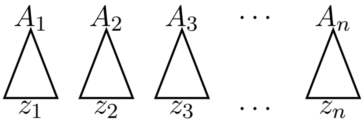


Figure 6.1 – Individual derivation streams for string $A_1A_2A_3\cdots A_n$ deriving $z_1z_2z_3\cdots z_n$

We ended up doing far more complex reasoning about derivation lists than we expected. Nevertheless, it did turn out to be a fruitful exercise in terms of looking at how much proofs blow up based on the initial modeling decisions. Proof reasoning that is straightforward with a few diagrams becomes incredibly complex when mechanised, as indicated by the number of clauses as well as the number of quantifiers in the various lemmas.

The proof follows the same structure as the parse tree approach albeit the statements for the properties and theorems are more bloated. We first derive the existence of a recurring nonterminal in the derivation. We then get a handle on the very last occurrence of such a repetition. Finally, we provide the witnesses for the pumping lemma variables.

6.3.1.1 A nonterminal must recur in the derivation

The first lemma we prove corresponds to Lemma 4.2.2. The lemma states that if k is the number of nonterminals in the grammar and $|z| \geq 2^k$ then there must be a nonterminal which expands to itself within the derivation of z.

We begin the mechanisation by modeling the notion of a derivation containing a repeated symbol, `symRepProp`. This is HOL Definition 6.3.1

The property states that the given derivation list dl has a subderivation such that a nonterminal symbol B expands to a list containing itself, vBw. In our prose, we omit the symbol constructor `NTS` when referring to a nonterminal symbol such as B, or (later) N_1 and N_2. The last clause (the condition on the members of s_0) ensures that the derivation of the sub-list is a result of expanding B. The suffix sfx remains untouched.

The definition of `symRepProp` is captured pictorially using trees in Figure 6.3. Note how the formal definition with its nested lists (lists of sentential forms, that are in turn

HOL Definition 6.3.1 (`symRepProp`)

$$
\begin{array}{l}
\texttt{symRepProp}\ dl \iff \\
\quad \exists p\ tsl\ B\ sfx\ v\ w\ s. \\
\quad\quad dl = p \mathbin{\texttt{++}} [tsl \mathbin{\texttt{++}} [\texttt{NTS}\ B] \mathbin{\texttt{++}} sfx] \mathbin{\texttt{++}} s \land \texttt{isWord}\ tsl \land \\
\quad\quad \exists s_0\ s_1. \\
\quad\quad\quad s = s_0 \mathbin{\texttt{++}} [tsl \mathbin{\texttt{++}} v \mathbin{\texttt{++}} [\texttt{NTS}\ B] \mathbin{\texttt{++}} w \mathbin{\texttt{++}} sfx] \mathbin{\texttt{++}} s_1 \land \\
\quad\quad\quad \texttt{isWord}\ v \land \\
\quad\quad\quad \forall e. \\
\quad\quad\quad\quad e \in s_0 \Rightarrow \\
\quad\quad\quad\quad \exists p_0\ p_1\ nt. \\
\quad\quad\quad\quad\quad e = tsl \mathbin{\texttt{++}} p_0 \mathbin{\texttt{++}} [\texttt{NTS}\ nt] \mathbin{\texttt{++}} p_1 \mathbin{\texttt{++}} sfx \land \texttt{isWord}\ p_0
\end{array}
$$

Figure 6.2 – Definition of `symRepProp`.

lists of symbols), leads to a painfully complicated statement at such an early stage in the proof. Compare this to HOL Definition 4.2.2 which uses parse trees to express the same idea.

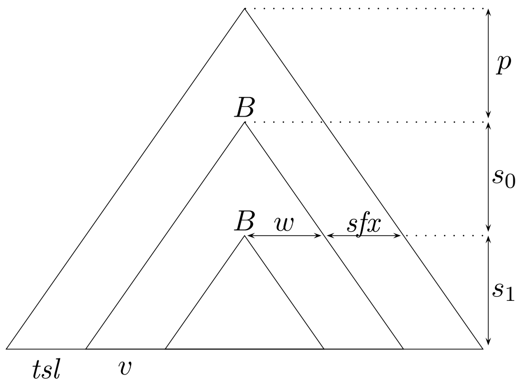


Figure 6.3 – Tree representation of property `symRepProp`. Horizontal lines are sentential forms. Variables p, s_0 and s_1 correspond to a partition of the derivation (a list of sentential forms).

If each derivation stream in dl has no nonterminals that are repeated, the length of the final terminal string is constrained by the number of distinct nonterminals in dl.

HOL Theorem 6.3.1

$$
\begin{array}{l}
\texttt{lderives}\ g \vdash dl \lhd [\texttt{NTS}\ A] \rightarrow z \land k = |\texttt{distinctldNts}\ dl| \land \\
\texttt{isWord}\ z \land \texttt{isCnf}\ g \land \neg\texttt{symRepProp}\ dl \Rightarrow \\
|z| \leq 2 \star\star (k - 1)
\end{array}
$$

(The $\star\star$ operator represents exponentiation. Function `distinctldNts` *dl is the list of nonterminals occurring in dl.)*

Proof This property directly follows from the grammar being in CNF. The proof is by induction on the length of the derivation.

Because the total number of nonterminals in the grammar is at least as large as the number occurring distinctly in any given derivation, if $|z|$ is larger than 2^k, then we know that within the derivation of z, a nonterminal symbol must be repeated in its own derivation stream.

6.3.1.2 Isolating the last repetition of a nonterminal

The next step is to pick the very last instance when the repetition of a nonterminal occurs, *i.e.*, as close as possible to the final terminal string. This corresponds to witnessing the tree structure represented in Figure 6.4. There are two important features of this diagram. Not only do we isolate the last occurrence of nonterminal B, but we also know that the previous occurrence of B expanded to the two nonterminals N_1 and N_2 (again, thanks to the grammar being in CNF).

The last expansion property (HOL Definition 6.3.2) has two parts. The first part `ntProp` describes the concrete expansions in the tree (that $B \Rightarrow N_1 N_2$ at this point).

Secondly, we state that the derivation below B can also be divided into subderivations to get the individual streams; $N_1 N_2$ to vBw, which is the definition's dl_1; vBw to $v\,rst$ (dl_2); and sfx to rst' (dl_3). Because this is the very last occurrence of a nonterminal being repeated, `symRepProp` does not hold in any of the derivation streams below the last but one occurrence of B. This corresponds to the dashed triangle in Figure 6.4.

