Overcoming performance barries: efficient verification techniques for logical frameworks

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

Abstract. In recent years, logical frameworks which support formalizing language specifications together with their meta-theory have been pervasively used in small and large-scale applications, from certifying code [2] to advocating a general infrastructure for formalizing the meta-theory and semantics of programming languages [5]. In particular, the logical framework LF [9], based on the dependently typed lambda-calculus, and light-weight variants of it like LF_i [17] have played a major role in these applications. While the acceptance of logical framework technology has grown and they have matured, one of the most criticized points is concerned with the run-time performance. In this tutorial we give a brief introduction to logical frameworks, describe its state-of-the art and present recent advances in addressing some of the existing performance issues.

1 Introduction

Logical frameworks [9] provide an experimental platform to specify, and implement formal systems together with the proofs about them. One of its applications lies in proof-carrying code, where it is successfully used to specify and verify formal guarantees about the run-time behavior of programs. More generally, logical frameworks provide an elegant language for encoding and mechanizing the meta-theory and semantics of programming languages. This idea has recently found strong advocates among programming languages researchers who proposed the POPLmark challenge, a set of benchmarks to evaluate and compare tool support to experiment with programming language design, and mechanize the reasoning about programming languages [5].

In this tutorial, we consider the Twelf system [22], an implementation of the logical framework LF [9]. Encodings in LF typically rely on ideas of higher-order abstract syntax where object variables and binders are implemented by variables and binders in the meta-language (i.e. logical framework). One of the key benefits behind higher-order abstract syntax representations is that one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. Moreover, specifications encoded in Twelf can be executed via a higher-order logic programming language [19] thereby allowing the user to experiment with the implemented formal specifications. Higher-order logic programming as found 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[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.

These features make Twelf an ideal framework for specifying properties about programming languages and mechanizing the meta-theory about them. For this reason several projects on proof-carrying code [6, 7, 4] have selected it as their system of choice. 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 a wide range of new questions and problems. For example, the size of safety proofs in logical frameworks may be unreasonably large and validating them may take a long time. In addition, performance of logical frameworks is inadequate for rapid prototyping and large-scale experiments for two main reasons: redundant computation hampers the execution and many optimizations known and well-understood in first-order reasoning are still poorly understood in the higher-order setting.

In this tutorial, we give a brief introduction to logical frameworks and describe different algorithms to overcome some of the existing performance issues and extend its expressive power. First, we describe the central ideas behind optimizing unification in logical frameworks. In particular, we will consider eliminating unnecessary occurs-checks [26] and eliminating redundancy of some dependent type arguments [25]. Second we discuss higher-order term indexing techniques [23] which are for example used in tabled higher-order logic programming [24] to sustain performance in large-scale examples. All these algorithms are described using contextual modal type theory [15] which provides a simple clean foundation to justify and explain concisely complex higher-order issues. We will also discuss experiments with our algorithms within the logical framework Twelf which demonstrate that these optimizations taken together constitute a significant step toward exploring the full potential of logical frameworks in practical 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].

2 Optimizing higher-order pattern unification

Unification lies at the heart of logic programming, theorem proving, and type-reconstruction. Thus, its performance affects in a crucial way the global efficiency of each of these applications. This need for efficient 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 poorly understood 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 [10] 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 [12, 20]. For this fragment, higher-order unification is decidable and deterministic. As was shown in [11], 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 contextual modal type theory. This allows us to distinguish between existential variables, which are represented as contextual 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.

Most recently, we have explored a different optimization to higher-order unification where we skip some redundant implicit type arguments during unification [25]. Unlike our prior optimization which is restricted to proof search, skipping some redundant type arguments during unification is a general optimization and hence impacts not only the proof search performance, but also any other algorithm relying on unification such as type-reconstruction, coverage checking, termination checking etc.

Our experimental results show that although the size of redundant arguments is large and there is a substantial number of them, their impact on run-time performance is surprisingly limited (roughly 20% improvement). Our experimental results also demonstrate that optimizations such as eliminating the occurs checks are more important than previously thought. These results provide interesting insights into efficient implementations of dependently typed systems in general, and can provide guidance for future implementations.

