Overcoming performance barries:
efficient proof search in logical frameworks

Brigitte Pientka*

Department of Computer Science
Carnegie Mellon University
bp@cs.cmu.edu

A logical framework is a general meta-language for specifying and implementing deductive systems, given by axioms and inference rules. 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. Moreover, it is possible to experiment with safety policies and verify that program is in fact safe by executing the safety policy via a logic programming interpretation.

The Twelf system [22] is an implementation of a logical framework based on the dependently typed lambda calculus [8]. By assigning an operational interpretation to types, we obtain a higher-order logic programming language [19]. Higher-order logic programming 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[21]. Second, we not only have a static set of program clauses, but assumptions may be introduced dynamically 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 the actual proof as a term in the dependently typed lambda-calculus.

The Twelf system has been used in several projects on certified code [4, 5, 3]. The code size in the foundational proof-carrying code project [1] at Princeton ranges 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 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.

Here we describe three techniques to overcome some of the existing performance barriers and extend its expressive power. This summarizes some recent work which was carried out as part of the Twelf project at Carnegie Mellon University. First, we describe the central idea behind optimizing unification in logical frameworks by eliminating unnecessary occurs-checks. Second we present a novel execution model for higher-order logic programming based on selective memoization and third we will discuss higher-order term indexing techniques to sustain performance in large-scale examples. Experiments carried out in Twelf demonstrate, these optimizations taken together constitute a significant step toward exploring the full potential of logical frameworks in real-world applications. Although the main focus of this work has been the logical framework Twelf, we believe the presented optimizations are applicable to any higher-order reasoning system such as λProlog [13] or Isabelle[18].

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. This need for efficient

* This work has been partially supported by NSF Grant CCR-9988281 "*Logical and Meta-Logical Frameworks*" and by a *Siebel Scholarship*.

unification algorithms has led to many investigations in the first-order setting. However, the efficient implementation of higher-order unification, especially for dependently typed λ-calculus, is still a central open problem limiting the potential impact of higher-order reasoning systems such as Twelf [22], Isabelle [18], or λProlog [13].

The most comprehensive study on efficient and robust implementation techniques for higher-order unification so far has been carried out by Nadathur and colleagues for the simply-typed λ-calculus in the programming language λProlog [14] Higher-order unification is implemented via Huet's algorithm [9] and special mechanisms are incorporated into the WAM instruction set to support branching and postponing unification problems. To only perform an occurs-check when necessary, the compiler distinguishes between the first occurrence and subsequent occurrences of a variable and compiles them into different WAM instructions. While for the first occurrence of a variable the occurs-check may be omitted, full unification is used for all subsequent variables. This approach seems to work well in the simply-typed setting, however it is not clear how to generalize it to dependent types. In addition, it is well known, that Huet's algorithm is highly non-deterministic and requires backtracking. An important step toward efficient implementations, has been the development of higher-order pattern unification [11, 20]. For this fragment, higher-order unification is decidable and deterministic. As was shown in [10], most programs written in practice fall into this fragment. Unfortunately, the complexity of this algorithm is still at best linear [27] in the sum of the sizes of the terms being unified, which is impractical for any useful programming language or practical framework.

In [26], the author and Pfenning present an abstract view of existential variables in the dependently typed lambda-calculus based on modal type theory. This allows us to distinguish between existential variables, which are represented as modal variables, and bound variables, which are described by ordinary variables. This leads to a simple clean framework which allows us to explain a number of features of the current implementation of higher-order unification in Twelf [22] and provides insight into several optimizations such as lowering and raising. In particular, it explains one optimization called linearization, which eliminates many unnecessary occurs-checks. Terms are compiled into linear higher-order patterns and some additional variable definitions. Linear higher-order patterns restrict higher-order patterns in two ways: First, all existential variables occur only once. Second, we impose some further syntactic restrictions on existential variables, i.e. they must be applied to *all* distinct bound variables. This is in contrast to higher-order patterns, which only require that existential variables are applied to *some* bound variables. Linear higher-order patterns can be solved with an assignment algorithm which resembles first-order unification (without the occurs check) closely and is constant time. Experimental results show that a large class of programs falls into the linear higher-order pattern fragment and can be handled with this algorithm. This leads to significant performance improvement (up to a factor of 5) in many example applications including those in the area of proof-carrying code.

Selective memoization in higher-order logic programming

Memoization, also called tabling, has been used very successfully in first-order logic programming to solve complex problems such as implementing recognizers and parsers for grammars [29], representing transition systems (CCS) and writing model checkers [6]. 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 [28] demonstrates impressively that tabled logic programs can be executed efficiently and in fact can be mixed with Prolog programs to achieve the best of both worlds.

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 [16] that typical safety proof 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.

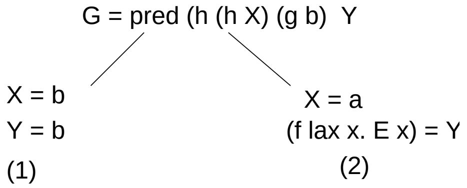
In [24], we have presented a characterization of tabled higher-order logic programing based on uniform proofs [12]. This abstract view provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms. It also facilitates applying memoization to other logic programming environments such as λProlog [13] or using memoization in Isabelle [18].