HOL Definition 6.3.2 (`lastExpProp`)

$$\begin{array}{l}
\texttt{lastExpProp}\ (g, dl, z) \iff \\
\quad \exists\, p\ s\ tsl\ B\ sfx\ N_1\ N_2. \\
\qquad \texttt{ntProp}\ dl\ p\ s\ tsl\ B\ sfx\ N_1\ N_2 \land \\
\qquad \exists\, dl_1\ dl_2\ dl_3\ v\ w. \\
\qquad\quad \texttt{lderives}\ g \vdash dl_1 \lhd \\
\qquad\qquad\quad [\texttt{NTS}\ N_1;\ \texttt{NTS}\ N_2] \rightarrow (v \mathbin{++} [\texttt{NTS}\ B] \mathbin{++} w) \land \\
\qquad\quad \exists\, rst. \\
\qquad\qquad \texttt{lderives}\ g \vdash dl_2 \lhd v \mathbin{++} [\texttt{NTS}\ B] \mathbin{++} w \rightarrow (v \mathbin{++} rst) \land \\
\qquad\qquad \exists\, rst'. \\
\qquad\qquad\quad \texttt{lderives}\ g \vdash dl_3 \lhd sfx \rightarrow rst' \land \\
\qquad\qquad\quad z = tsl \mathbin{++} v \mathbin{++} rst \mathbin{++} rst' \land \\
\qquad\qquad\quad \texttt{distElemSubset}\ dl\ dl_1 \land \texttt{distElemSubset}\ dl\ dl_2 \land \\
\qquad\qquad\quad \texttt{distElemSubset}\ dl\ dl_3 \land \\
\qquad\qquad\quad (\forall\, e.\ e \in dl_1 \Rightarrow tsl \mathbin{++} e \mathbin{++} sfx \in dl) \land \\
\qquad\qquad\quad (\forall\, e.\ e \in dl_2 \Rightarrow tsl \mathbin{++} e \mathbin{++} sfx \in dl) \land \\
\qquad\qquad\quad \neg\texttt{symRepProp}\ dl_1 \land \neg\texttt{symRepProp}\ dl_2 \land
\end{array}$$

```
        ¬symRepProp (dl1 ++ TL dl2)
```

(Condition `distElemSubset` *dl dl' denotes* `distinctldNts` *dl' is a subset of* `distinctldNts` *dl.)*

The `distElemSubset` conditions, and those on the structure of elements that belong in dl_1 and dl_2, are inductive invariants, and typical examples of the sort of implementation detail one glosses over when following a textbook proof.

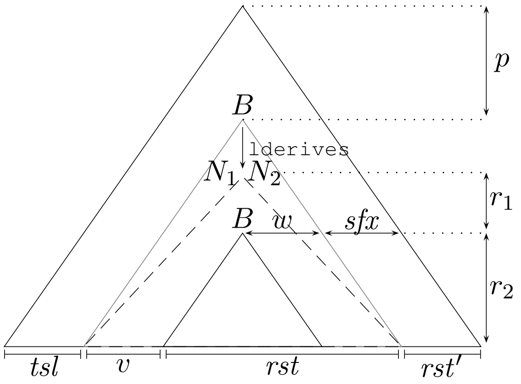


Figure 6.4 – Tree representation of properties `lastExpProp` and `ntProp`. We assume that there are no further occurrences of B in the triangle below the second B. p, $r1$, $r2$ correspond to splitting the derivation corresponding to the definition of `ntProp`.

HOL Definition 6.3.3 (`ntProp`)

```
  ntProp dl p s tsl B sfx N1 N2 ⟺
    dl =
      p ++ [tsl ++ [NTS B] ++ sfx] ++
      [tsl ++ [NTS N1; NTS N2] ++ sfx] ++ s ∧ isWord tsl ∧
    (N1 = B ∨
     ∃r1 r2 v w.
       s = r1 ++ [tsl ++ v ++ [NTS B] ++ w ++ sfx] ++ r2 ∧
       isWord v)
```

Property `ntProp` states that the very next expansion from B has to be a nonterminal one. In addition it also asserts that either B is part of the above expansion or occurs later on in the derivation.

We can now prove that if `symRepProp` holds of the original derivation list dl, then we should be able to deduce that `lastExpProp` also holds:

HOL Lemma 6.3.1

$$\begin{array}{l}
\mathtt{lderives}\ g \vdash dl \lhd z_0 \rightarrow z \land \mathtt{isCnf}\ g \land \mathtt{isWord}\ z \land \\
z_0 = pz_0 \mathbin{+\!\!+} sz_0 \land \mathtt{isWord}\ pz_0 \land \mathtt{EVERY\ isNonTmnlSym}\ sz_0 \land \\
\mathtt{symRepProp}\ dl \Rightarrow \\
\mathtt{lastExpProp}\ (g, dl, z)
\end{array}$$

Proof By induction on dl.

This and all of the other significant lemmas depend on the following result:

HOL Lemma 6.3.2 *If a grammar is in CNF, then for all derivations in that grammar, if the first sentential form is composed of all terminals followed by nonterminals than the rest of the elements in the derivation will also satisfy this property.*

The corresponding HOL statement is

HOL Theorem 6.3.2

$$\begin{array}{l}
\mathtt{lderives}\ g \vdash dl \lhd x_1 \mathbin{+\!\!+} x_2 \rightarrow y \land \mathtt{isCnf}\ g \land e \in dl \land \\
\mathtt{isWord}\ x_1 \land \mathtt{EVERY\ isNonTmnlSym}\ x_2 \Rightarrow \\
\exists p\ s.\ e = p \mathbin{+\!\!+} s \land \mathtt{isWord}\ p \land \mathtt{EVERY\ isNonTmnlSym}\ s
\end{array}$$

6.3.1.3 Splitting z into components

We now split the derivation streams even further. This will allow us to obtain the individual components of the pumping lemma (u, v, w, x and y). For this we need the following lemma.

Property `splitDerivProp` relates the two individual derivation streams (dl_1 and dl_2) to the original derivation (dl). Among the properties preserved, we have

- Length of each derivation stream is less than or equal to the original one;
- The nonterminals in dl_1 and dl_2 are a subset of those in dl;
- The elements of dl_1 get expanded first. The dl_2 part remains constant. When dl_1 has expanded to a terminal string (x'), then the expansion of dl_2 begins. We are assuming that all derivations are leftmost. Thus, elements of dl_1 concatenated with the unexpanded start string of dl_2 (y), followed by x' concatenated with dl_1 gives us back our original derivation list dl.
- The number of distinct nonterminals in dl_1 and dl_2 is not greater than that of dl.

