Tabled higher-order logic programming

Brigitte Pientka

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:
Frank Pfenning, Carnegie Mellon University, Chair
Robert Harper, Carnegie Mellon University
Dana Scott, Carnegie Mellon University
David Warren, University of New York at Stony Brook

Summary

A logical framework is a general meta-language for specifying and implementing deductive systems, given by axioms and inference rules. Examples of deductive systems are plentiful in computer science. In computer security, we find authentication and security logics to describe access and security criteria. In programming languages, we use deductive systems to specify the operational semantics, type-systems or other aspects of the run-time behavior of programs. Recently, one major application of logical frameworks has been in the area of "certified code". To provide guarantees about the behavior of mobile code, safety properties are expressed as deductive systems. The code producer then verifies the program according to some predetermined safety policy, and supplies a binary executable together with its safety proof (certificate). Before executing the program, the host machine then quickly checks the code's safety proof against the binary. The safety policy and the safety proofs can be expressed in the logical framework thereby providing a general safety infrastructure.

There are two main variants of logical frameworks which are specifically designed to support the implementation of deductive systems. λProlog and Isabelle are based on hereditary Harrop formulas, while the Twelf system [14] is an implementation of the logical framework LF, a dependently typed lambda calculus. In this thesis, we will mainly focus on the latter. By assigning a logic programming interpretation to types [12], we obtain a higher-order logic programming language. Higher-order logic programming in Twelf extends traditional first-order logic programming in three ways: First, we have a rich type system based on dependent types, which allows the user to define her own higher-order data-types and supports higher-order abstract syntax [13]. Variables in the object language can be directly represented as variables in the meta-language thereby directly inheriting capture-avoiding substitution and bound variable renaming. Second, we not only have a static set of program clauses, but clauses may be introduced dynamically and used within a certain scope during proof search. Third, we have an explicit notion of proof, i.e. the logic programming interpreter does not only return an answer substitution for the free variables in the query, but also the actual proof of the query as a term in the dependently typed lambda-calculus. This stands in sharp contrast to higher-order features supported in many traditional logic program-

ming languages (see for example [3]) where we can encapsulate predicate expressions within terms to later retrieve and invoke such stored predicates. Twelf's higher-order logic programming interpreter is complemented by a meta-theorem prover, which combines generic proof search based on higher-order logic programming with inductive reasoning [14, 20].

The Twelf system has been successfully used to implement, execute and reason about a wide variety of deductive systems. However, experience with real-world applications in different projects on certified code [2, 4, 1] have increasingly demonstrated the limitations of Twelf's higher-order logic programming proof search. To illustrate, let us briefly consider the foundational proof-carrying code project at Princeton. As part of this project, the researchers at Princeton have implemented between 70,000 and 100,000 lines of Twelf code, which includes data-type definitions and proofs. The higher-order logic program, which is used to execute safety policies, consists of over 5,000 lines of code, and over 600 – 700 clauses. Such large specifications have put to test implementations of logical frameworks and exposed several problems. First, performance of the higher-order logic programming interpreter may be severely hampered by redundant computation, leading to long response times and slow development of formal specifications. Second, many straightforward specifications of formal systems, for example recognizers and parsers for grammars, rewriting systems, type systems with subtyping or polymorphism, are not executable, thus requiring more complex and sometimes less efficient implementations. Thirdly, redundancy severely hampers the reasoning with and about deductive systems in general, limiting the use of the meta-theorem prover.

In applications to certified code, efficient proof search techniques not only play an important role to execute safety polices and generate a certificate that a given program fulfills a specified safety policy, but it also can be used to check the correctness of a certificate [10]. Necula and Rahul [10] propose as a certificate a bit-string of the non-deterministic choices in the proof. Hence, a proof can be checked by guiding the higher-order logic programming interpreter with the bit-string and reconstructing the actual proof. As pointed out by Necula and Lee, typical safety proof in the context of certified code commonly have repeated sub-proofs that should be hoisted out and proved only once. The replication of common sub-proofs leads to redundancy in the

bit-strings representing the safety proof and it may take longer to reconstruct the safety proof using a guided higher-order logic programming interpreter.