We have implemented the search strategy based on memoization for the Twelf system and have conducted several experiments with parser and recognizers for grammars, refinement type systems, and reduction semantics for functional programs, rewriting systems for the lambda-calculus etc [23]. Experimental results demonstrate that memoization not only allows us to execute more specifications, but also execute them more efficiently. Since computation based on memoization terminates for programs with bounded-term size property and it is a complete search strategy, we also get better and more useful failure behavior. We see this work is an important step toward a practical programming environment for developing and prototyping safety policies and deductive systems in general.

Higher-order term indexing

Efficient data-structures and implementation techniques play a crucial role in utilizing the full potential of a reasoning environment in real-world applications. In particular, proof search strategies based on 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. This need has been widely recognized and lead to numerous term indexing techniques for first-order languages. However, term indexing techniques for higher-order languages are essentially non-existent thereby limiting the application and the potential impact of higher-order reasoning systems.

We have designed and implemented higher-order term indexing techniques based on substitution trees [25] thereby extending first-order substitution tree indexing to the higher-order setting. Substitution tree indexing is a highly successful technique in first-order theorem proving [7], which allows the sharing of common sub-expressions via substitutions. Given two terms $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{b}$ and $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{a}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda\ \mathsf{x}.\ \mathsf{U}\ \mathsf{x})$, we compute their most specific generalization $\mathsf{G} = \mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{X}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{Y}$. We obtain the first term by applying the substitution $\mathsf{X} = \mathsf{a}$ and $\mathsf{Y} = (\mathsf{f}\ \lambda\ \mathsf{x}.\ \mathsf{U}\ \mathsf{x})$ to $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{X}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{Y}$. Similarly, we obtain the second term by applying the substitution $\mathsf{X} = \mathsf{b}$ and $\mathsf{Y} = \mathsf{b}$ to it. Next, we show the corresponding substitution tree.



We can obtain the first term by composing all the substitutions in the left-most path.

This poses several challenges: First of all, building a substitution tree relies on computing the most specific generalization of two terms. However in the higher-order setting, the most specific generalization of two terms may not exist in general. Second, retrieving all terms, which unify or match, needs to be efficient – but higher-order unification is undecidable in general. Although, most specific generalizations exist for higher-order patterns and decidable higher-order pattern unification [11, 20], these algorithms may not be efficient in practice [26]. Therefore, it is not obvious that they are suitable for higher-order term indexing techniques. Instead, we use linear higher-order patterns as a basis for higher-order term indexing [26]. This allows us to reduce the problem of computing most specific generalizations for higher-order terms to an algorithm which resembles closely its first-order counterpart [25]. This technique has been implemented to speed-up the execution of the tabled higher-order logic programming engine in Twelf. Experimental results demonstrate that higher-order term indexing leads to substantial performance improvements (by up to a factor of 10), illustrating the importance of indexing in general [25].

Conclusion

We presented several techniques which considerably improve the performance of higher-order reasoning systems. This a first step toward exploring the full potential of logical frameworks in practice and apply it to new areas such as security and authentication [2]. However, the presented techniques are just a first step toward narrowing the performance gap between higher-order and first-order systems. There are many more optimizations which have been already proven successful in the first-order setting and we may be able to apply to higher-order languages. For example, term indexing may not only be used for managing the memo-tables, but also be apply to indexing program clauses. There are also new considerations due to the dependently typed setting. For example, typing information needs be carried around during run-time and leads to redundancies in the representation of terms. A first step toward eliminating this kind of redundancy has been proposed by Necula and Lee [17] for a fragment of the logical framework. In addition, the application of logical frameworks to certified code raises new question which traditionally have not played a central role in logic programming. The central idea in certified code is not only to verify that a program is safe, but also to efficiently transmit and then check the proof. Necula and Rahul [15] explored the novel use of higher-order logic programming for checking the correctness of a certificate. In their case, the certificate is a bit-string which encodes the non-deterministic choices of a proof and can be used to guide a logic programming interpreter. Hence, a proof can be checked by guiding the higher-order logic programming interpreter with the bit-string and reconstructing the actual proof. Here, memoization can be used to factor out common sub-proofs and eliminate the checking of redundant sub-proofs.

References

1. Andrew Appel. Foundational proof-carrying code project. personal communication.
2. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
3. 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.
4. 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.
5. 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.
6. 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.
7. Peter Graf. Substitution tree indexing. In *Prooceedings of the sixth International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*, Lecture Notes in Computer Science (LNCS) 914, pages 117–131. Springer-Verlag, 1995.
8. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
9. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
10. Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

11. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
12. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
13. Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
14. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of lambda prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.
15. 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.
16. George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *Special Issue on Mobile Agent Security*, Lecture Notes in Computer Science (LNCS) 1429, 1997.
17. George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
18. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
19. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
20. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of TAPSOFT'93*, pages 391–405. Springer-Verlag Lecture Notes in Computer Science (LNCS) 668, 1993.
28. 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.
29. David S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/ warren/xsbbook/book.html, 1999.