HOL Definition 6.3.4 (`splitDerivProp`)

```
splitDerivProp (g,dl,v) (dl₁,x,x′) (dl₂,y,y′) ⟺
  v = x′ ++ y′ ∧ lderives g ⊢ dl₁ ◁ x → x′ ∧
  lderives g ⊢ dl₂ ◁ y → y′ ∧ |dl₁| ≤ |dl| ∧ |dl₂| ≤ |dl| ∧
  distElemSubset dl dl₁ ∧ distElemSubset dl dl₂ ∧
  dl =
    MAP ((λe ℓ. addLast ℓ e) y) dl₁ ++
    MAP (addFront x′) (TL dl₂) ∧ ¬symRepProp dl₁ ∧
  ¬symRepProp dl₂ ∧ distElemLen dl dl₁ ∧ distElemLen dl dl₂ ∧
  |distinctldNts dl₂| ≤ |distinctldNts dl|
```

(The `addLast` ℓ e *function adds* e *at the end of each element of* l *and then* `addFront` e ℓ *adds* e *at the start of each element of* l*.)*

Lemma 6.3.1 *Let* dl *be a derivation where no nonterminal symbol gives rise to itself. We can divide* dl *into individual derivation streams such that* `splitDerivProp` *is always preserved.*

Proof By induction on dl.

HOL Lemma 6.3.3

```
lderives g ⊢ dl ◁ x ++ y → v ∧ isCnf g ∧ isWord v ∧
pfx ++ sfx = x ++ y ∧ isWord pfx ∧ EVERY isNonTmnlSym sfx ∧
¬symRepProp dl ⇒
∃dl₁ dl₂ x′ y′. splitDerivProp (g,dl,v) (dl₁,x,x′) (dl₂,y,y′)
```

6.3.1.4 Putting it all together

We can now bring together the proof for pumping lemma using the elements described above. Figure 6.5 shows what each of the quantifiers in the pumping lemma proof get instantiated to.

The figure shows the derivation of terminal string z from the start symbol S of some grammar g. B is the symbol that gets repeated in its own derivation closest to z. The triangles for B show the very last occurrence of any nonterminal that expands to itself. B takes a single step to expand to the two nonterminals N_1 and N_2 which eventually expand out to the terminal string vwx. There are no repeated nonterminals in the expansion that follows from N_1N_2. The components u, v, w, x and y split up z so that they satisfy the pumping lemma clauses.

The description of the proof follows.

Clause 1 $|vx| \geq 1$.

Proof We have $B \overset{l}{\Rightarrow}^* vBt \overset{l}{\Rightarrow}^* vwx$, where v, w, x are terminal strings. Since the grammar is in CNF, it follows that either v or t must be nonempty. If v is nonempty then we are done.

From the two facts, $t \neq \epsilon$, and $t \overset{l}{\Rightarrow}^* x$, again we use the properties of CNF to deduce that x must be nonempty since CNF excludes ϵ-productions.

Clause 2 $|vwx| \leq 2^k$, where k is the number of nonterminals in the grammar.

Proof With $|z| \geq 2^k$, we use HOL Lemma 6.3.1 to get `symRepProp` dl.

From HOL Lemma 6.3.1, we have `lastExpProp` (g, dl, z). This gives us a handle on the structure of the tree as shown in Figure 6.5.

From this structure we are able to deduce, $B \overset{l}{\Rightarrow} N_1N_2 \overset{l}{\Rightarrow}^* vwx$.

Using HOL Lemma 6.3.3, we get the following facts. There must exist strings m_1 and m_2 such that $vwx = m_1m_2$ and $N_1 \overset{l}{\Rightarrow}^* m_1$ and $N_2 \overset{l}{\Rightarrow}^* m_2$ such that no nonterminal symbol is repeated in the derivations of m_1 and m_2. This corresponds to the area of the dashed triangle, which covers all the expansion under the top B except the one step derivation to N_1N_2.

Since no symbols are repeated, from HOL Lemma 6.3.1, we get the bound on the sizes of m_1 and m_2. Let d_1 be the number of unique nonterminals in the derivation of m_1 and d_2 be the number of unique nonterminals in the derivation of m_2. Then, $|m_1| \leq 2^{d_1-1}$ and $|m_2| \leq 2^{d_2-1}$. We also know that, $d_1 \leq k$ and $d_2 \leq k$.

Therefore $|vwx| \leq 2^k$.

Clause 3 uv^iwx^iy is in L for every natural number i.

Proof By the application of HOL lemmas 6.3.1 and 6.3.3 we have the shape of the tree as shown in Figure 6.5, and the following derivations: $S \overset{l}{\Rightarrow}^* uBy$, $B \overset{l}{\Rightarrow}^* vwx$, and $B \overset{l}{\Rightarrow}^* vBx$.

The applications of the above expansions give us the following two possibilities to satisfying this clause.

- $S \overset{l}{\Rightarrow}^* uBy \overset{l}{\Rightarrow}^* uwy$, for $i = 0$.
- $S \overset{l}{\Rightarrow}^* uBy \overset{l}{\Rightarrow}^* uvBxy \cdots \overset{l}{\Rightarrow}^* uv^iwx^iy$, for $i \geq 1$.

Table 6.1 shows the statistics for the two approaches. These numbers along with the experience of mechanisation confirms the simplicity of the second approach using parse trees. Even though the number of definitions used in both approaches is the same, for the parse tree approach the size is smaller and the definitions are more readable. The common definition is that of `lastExpProp`. The extra definition in the derivation list corresponds to breaking down a derivation into subderivations. Definitions such as

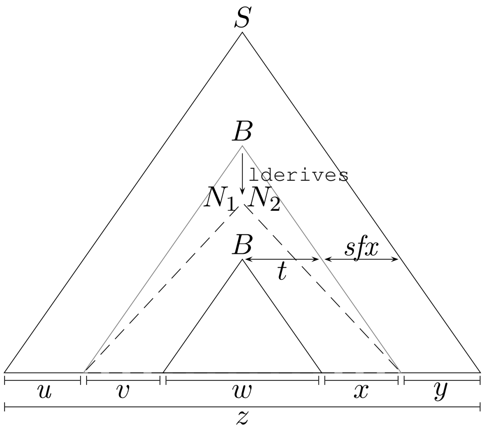


Figure 6.5 – Tree showing the witnesses for the existential quantifiers in the pumping lemma.

`symRepProp`, `rootRep`, etc are part of the library theories. We observed a reduction of around 80% for both the size of the mechanised proofs and for the number of proofs when we used parse trees to mechanise the pumping lemma.

Mechanisation approach	LOC	#Definitions	#Proofs
Derivation lists	2500	2	26
Parse trees	520	1	5

Table 6.1 – Summary of the mechanisation effort for pumping lemma

The above numbers exclude library proofs and the general framework for CFGs. In contrast, the proof in Hopcroft and Ullman is about two pages long.

6.4 Finiteness

When we deal with sets in a textual proof, it is common to assume that certain operations are permissible with respect to them. A simple one is that of picking an element that does not belong in a particular set. This is quite a common thing to do even though there are usually two implicitf assumptions in place. The first one is that the particular set is finite, and the second is that the universal set of elements is infinite.