In this thesis, we develop different techniques which improve the overall performance and the expressive power of the higher-order logic programming interpreter. We also apply these ideas in the meta-theorem prover to overcome existing limitations when reasoning about deductive systems. These optimizations taken together constitute a significant step toward exploring the full potential of logical frameworks in real-world applications. Some of the work in this thesis has been previously published in different forms [15, 16, 17, 18, 9]

Contributions

The contributions in this thesis are in three main areas: First, we introduce tabled higher-order logic programming, a novel execution model where some redundant information is eliminated using selective memoization. This forms the basis of the tabled higher-order logic programming interpreter. Second, we develop efficient data-structures and algorithms for higher-order proof search. These optimizations are crucial to make tabled higher-order logic programming successful in practice. Although we develop these techniques in the context of tabled logic programming, they are also independently useful and important to other areas such as higher-order rewriting, higher-order theorem proving and higher-order proof checking. Third, we use memoization-based proof search in the meta-theorem prover, to reason efficiently with and about deductive systems. This demonstrates the importance of memoization in general. Next, we will discuss briefly each of these contributions.

1. Tabled higher-order logic programming

Tabled first-order logic programming has been successfully applied to solve complex problems such as implementing recognizers and parsers for grammars [21], representing transition systems CCS and writing model checkers [5]. The idea behind it is to eliminate redundant computation by memoizing sub-computation and re-using its results later. The resulting search procedure is complete and terminates for programs with

the bounded-term size property. The XSB system [19], a tabled logic programming system, demonstrates impressively that tabled together with non-tabled programs can be executed efficiently in the first-order setting.

The success of memoization in first-order logic programming strongly suggests that memoization may also be valuable in higher-order logic programming. In fact, Necula and Lee point out in [11] that typical safety proofs in the context of certified code commonly have repeated sub-proofs that should be hoisted out and proved only once. Memoization has potentially three advantages. First, proof search is faster thereby substantially reducing the response time to the programmer. Second, the proofs themselves are more compact and smaller. This is especially important in applications to secure mobile code where a proof is attached to a program, as smaller proofs take up less time to check and transmit to another host. Third, substantially more specifications, for example recognizers and parser for grammars, evaluators based on rewriting or type systems with subtyping, are executable under the new paradigm thereby extending the power of the existing system.

Using memoization in higher-order logic programming poses several challenges, since we have to handle type dependencies and may have dynamic assumptions which are introduced during proof search. This is unlike tabling in XSB, where we have no types and it suffices to memoize atomic goals. Moreover, most descriptions of tabling in the first-order setting are closely oriented on the WAM (Warren Abstract Machine) making it hard to transfer tabling techniques and design extensions to other logic programming interpreters.

In this thesis, we introduce a novel execution model for logical frameworks based on selective memoization.

Proof-theoretic characterization of uniform proofs and memoization We give a proof-theoretic characterization of tabling based on uniform proofs, and show soundness of the resulting interpreter. This provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms.

Implementation of a tabled higher-order logic programming interpreter We give a high-level description of a semi-functional implementation for adding tabling

to a higher-order logic programming interpreter. We give an operational interpretation of the uniform proof system and discuss some of the implementation issues such as suspending and resuming computation, retrieving answers, and trailing. Unlike other description, it does not require an understanding or modifications, and extensions to the WAM (Warren abstract machine). It is intended as a high-level explanation and guide for adding tabling to an existing logic programming interpreter. This is essential for rapidly prototyping tabled logic programming interpreters, even for linear logic programming and other higher-order logic programming systems.

Case studies We discuss three case studies to illustrate the use of memoization in the higher-order setting. We consider a parser and recognizer for first-order formulas into higher-order abstract syntax. To model left and right associativity of the different connectives, we mix left and right recursion in the specification of the parser. Although this closely models the grammar, it leads to an implementation which is not executable with traditional logic programming interpreters which are based on depth-first search.

The second case study is an implementation of a bi-directional type-checker by Davies and Pfenning [6]. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type.

2. Efficient data-structures and algorithms

Efficient data-structures and implementation techniques play a crucial role in utilizing the full potential of a reasoning environment in large scale applications. Although this need has been widely recognized for first-order languages, efficient algorithms for higher-order languages are still a central open problem.

Proof-theoretic foundation for existential variables based on modal logic We give a dependent modal lambda calculus, which extends the theory of the logical framework LF [7, 8] conservatively with modal variables. Modal variables

can be interpreted as existential variables, thereby clearly distinguishing them from ordinary bound variables. This is critical to achieve a simplified account of higher-order unification and allows us to justify different optimizations such as as lowering, raising, and linearization [18, 9]. It also serves as a foundation for designing higher-order term indexing strategies.

