The Poor Man's Proof Assistant

Using Prolog to Develop Formal Language Theoretic Proofs

Joey Eremondi
University of Saskatchewan
joey.eremondi@usask.ca

Abstract

While proving a theorem from a set of axioms is undecidable in first order logic, recent development has produced several tools which serve as automated theorem provers. However, often these systems are too complex for a given problem. Their usefulness is outweighed by the difficulty of learning a new tool or translating results into computer-readable form.

I describe tools developed in Prolog to partially characterize the shuffle-inclusion problem. These tools allowed for rapid development of proofs with little intellectual overhead. While focused around a specific problem, the techniques described are general, and well suited to many problems on discrete structures.

Categories and Subject Descriptors D.1.6 [*PROGRAMMING TECHNIQUES*]: Logic programming

Keywords Prolog; shuffle; proof assistant

1. Problem and Motivation

This paper shows how logic languages such as Prolog provide a fast and easy way to develop assistive tools in theorem proving. As a case study, I use the characterization of the shuffle inclusion problem.

The shuffle operator, which represents the set of all ways of "merging" two strings, has received much attention in recent research. Its main application is in modelling parallelism [10]. Moreover, it represents a fundamental operation on strings, so researching it contributes to a greater understanding of the theory underlying computation.

I examined shuffle inclusion, the question of whether one shuffle set was a subset of another. The search tools I developed for this problem serve as an example of how logic languages can be used to facilitate the development of proofs in formal languages and discrete mathematics.

2. Background and Related Work

2.1 The Prolog Language

A detailed introduction to Prolog can be found in [6]. There are three features which are particularly relevant here. *Unification* allows logical predicates to generate answers, rather than simply test conditions, by leaving some variables *unbound*. Roughly, Prolog searches for a value causing a goal to succeed. *Nondeterminism* allows for multiple definitions of a predicate. These are each tried in sequence when proving a goal, allowing queries to return multiple answers. The *Definite Clause Grammar* (DCG) feature provides tools for string parsing and generation. While based on context-free grammars, they can be mixed with arbitrary Prolog code for context-sensitve testing.

2.2 Shuffle Inclusion

I use Σ^* to denote the set of all (possibly empty) words over the alphabet Σ, and $|w|$ to denote the length of a word w. The shuffle operator $\sqcup\!\sqcup$ denotes all ways of interleaving the letters from two words while preserving their order relative to the original words. It maps two strings to a finite set of strings, and is formally defined in [3] as follows:

$$u \sqcup\!\sqcup v = \{u_1 v_1 \cdots u_n v_n \mid u = u_1 \cdots u_n, v = v_1 \cdots v_n, \\ u_i \in \Sigma^*, v_i \in \Sigma^*, 1 \leq i \leq n\}$$

In [3] and [2] it was shown that $u \sqcup\!\sqcup v = x \sqcup\!\sqcup y \iff \{u, v\} = \{x, y\}$ when u and v contain at least two distinct letters. When equality is so easily described, it is natural to ask a similar question about the inclusion problem, and to examine when $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$.

2.3 Proof Assistants

Despite the undecidability of first-order logic [5], many successful proof-assistants exist. HOL [9] is based on higher-order logic. Coq uses dependent types, and was used to verify proof of the four color theorem [8]. Several of these languages have counter-example search packages, such as Nitpick [4] and Kodkod [11]. The Alloy Analyzer provides counter-example search with SAT-solvers, with a focus on specifying Object-Oriented systems [1]. However, these languages can be challenging to learn, and translating statements into computer-readable code is difficult and tedious. Some Prolog tools, such as PFLAT [12] have been developed for formal language work, but these are geared more towards education rather than real theorem proving.

3. Approach and Uniqueness

My approach to the inclusion problem was to find a complete characterization of cases when the subset relation holds. For example, $xwy \sqcup\!\sqcup w \subseteq xw \sqcup\!\sqcup wy$ for all $w, x, y \in \Sigma^*$. My research followed a cyclic approach. Initially, a list of all u, v, x, y fulfilling $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$ and $|uv| \leq k$ was generated. I examined this list to find patterns and develop hypotheses about the subset relation. Once a conjecture was formulated, I would use a search framework to search for counter-examples of length k. If no counter-example was found, I attempted a pen-and-paper formal proof, then wrote a

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2508075.2508088

predicate to test which strings matched its pattern. The process was repeated, but strings matching previously proved theorems were filtered out. Thus I could gauge the completeness of my characterization, continually narrowing the list of strings I examined.

3.1 Enumeration

Since strings are discrete objects, it is possible to enumerate all cases where $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$ with bounds on $|uv|$. In order to help find patterns, my program printed all such u, v, x, y of a given length. With Prolog's nondeterminism and unification, the predicate specifying such u, v, x, y also served to enumerate them. This predicate declared a list with unbound members, then declared u, v as a partition of that list. Another predicate, `hasSameParikh`, selected a number of a's and b's for uv to have, then unified x, y with words having the same number of each letter. Finally, x, y were unified with words fulfilling the subset relation. Once a theorem was proved and added to the characterization, any u, v, x, y which fulfilled the conditions of the theorem were filtered out. This enumeration allowed me to experimentally prove that my characterization was complete up to length 8.