A mechanised proof of *pick a new element not in set...* requires the two conditions stated above. The infinite nature of the universal set features as an assumption to all the proofs that require introducing a new element in some manner. The finiteness of a given set usually has to be shown explicitly.

6.4.1 **aProds** lemma

In Section 2.6.1 we presented the aProds lemma, which we can use to eliminate the leftmost nonterminal of the right-hand side of a production.

Once again, the rules of the new grammar are given by the following function in HOL:

```
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 language equivalence theorem

```
aProds A ℓ g g′ ⇒ L g = L g′
```

is vacuously true if the condition aProds A ℓ g g' does not hold. To avoid this we prove the following theorem.

HOL Theorem 6.4.1

```
A ∉ ℓ ⇒ ∀g. ∃g′. aProds A ℓ g g′
```

On the surface this seems like an easy problem. The solution could be obtained by instantiating g' to be a grammar whose rules have been chosen using aProdsRules. This does not work immediately. The reason is that the rules of g is a *list* of rules while aProdsRules returns a *set*. A set can have a corresponding list if and only if the set is finite. Once we establish the finiteness of aProdsRules A ℓ ru, we can construct a corresponding list of rules ru'. Now g' can be instantiated as the grammar G ru' (startSym g) giving us the desired theorem.

For all this to work we need to prove:

HOL Theorem 6.4.2

```
FINITE ru ⇒ FINITE (aProdsRules A ℓ ru)
```

Once again we tackle the obvious: *yes*, the rules in the set aProdsRules A ℓ ru are finite because they are constructed from rules of g which is modeled as a list. But this sequence of reasoning is beyond the capability of a theorem prover's automatic tools.

One way of proving sets to be finite is to represent them as a union of sets that are the result images of a simpler function, *i.e.* if we want to prove $\texttt{FINITE}\ s$ then we express the set s as $\texttt{BIGUNION (IMAGE}\ f\ s'\texttt{)}$ for a some function f and some finite set s'. Function `BIGUNION` is defined as:

HOL Theorem 6.4.3 (BIGUNION)

$$\texttt{BIGUNION}\ P = \{x \mid \exists s.\ s \in P \land x \in s\}$$

The new expression is finite because it works on the finite set s'. Now we have reduce the problem of proving finiteness to establishing two statements.

First is showing equality between s and $\texttt{BIGUNION (IMAGE}\ f\ s'\texttt{)}$. Second is showing that the application of f on the elements of set s' results in finite only finite sets. From this and the theorem below

HOL Definition 6.4.1

$$\texttt{FINITE}\ s \Rightarrow \forall f.\ \texttt{FINITE (IMAGE}\ f\ s\texttt{)}$$

we can prove $\texttt{FINITE (IMAGE}\ f\ s'\texttt{)}$. We then use the following theorem

HOL Theorem 6.4.4

$$\texttt{FINITE (BIGUNION}\ P\texttt{)} \iff \texttt{FINITE}\ P \land \forall s.\ s \in P \Rightarrow \texttt{FINITE}\ s$$

to get the result that $\texttt{FINITE (BIGUNION (IMAGE}\ f\ s'\texttt{))}$ *i.e.* $\texttt{FINITE}\ s$.

Let M stand for $\texttt{aProdsRules}\ A\ \ell\ ru$. The expression for proving M finite requires constructing a series of simpler functions that can be used to reconstruct M in the following way.

We first come up with the function f.

```
$f$ =
  ($\lambda r$.
     case $r$ of
       rule $N$ $rhs$ $\to$
         if $N \neq A$ then
           $\emptyset$
         else
           {rule $A$ ($x$ ++ $r_0$) |
            $\exists B.\ B \in \ell \land$ rule $B$ $x \in ru \land rhs$ = NTS $B$::$r_0$})
```

Now f can be used to express the original expression M. We can represent M as BIGUNION (IMAGE f ru) for some ru. Let ru' be the set used in the else-branch of function f. Obviously the empty set returned in the if-branch is finite. Now all we have to show is that the set ru' is finite as well. We do this by simplifying f even further to obtain the function g.