Optimizing unification Unification lies at the heart of logic programming, theorem proving, and rewriting systems. Thus, its performance affects in a crucial way the global efficiency of each of these applications. Higher-order unification is in general undecidable, but decidable fragments, such as higher-order patterns unification, exist. Unfortunately, the complexity of this algorithm is still at best linear, which is impractical for any useful programming language or practical framework. In this thesis, we present an assignment algorithm for linear higher-order patterns which factors out unnecessary occurs checks. Experiments show that we get a speed-up by to a factor $2 - 5$ making the execution of some examples feasible. This is a significant step toward efficient implementation of higher-order reasoning systems in general [18].

Higher-order term indexing Proof search strategies, such as memoization, can only be practical if we can access the memo-table efficiently. Otherwise, the rate of drawing new conclusions may degrade sharply both with time and with an increase of the size of the memo-table. Term indexing aims at overcoming program degradation by sharing common structure and factoring common operations. Higher-order term indexing has been a central open problem, limiting the application and the potential impact of higher-order reasoning systems. In this thesis, we develop and implemented higher-order term indexing techniques. They improve performance by up to a factor of 10, illustrating the importance of indexing [17].

3. Meta-theorem proving based on memoization

The traditional approach for supporting theorem proving in these frameworks is to guide proof search using tactics and tacticals. Tactics transform a proof structure

with some unproven leaves into another. Tacticals combine tactics to perform more complex steps in the proof. Tactics and tacticals are written in ML or some other strategy language. To reason efficiently about some specification, the user implements specific tactics to guide the search. This means that tactics have to be rewritten for different specifications. Moreover, the user has to understand how to guide the prover to find the proof, which often requires expert knowledge about the systems. Proving the correctness of the tactic is itself a complex theorem proving problem.

The approach taken in the Twelf system is to endow the framework with the operational semantics of logic programming and design general proof search strategies for it. Twelf's meta-theorem prover combines general proof search based on higher-order logic programming with inductive reasoning. Using the proof-theoretic characterization of tabling, we develop a general memoization-based proof search strategy which is incorporated in Twelf's meta-theorem prover. As experiments demonstrate, eliminating redundancy in meta-theorem proving is critical to prove properties about larger and more complex specifications. We discuss several examples including type preservation proofs for type-system with subtyping, several inversion lemmas about refinement types, and reasoning in classical natural deduction calculus. These examples include several lemmas and theorems which were not previously provable. Moreover, we show that in many cases no bound is needed on memoization-based search. As a consequence, if a sub-case is not provable, the user knows, that in fact no proof exists. This in turn helps the user to revise the formulation of the theorem or the specification. Overall the benefits of memoization are an important step towards a more robust and more efficient meta-theorem prover.

Bibliography

[1] W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, Jan. 2000.

[2] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002.

[3] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[4] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.

[5] B. Cui, Y. Dong, X. Du, K. N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In Hugh Glaser Catuscia Palamidessi and Karl Meinke, editors, *Principles of Declarative Programming (Proceedings of PLILP/ALP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.

[6] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the International Conference on Functional Programming (ICFP 2000), Montreal, Canada*, pages 198–208. ACM Press, 2000.

[7] Robert Harper and Frank Pfenning. On equivalence and canonical forms in lf type theory. Technical Report CMU-CS-00-148, School of Computer Science, Carnegie Mellon University, 2000.

[8] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 2003. To appear. Preliminary version available as Technical Report CMU-CS-00-148.

[9] Aleksander Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta-variables. In *2nd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with variable binding (Merlin), Uppsala, Sweden*. to appear, August 2003.

[10] G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.

[11] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91. Springer-Verlag LNCS 1419, August 1998.

[12] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[13] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[14] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[15] Brigitte Pientka. Memoization-based proof search in LF: an experimental evaluation of a prototype. In *Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02), Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science (ENTCS), 2002.

[16] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.

[17] Brigitte Pientka. Higher-order substitution tree indexing. In C. Palamidessi, editor, *19th International Conference on Logic Programming, Mumbai, India*, Lecture Notes in Computer Science (LNCS), to appear. Springer-Verlag, 2003.

[18] Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Computer Science (LNCS), to appear. Springer-Verlag, 2003.

[19] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[20] Carsten Schürmann. *Automating the meta theory of deductive systems.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-00-146, 2000.

[21] David S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/~warren/xsbbook/book.html, 1999.