3.2 Counter-example Search and Hypothesis Testing

The hypothesis-testing tool generated strings u, v, x, y matching a hypothesis' conditions with $|uv| = k$, then tested if $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$. Two k-element lists were declared with unbound contents. Then u, v and x, y were declared as a partitioning of these lists.

The search called a rule `counterCond` which unified a query u, v, x, y with strings matching my hypothesis. The rule was rewritten for each conjecture being tested, usually using DCG clauses.

By passing an unbound list to `counterCond`, the predicate generated strings that matched the hypothesis condition. This was more efficient than generating every pair of strings of a given length and testing if they matched the hypothesis condition. The logic-programming approach meant that no additional code was written to perform this optimization.

Restricting search to a given length was necessary, since recursion on unbound-length strings caused infinite loops. Continually increasing k led to an iterative-deepening search, ensuring that the search would halt and find a counter-example of length k if it existed.

3.3 Definite clause grammars

The task of matching a set of strings against a given "pattern" arose frequently. DCG's provided a useful tool for expressing such patterns. Regularly used clauses were written, such as `aToMbToN` to match $a^m b^n, m, n \in \mathbb{N}_0$. These could be combined and composed. For example, `concatStr` concatenated strings matching different DCG clauses. Likewise, calling `concatStr` on two different words could then be used to test for a common infix between them. By using Prolog's DCG tool, patterns and conditions could be coded in an easily readable form without requiring a parser or generator to be written.

3.4 Inductive queries

Prolog is designed to have programs run interactively from a console, rather than compiled to an executable. This allowed my framework to answer queries without having to write any sort of user interface. A common usage of this feature was querying which discovered patterns, if any, proved $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$ for a given u, v, x, y. Many of the proofs that patterns held were based on induction, constructing new relations from ones I had already proved. While these proofs were completed manually, the process was made much easier with the ability to query previously-proved theorems.

4. Results and Contribution

Using the techniques described above, I found a list of over 40 patterns which completely characterize when $u \sqcup\!\sqcup v \subseteq x \sqcup\!\sqcup y$, with $uv \in \{a, b\}^*, |uv| \leq 8$. For example, $w \sqcup\!\sqcup xwy \subseteq xw \sqcup\!\sqcup wy$, and $a^p w b^k \sqcup\!\sqcup a^q w b^l \subseteq a^{p+q} w \sqcup\!\sqcup w b^{k+l}$.

While human-developed proofs verified the correctness of each subset relation, their completeness for length 8 was verified using my search tools. The development of pen-and-paper proofs was made easier with the use of my assistive tools. The full list of patterns, as well as proof of each pattern's correctness, can be found in [7]. The techniques used here can be generalized to other problems. In particular, the counter-example search could be used to test any hypothesis involving a finite number of discrete structures.

5. Conclusion

I presented Prolog tools for assisting with proofs and formal language-theoretic research. These demonstrate how software can help automate the proof developing process, and how Prolog's unique feature-set allows for rapid development and interactive use of such software. While more complete tools exist, my framework provided a shallower learning curve better suited to the day-to-day challenges of formal language research. My success in partially characterizing the shuffle inclusion problem provides a strong example of both the ease and effectiveness of these tools.

Acknowledgments

The author was supervised by Dr. Ian McQuillan and supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from uml to alloy. *Software & Systems Modeling*, 9(1):69–86, 2010. ISSN 1619-1366. .

[2] J. Berstel and L. Boasson. Shuffle factorization is unique. *Theoretical Computer Science*, 273:47–67, 2002.

[3] F. Biegler, M. Daley, M. Holzer, and I. McQuillan. On the uniqueness of shuffle on words and finite languages. *Theoretical Computer Science*, 410:3711–3724, 2009.

[4] J. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14051-8. .

[5] A. Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1): 40–41, 1936.

[6] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, 5th edition, 9 2003. ISBN 9783540006787.

[7] J. Eremondi and I. McQuillan. A Partial Characterization of the Shuffle Inclusion Problem on Words. Manuscript in preparation, 2013.

[8] G. Gonthier. Formal proof the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[9] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993. ISBN 0-521-44189-7.

[10] A. Mateescu, G. Rozenberg, and A. Salomaa. Shuffle on trajectories: Syntactic constraints. *Theoretical Computer Science*, 197(1–2):1–56, 1998.

[11] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and ...*, pages 632–647, 2007.

[12] M. Wermelinger and A. M. Dias. A prolog toolkit for formal languages and automata. *SIGCSE Bull.*, 37(3):330–334, June 2005. ISSN 0097-8418. .