```
g =
  (λ r.
     case r of
        rule M rr →
          if M ∈ ℓ then
            {rule A (rr ++ r₀) | l′ = NTS M::r₀}
          else
            ∅)
```

From this we can state that $ru' =$ BIGUNION (IMAGE g ru). The result of function g can be easily proven as finite by application of HOL tactics. We can then propagate the result to prove the result of f as finite and finally the expression M as finite.

6.5 The HOL issue

There is no doubt about the fact that using software systems simplifies a lot of things but such systems come with their own set of problems, endemic to developing and evolving any system.

As an open source system, HOL is constantly being updated. This has its own pros and cons. The major advantage of this is the easy access to new definitions, better proof resolution techniques and new capabilities. To access the latest facilities in HOL, one has to update the system. The problems with constant updates are it tends to break previously proven goals. In some sense a proof in a theorem prover is only as good as the version it works for! We had to spend considerable time trying to fix proofs whenever system updates were made, updates that were necessary to access new pretty printing functionality. Currently HOL employs a way whereby one can prevent the use of new simplifier components which have been affected in the new version. It solves most, but not all, of the problems.

HOL has evolved a lot especially in the area of typesetting HOL elements: terms, theorems and definitions. In the beginning, it was just a copy-paste-format job. A lot of time had to be spent ensuring consistency of format between the different HOL elements. Not to mention if one decided to simply change a font, all the elements had to be manually adjusted. This, obviously, is not a user-friendly way of being able to share certified proofs. Then a better version with a nice pretty printer version came out that allowed one to write HOL elements , formatted for LaTeX, to files and then include them in documents such as LaTeX. This is great because now the formatting is done by HOL itself and can also be overridden. It is amazing how much consistent typesetting of HOL elements can reduce the work by a factor of more than a half. The drawback of this approach is that one has to run a script every time proofs change. Also, if one wants to remove quantifiers and such from the theorems, this requires editing the proof files by hand.

The latest HOL goes one better on this process and provides the *munger* technology. Theorems and terms can be directly referenced from a pre-LaTeX file. This file along with the compiled theories is used to spit out LaTeX files which have the theorems and terms substituted in. All one needs are the latest compiled theories. Now writing up discussion of HOL proofs in LaTeX is a piece of cake. This has saved us quite a few days of getting LaTeX typesetting 'just right' for this thesis.

Despite the cons, provers definitely have potential as proof validating and publishing system and complete the picture of proof development as we have always known it: assert a statement, prove it and then share the results in the community. Broadly speaking, the sharing should be possible between and within three groups of people. First, within the community using a particular system, easily achieved and which already exists. Second, sharing of results with an external audience. This is achieved by the system providing some kind of proof publishing facility by exporting proofs to a more readable format like LaTeX. And last but not the least is the ability to share proofs across different provers. One should be able to translate or export proofs from other systems easily. Since mechanised proofs are time consuming such sharing would be of immense help. This is not a novel idea and work on such a pairing already exists. Harrison and Théry in [30] explore how to harness the individual strengths of HOL and the computer algebra system Maple. Their approach is based on separation of proof discovery and proof checking between the two systems. Similar works have been done on integrating Maple with provers such as Isabelle (Ballarin et al. [5]) and PVS (Dunstan et al. [19]). Gordon *et al.* [25] describe how to link HOL with the first-order theorem prover, ACL2. This synthesis is aimed at verifying large hardware and software models. Obua *et al.* [53] have developed an importer from HOL4 and HOL Light into Isabelle/HOL which allows replaying of HOL proofs inside Isabelle/HOL. Along similar lines, McLaughlin [45] has defined an interpretation of the Isabelle/HOL logic in HOL Light. In spite of these works, the swapping of proof results between theorem provers is not an established practice yet.

6.6 Changing the tool

Isabelle (Paulson [57]) is a generic proof assistant that allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. It is developed at University of Cambridge by Larry Paulson, Technische Universität München by Tobias Nipkow and Université Paris-Sud by Makarius Wenzel.

We used Isabelle to formalise some of the simplification procedures for context-free grammars. The statistics are summarised in Table 6.2.

Mechanisation	LOC		#Definitions		#Proofs	
	Isabelle	HOL	Isabelle	HOL	Isabelle	HOL
ϵ-productions	291	299	4	5	12	20
Generating symbols	164	174	2	2	13	13
Reachable symbols	96	239	2	2	8	21
Unit productions	430	785	6	7	24	45
Chomsky Normal Form	1408	1438	13	13	71	96

Table 6.2 – Comparison of the mechanisation effort between HOL and Isabelle

The size of the proofs is almost the same except in the case of elimination of non-reachable symbols and elimination of unit productions. This is because in HOL the rules for the derived grammars which contain no non-reachable symbols and no unit productions are modeled as sets. Since sets can be infinite, one has to explicitly provide that the newly created rule sets are finite leading to extra proofs. This finiteness on the rule sets is required to establish the existence of the relation that asserts the elimination property. The biggest advantage of using sets is access to set comprehension notation which gives succinct and readable definitions. In Isabelle, we model the rules as lists. The reason for this is the existing support for list comprehension. Using list comprehension one already has the grammar rules as finite thus avoiding the numerous finiteness proofs. Using lists also simplifies the definitions and one can get rid of all the auxiliary definitions required when set comprehension is used. This has given us reduced number of definitions and theorems.

What differs between HOL proofs and Isabelle proofs is the extent to which the proofs are readable to the users. HOL uses a tactical style of proving statements. In a tactical style one applies the inbuilt tactics in some combination to achieve the final result.

Along with a tactical language for proofs, Isabelle also has what is called the *Isar* language. The statements in Isar are closer to those in a text proof. As an example let us consider a simple property of the *derives* relation for context-free grammars. If we have $u \Rightarrow_g^* v$ and $w \Rightarrow_g^* x$ then we can conclude $uw \Rightarrow_g^* vx$. In HOL, the proof is expressed using the tactical style as follows:

```
val derives_append = store_thm ("derives_append",
  ‘‘RTC (derives g) M N ∧ RTC (derives g) P Q ⇒
    RTC (derives g) (M ++ P) (N ++ Q)’’,
   METIS_TAC [rtc_derives_same_append_right,
              rtc_derives_same_append_left,
              RTC_RTC]
```

In Isabelle the same proof is expressed using the Isar language as follows:

```
lemma derives_append:
assumes uv: "(derives g)^** u v"
and     wx: "(derives g)^** w x"
shows "(derives g)^** (u @ w) (v @ x)"
proof -
  from uv have "(derives g)^** (u @ w) (v @ w)"
    by (rule rtc_derives_same_append_right)
  also from wx have "(derives g)^** (v @ w) (v @ x)"
    by (rule rtc_derives_same_append_left)
  finally show ?thesis .
qed
```

For this particular example using the tactical style gives a smaller proof. However, using a series of such tactical statements can easily result in an incomprehensible proof script. When the proof is done in Isar, one has to still provide the required framework before proceeding with the proof. This leads to extra lines of code. But this is a small price to pay when using Isar makes the proofs easier to understand and modify. With Isar style proofs, all inferences including instantiation of variables have to be done explicitly. On one hand this makes for robust theorem proving but on the other hand going from HOL to Isabelle requires that some steps that are ‘magically’ solved by HOL tactics need to be provided upfront in case of Isabelle. In HOL, variables can be instantiated both by the system and the user.

Isabelle also comes with a neat suite of tools to help with theorem proving. Sledgehammer (Paulson and Blanchette [58]) is a tool that can be used to automatically deduce proofs for simple goals. The tool gives back the right tactic to apply at a particular point either to solve the goal completely or just to simplify it. It can also give back an Isar style expression instead of the tactical style. Since Sledgehammer searches for the solution using the existing libraries, one can use it as a ‘search’ tool to find relevant theorems. This is particularly helpful when the user has experience with a different theorem proving system. In such cases, the user can easily guess what kind of theorems *may* exist in the libraries based on their knowledge of other theorem provers but may not know what they are called in Isabelle. Another useful tool is Nitpick (Blanchette and Nipkow [10]), a counter-example generator. Nitpick runs in the background when Isabelle is being used. If one enters a goal for which Nitpick can detect counter-examples, Isabelle displays that the theorem is incorrect and provides the values that make it false. Theorem provers have a varying degree of user support which can make a huge difference to how much effort is required for automating proofs.

6.7 Conclusions

Over the course of discussing the formalisation of various proofs, we have provided some statistics to show how big the proofs are when compared to their text versions. In [76], Wiedijk describes a criteria called the de Bruijn factor. This is the *loss factor* when translating the original text proofs into a formal system. This factor is determined by the amount of detail present in the original text and the expressivity of the system that has been used to do the formalisation. The *apparent* de Bruijn factor is the ratio of the size (in bytes) of the formal proof to the size of the text proof. The *intrinsic* de Bruijn factor is the ratio of the size of the compressed formal proof to the size of the compressed text proof. Suprisingly, their isn't much difference between the values of the apparent and the intrinsic factors. The main (and also the more involved proofs) covered in our work use a lot of diagrammatic explanation in the original text, Hopcroft and Ullman. For example, the proofs of Greibach Normal Form, the pumping lemma and SLR parsing rely a lot on graphical intuition. As such we have omitted the calculation of the de Bruijn factor for the formalisation presented in this thesis.

This chapter has highlighted some of the concerns encountered during the mechanisation process. *So what can we do about it?* Establishing libraries which allow mechanisation of more abstract concepts can help overcome some of these issues. These problems are very domain specific and require solutions aimed at particular problems as was also the observation by Wiedijk [77]. Even within our own mechanisation, which was restricted to context-free languages, proofs of finiteness and termination had to be done on a case-by-case basis. Also, making proof steps fine enough so that a theorem prover can make connections requires a good understanding of both the text proof and the system, again not a job a machine could do easily. Thus, when automation is based on conceiving the outline of a complicated argument, the proofs require a lot of direction from the user.

Chapter 7

Conclusions

> The argument goes something like this: ‘I refuse to prove that I exist,’ says God, ‘for proof denies faith, and without faith I am nothing.’
> ‘But,’ says Man, ‘The Babel fish is a dead giveaway, isn’t it? It could not have evolved by chance. It proves you exist, and so therefore, by your own arguments, you don’t. QED.’
> ‘Oh dear,’ says God, ‘I hadn’t thought of that,’ and promptly vanished in a puff of logic.
> ‘Oh, that was easy,’ says Man, and for an encore goes on to prove that black is white and gets himself killed on the next zebra crossing.
>
> Douglas Adams

Contents

We have provided a formalisation of the theory of context-free languages and an application in the form of a verified SLR parser generator. The work presented in this thesis is aimed at providing foundational automated theories for further work in language theory for verifying existing proofs and for possibly developing new verified proofs. The experience has been quite an enlightening one, with regard to both the theorems proved and the gap proofs that were tackled. The granularity of reasoning that is characteristic of a theorem prover forces the user to be rigorous in their own approach when attempting proofs. The user has to be adept at synthesising the textual reasoning and the machine reasoning. Thus, a deep knowledge of both areas is a must, which clearly takes a lot of time and effort but the verified verified theory will stand the test of time. Proofs verified with respect to a theoretical framework will always be true for that particular framework. An incorrect framework doesn't invalidate the proofs themselves: the proofs in such a case are just not useful to us. In mechanising the theory of context-free languages we had first hand experience of this. The first attempt at the relation for conversion from a PDA to a CFG (covered in Section 3.3.2) did not contain all the necessary clauses. We were still able to prove the equivalence property. The problem came out only when we tried to prove that the newly generated rules for the grammar are finite. This highlights another overlapping aspect of using a theorem prover as a checker for debugging a proof. Using an incorrect framework to prove a statement will lead to one of the following two cases, either one gets stuck at a particular point or the problem is easily traceable to the source. Tracing can be hard when dealing with complex textual proofs. The process of correcting underlying definition will most likely break a lot of previously proven conjectures which means more time investment. But rigour comes at an expense and that is time. We hope the time we have invested in formalisation of basic concepts for CFGs and PDAs makes further automation in this area easier.

In Chapter 2, we discussed the formalisation of the types and definitions to represent context-free grammars in HOL. Once this ground work has been done, we go on to prove the property that simplifying CFGs in particular ways has no affect on the generated language. These results allow us to assume a simpler grammar form in the subsequent chapters and helps simplify many of the complicated proofs. An alternate formalism of recognising context-free languages are pushdown automata. Thus, the class of languages accepted by PDAs is precisely the class of context-free languages. Pushdown automata play a key role in compiler design and at times provide a neater approach for properties such as closure under inverse homomorphism. We discussed the mechanisation of PDA in Chapter 3. These two formalisms, CFGs and PDA, give us a good basis for approaching proofs of closure properties and pumping lemma, which was discussed in Chapter 4. The above three chapters have presented the basic mechanisation of the theory. The background mechanisation, for any field, is a one off job. Once completed, one can draw on these results and apply them to areas such as parsing. In Chapter 5, we did just that and discussed the mechanisation of an SLR parser generator. Among other results, we presented proofs for soundness and correctness of the parser generator.

The 'basic' theory mechanisation is basic only in the sense that it is foundational work. The actual work of mechanisation is far from being basic and as discussed throughout the thesis has been a demanding and time consuming process. We synthesised our experiences and presented them in Chapter 6. Some of these highlight how truly interactive the process of theorem proving has to be. Some of the concerns such as choice of data structure occurs in any programming area. The user always has to make such decisions. The other concerns such as readability *vs.* executability, and proofs of finiteness and termination, are a natural artifact of how even at the best of times our deduction steps, when written or explained, have gaps. When we prove a statement textually, it suffices to convince the reader of the logical flow of the argument. Because we reason at such a high level and draw on numerous sources of knowledge, some points get inevitably taken for granted. In case of an automatic system, these are exactly the points that need to be explicitly spelt out. Dana Scott sums it up nicely in [77], "Algebra is smarter than you are! By which I mean that the laws of algebra allow us to make many steps which combine information and hide tracks after simplifications". This is equally true of any other area of reasoning.

The theorem provers are constantly evolving to be better at automatic proof checking and usability. Both of these affect how widely such systems are used, be it for industry, research or just teaching. Our aim has been to provide tools, techniques and infrastructure for context-free languages. Hopefully, from here on, proof development in this area can be an easier task. We have provided a large chunk of the foundational work for context-free languages. Still, many exciting avenues remain for extending this work.

Going from text to automated text is an arduous task of going deeper into the details. Intuitive inference, so common to our own argument style, is hard to teach to a machine. We hope to reach a stage stage where reasoning expressed at a level of a rigorous proof suffices to convince the theorem prover of its validity.

At the end of the day (or more precisely the years it took to accomplish this work), as John Harrison aptly puts it, "formalized mathematics is quite addictive (at least, for a certain kind of personality)" [31], and we agree.

7.1 Future work

At the end of it all *there are always things left unsaid and more places to go*. Some avenues for extending our current work are presented below.

- ◇ A pedagogical application of this work would be to develop usable and interactive expert assistants for proofs that usually are the trouble spots when it comes to teaching. Such assistants can be a tremendous teaching aid in disambiguiating between the correct and the incorrect proof methods. They can help clarify the proof structure in an environment where a student is forced to follow *a* correct series of inferences to prove a statement.
- ◇ An interesting set of proofs that will complete the theory are the undecidability results for CFLs. For example, deciding whether the intersection of two CFLs is empty or not.

- We have looked at the most basic form of context-free grammar. A possibility would be to look at extensions to the theory of context-free grammars to be able to handle other variations such as stochastic context-free (a special form of weighted CFGs) that have applications in areas as diverse as natural language processing to the modeling of RNA molecules.

- We mostly use relational style for defining various transformations. This means that our algorithms such as those for simplifying a grammar are not executable. A possibility would be to explore changing such definitions to be in a functional form so that they can be executed and looking at how proofs based on functional definitions scale.

- A proof of Ogden's lemma, a stronger version of the pumping lemma that allows one to focus on a small number of positions in the string and pump them. This extension, easy for regular sets, is much harder to obtain for CFLs.

- The theory mechanised in this thesis has a lot of applications. An area of extension would be to apply the mechanised theory to well-known algorithms such as the Earley parser and CYK. Implementations of Earley parser exist in Perl, C, Python and Java libraries.

- The majority of our work on parsing is concerned with proving the correctness and completeness of the algorithm and presenting an executable counterpart. A sizable chunk of work has been to implement the infrastructure for the parser generator before we could achieve the above goals. The future work is to extend the proofs to demonstrate the decidability of the SLR language, *i.e.* to show that the parser terminates on all inputs, not just on inputs in the language. The parser should throw an error for strings *not* in the language. Another area is to improve the efficiency of the parser. Currently, the DFA states are computed on the fly. Changing this to be computed statically will enhance the performance of the parser.

  For the sake of simplicity, we have dealt with SLR parsers. However, LALR parsers are the ones generated by compiler-compilers such as yacc and GNU bison and used by real-world computer languages. We envisage that the work on SLR will scale up to LALR. The key functions will be the same for LALR. However, instead of follow sets, LALR uses lookahead sets, which are more specific as they take more of the parsing context into account allowing finer distinctions than the follow set. It will be interesting to see to what extent our current work on SLR can assist us in verifying an LALR parser generator.

  To summarise, the extensions for the work on SLR parser would cover:

  - proof of decidability;

  - runtime efficiency; and
  - extension of SLR parser generator and corresponding proofs of soundness and correctness to more widely used LALR parsers.

Bibliography

[1] ACL2 Theorem Prover. ACL2 website. `http://www.cs.utexas.edu/users/moore/acl2/`.

[2] J. S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive Theorem Proving: An Empirical Study of User Activity. *Journal of Symbolic Computation*, 25:263–284, 1995.

[3] Jeremy Avigad. Mathematics in Isabelle. `http://www.andrew.cmu.edu/user/avigad/isabelle/`.

[4] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9(1):2, 2007. ISSN 1529-3785.

[5] Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and algorithms: an interface between Isabelle and Maple. In *ISSAC '95: Proceedings of the 1995 international symposium on Symbolic and algebraic computation*, pages 150–157, New York, NY, USA, 1995. ACM. ISBN 0-89791-699-9.

[6] 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. ISBN 3642005896.

[7] Aditi Barthwal and Michael Norrish. A Formalisation of the Normal Forms of Context-Free Grammars in HOL4. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 95–109. Springer, August 2010.

[8] Aditi Barthwal and Michael Norrish. Mechanisation of PDA and Grammar Equivalence for Context-Free Languages. In Anuj Dawar and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 17th International Workshop, WoLLIC 2010*, volume 6188 of *Lecture Notes in Computer Science*, pages 125–135, 2010.

[9] Jérémy Blanc, J.P. Giacometti, André Hirschowitz, and Loïc Pottier. Proofs for freshmen with Coqweb. In H. Geuvers and P. Courtieu, editors, *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, June 2007. URL `http://www.cs.ru.nl/~herman/PUBS/proceedingsPATE.pdf`.

[10] Jasmin Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin/Heidelberg, 2010.

[11] Jasmin Christian Blanchette. Proof Pearl: Mechanizing the Textbook Proof of Huffman's Algorithm. *J. Autom. Reasoning*, 43(1):1–18, 2009.

[12] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal Verification of a C Compiler Front-End. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006. URL `http://gallium.inria.fr/~xleroy/publi/cfront.pdf`.

[13] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150, London, UK, 1997. Springer-Verlag. ISBN 3-540-63141-0.

[14] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[15] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959. ISSN 0019-9958.

[16] N. Chomsky. Context-free grammars and pushdown storage. Technical report, MIT Research Laboratory Electronics, 1962.

[17] CoqWeb: A web interface for Coq theorem prover. CoqWeb website. `http://coq.inria.fr/cocorico/CoqInTheClassroom`.

[18] Judicaël Courant and Jean-Christophe Filliâtre. Beginning of formal language theory, 1993. Available from `http://coq.inria.fr/contribs-eng.html`.

[19] Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, and Ursula Martin. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In *Theorem Proving in Higher Order Logics (TPHOLs 2001), volume 2152 of LNCS*, pages 27–42. Springer-Verlag, 2001.

[20] Jacques Fleuriot and Lawrence C. Paulson. Proving Newton's *Propositio Kepleriana* using geometry and nonstandard analysis in Isabelle. In Xiao-Shan Gao, Dong-

ming Wang, and Lu Yang, editors, *Workshop on Automated Deduction in Geometry, Second International Workshop, ADG'98*, LNCS 1669, pages 47–66. Springer, 1999.

[21] Jacques D. Fleuriot and Lawrence C. Paulson. A Combination of Nonstandard Analysis and Geometry Theorem Proving, with Application to Newton's Principia. In *CADE-15: Proceedings of the 15th International Conference on Automated Deduction*, pages 3–16, London, UK, 1998. Springer-Verlag. ISBN 3-540-64675-2.

[22] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X.

[23] Dov M. Gabbay, C. J. Hogger, and J. A. Robinson. *Handbook of logic in artificial intelligence and logic programming (Vol. 4): epistemic and temporal reasoning*. Oxford University Press, Oxford, UK, 1995. ISBN 0-19-853791-3.

[24] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[25] Michael J. C. Gordon, James Reynolds, Warren A. Hunt, and Matt Kaufmann. An Integration of HOL and ACL2. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 153–160, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2707-8.

[26] Mike Gordon. *From LCF to HOL: a short history*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5.

[27] P. Deepak Sreenivas G.Phanindra, K.V.V.N. Ravi Shankar. A Fast Multiple Pattern Matching Algorithm using Context Free Grammar and Tree Model. *IJCSNS International Journal of Computer Science and Network Security*, 7(9):231–234, 2007.

[28] Sheila A. Greibach. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM*, 12(1):42–52, 1965. ISSN 0004-5411.

[29] David Griffioen and Marieke Huisman. A Comparison of PVS and Isabelle/HOL. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 123–142, London, UK, 1998. Springer-Verlag. ISBN 3-540-64987-5.

[30] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *J. Autom. Reason.*, 21(3):279–294, 1998. ISSN 0168-7433.

[31] John Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14

A, FIN-20520 Turku, Finland, 1996. Available on the Web as http://www.cl.cam.ac.uk/˜jrh13/papers/form-math3.html.

[32] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[33] John Harrison. Towards self-verification of hol light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[34] Wim H Hesselink. Dr Wim H Hesselink: University of Groningen: Homepage. http://www.cs.rug.nl/~wim/.

[35] HOL source code. HOL source code for mechanisation of the theory of CFGs and PDA. http://users.rsise.anu.edu.au/~aditi/.

[36] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Ma., USA, 1979. ISBN 0-201-02988-X.

[37] Peter Zilahy Ingerman. "Panini-Backus Form" suggested. *Commun. ACM*, 10(3): 137, 1967.

[38] Julie Rehmeyer. How to (really) trust a mathematical proof. *ScienceNews*, 2008.

[39] Gada Kadoda. A Cognitive Dimensions view of the differences between designers and users of theorem proving assistants, 2000.

[40] Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. In *Proceedings of the 19th European Symposium on Programming (ESOP '10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 345–365, 2010.

[41] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[42] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. URL http://dblp.uni-trier.de/db/conf/popl/popl2006.html#Leroy06.

[43] D. MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001.

[44] William McCune. Solution of the Robbins Problem. *J. Autom. Reason.*, 19(3): 263–276, 1997. ISSN 0168-7433.

[45] Sean McLaughlin. An Interpretation of Isabelle/HOL in HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 192–204. Springer Berlin/Heidelberg, 2006.

[46] Catherine Meadows. Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends, 2003.

[47] Charlotte L. Miller. Towards a Meta-Normal Form Algorithm for Context-Free Grammars. In *WIA '97: Revised Papers from the Second International Workshop on Implementing Automata*, pages 133–143, London, UK, 1998. Springer-Verlag. ISBN 3-540-64694-9.

[48] Yasuhiko Minamide. Verified Decision Procedures on Context-Free Grammars. In *TPHOLs*, pages 173–188, 2007.

[49] Masaki Murata, Kiyotaka Uchimoto, Qing Ma, and Hitoshi Isahara. Magical Number Seven Plus or Minus Two: Syntactic Structure Recognition in Japanese and English Sentences. In *CICLing*, pages 43–52, 2001.

[50] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3.

[51] R. P. Nederpelt. *Selected Papers on Automath*. Elsevier Publishing Company, 1994. ISBN 0444898220.

[52] 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. URL `citeseer.ist.psu.edu/nipkow98verified.html`.

[53] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer Berlin/Heidelberg, 2006.

[54] G. O'Keefe. Towards a Readable Formalisation of Category Theory. *Electronic Notes in Theoretical Computer Science*, 91:212–228, February 2004. ISSN 15710661.

[55] Ioana Paşca. A Formal Verification for Kantorovitch's Theorem. In *Journées Francophones des Langages Applicatifs*, January 2008. URL `http://hal.inria.fr/inria-00202808/en/`.

[56] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. *CoRR*, cs.LO/9301106, 1993.

[57] Lawrence Paulson. Basic concepts. In Lawrence Paulson, editor, *Isabelle*, volume 828 of *Lecture Notes in Computer Science*, pages 179–183. Springer Berlin/Heidelberg, 1994.

[58] Lawrence C. Paulson and Jasmin Blanchette. Three Years of Experience with Sledgehammer, a Practical Link between Automated and Interactive Theorem Provers. In *8th International Workshop on the Implementation of Logics*, 2010.

[59] Benjamin C. Pierce. Lambda, The Ultimate TA: Using a Proof Assistant to Teach Programming Language Foundations, September 2009. Keynote address at *International Conference on Functional Programming (ICFP)*.

[60] Benjamin C. Pierce. Proof Assistant as Teaching Assistant: A View from the Trenches, July 2010. Keynote address at *International Conference on Interactive Theorem Proving (ITP)*.

[61] C. J. Pollard. *Generalized phrase structure grammars, head grammars, and natural language*. PhD thesis, Stanford University, 1984.

[62] Project vdash. vdash website. `http://www.vdash.org/intro/`.

[63] G. K. Pullum and G. Gazdar. Natural languages and context-free languages. *Linguistics and Philosophy*, 4(4), 1982.

[64] QED group. The QED Manifesto. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 238–251, London, UK, 1994. Springer-Verlag. ISBN 3-540-58156-1.

[65] Xavier Rival and Jean Goubault-Larrecq. Experiments with Finite Tree Automata in Coq. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, pages 362–377, London, UK, 2001. Springer-Verlag. ISBN 3-540-42525-X. URL `http://portal.acm.org/citation.cfm?id=646528.695061`.

[66] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997. ISBN 3-540-60420-0.

[67] Piotr Rudnicki. An Overview of the MIZAR Project. In *University of Technology, Bastad*, pages 311–332, 1992.

[68] N. Shankar. A mechanical proof of the Church-Rosser theorem. *J. ACM*, 35(3): 475–522, 1988. ISSN 0004-5411.

[69] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technische Universität München, 1999.

[70] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. See also the HOL website at `http://hol.sourceforge.net`.

[71] Martin Strecker. Compiler Verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.

[72] T. Kasami, H. Seki and M. Fujii. Generalized context-free grammars, multiple context-free grammars and head grammars. Technical report, Osaka University, 1987.

[73] Laurent Théry. Formalising Huffman's algorithm. Technical report 034, Università di L'Aquila, L'Aquila, Italie, 2004.

[74] L. S. van Benthem. Review of Checking Landau's Grundlagen in the Automath system by L. S. Van Benthem Jutting. Mathematical Centre 1979.: "First order dynamic logic" by David Harel. Springer-Verlag 1979. And "A programming logic" by Robert L. Constable and Michael J. O'Donnell. Winthrop Publishers 1978. *SIGACT News*, 12(3):14–16, 1980. ISSN 0163-5700. Reviewer-Cherniavsky, John.

[75] Markus Wenzel and Freek Wiedijk. A Comparison of Mizar and Isar. *J. Autom. Reason.*, 29(3-4):389–411, 2002. ISSN 0168-7433.

[76] Freek Wiedijk. The De Bruijn Factor, 2000.

[77] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-30704-4.

[78] Freek Wiedijk. The QED manifesto revisited. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 121–133. University of Białystok, 2007. URL `http://mizar.org/trybulec65/`.

[79] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice Hall Professional Technical Reference, 1984. ISBN 0130544469.

[80] Larry Wos, Dolph Ulrich, and Branden Fitelson. Vanquishing the XCB Question: The Methodological Discovery of the Last Shortest Single Axiom for the Equivalential Calculus. *J. Autom. Reason.*, 29(2):107–124, 2002. ISSN 0168-7433.

[81] Victor H. Yngve. A Model and an Hypothesis for Language Structure. In *Proceedings of the American Philosophical Society*, volume 104, pages 444–466. American Philosophical Society, 1960.

[82] Vincent Zammit. A Comparative Study of Coq and HOL. In *Gunter and Felty*, pages 323–337. Springer, 1997.

Index