3 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 logic programming, for example, we need to select all clauses from the program which unify with the current goal. In tabled logic programming we memoize intermediate goals in a table and reuse their results later in order to eliminate redundant and infinite computation. Here we need to find all entries in the table such that the current goal is a variant or an instance of the table entry and re-use the associated answers. If there is no such table entry, we need to add the current goal to the table.

To address this problem, different indexing techniques have been proposed for first-order terms (see [28] for a survey), 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 [23]. Substitution tree indexing is a highly successful technique in first-order theorem proving, which allows the sharing of common sub-expressions via substitutions. This work extends first-order substitution tree indexing [8] to the higher-order setting.

Consider specifying well-know equivalence preserving transformation in first-order logic. In this example, we must represent formulas such as $\forall x.(A(x) \wedge B)$ or $\forall x.(C \wedge D(x))$. These formulas can be represented as terms using higher-order abstract syntax. The first one corresponds to $(\texttt{all}\ \lambda x.(A\ x\ \texttt{and}\ B))$ and the second one to $(\texttt{all}\ \lambda x.(C\ \texttt{and}\ D\ x))$. Inspecting the terms closely, we observe that they share a lot of structure which can be described by the following skeleton: $(\texttt{all}\ \lambda x.(*_1\ \texttt{and}\ *_2))$. We can obtain the first term by instantiating $*_1$ with the term $(A\ x)$ and $*_2$ with the term B. Similarly we can obtain the second term by by instantiating $*_1$ with the term C and $*_2$ with the term $(D\ x)$. Note that $*_1$ and $*_2$ are instantiated with open terms which are allowed to refer to the bound variable x. Our goal is to share subexpressions even in the presence of binders and instantiate holes denoted by $*$ by replacement. How could this be done? Computing the skeleton of two terms relies on finding 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. Moreover, retrieving all terms, which unify or match, needs to be simple and efficient – but higher-order unification is undecidable in general. Although, most specific generalizations exist for higher-order patterns and higher-order pattern unification [12, 20] is decidable, experience with these algorithms demonstrates that they 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 [23]. Contextual modal type theory provides a clean theoretical framework to describe and reason about holes and instantiations with open terms. 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 [23].

4 Conclusion

We have developed several important techniques using contextual modal type theory as a uniform framework. This allows a clean concise theoretical description which clarifies many higher-order issues related to bound variable dependencies. Moreover we have implemented and experimentally evaluated our techniques within the logical framework Twelf. Our results show that the presented techniques taken together 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 [3]. 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.

Finally, the application of logical frameworks to certified code raises new question, which traditionally have not played a central role in logic programming. One of the main ideas in certified code is not only to verify that a program is safe, but also to efficiently transmit and then check the proof. In [16, 29] the authors explore the novel use of higher-order logic programming for checking the correctness of a certificate. To reduce the proof size, the certificate encodes the non-deterministic choices of a higher-order logic programming interpreter as a bit-string. To reconstruct and check the proof, we rerun a deterministic higher-order logic programming interpreter guided by the certificate.

Last but not least, programming language researchers [5] have recently strongly emphasized the need for formalizing and experimenting with programming language designs. Higher-order logic programming environments are ideally suited for this kind of endeavor, since they allow high-level declarative descriptions and execution of formal specifications. Our community also has already a lot of experience in verifying meta-properties such as determinism, termination, or totality which provide valuable insights into properties of these formal specifications. Hence, we see exciting opportunities for encoding and experimenting with the meta-theory and semantics of programming languages within higher-order logic programming environments and applying logic programming technology to this domain.

References

1. Andrew Appel. Foundational proof-carrying code project. personal communication.
2. Andrew Appel. Foundational proof-carrying code. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 247–256. IEEE Computer Society Press, June 2001. Invited Talk.
3. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
4. 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.
5. B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge, 2005.
6. 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.
7. 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.

8. Peter Graf. Substitution tree indexing. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*, Lecture Notes in Computer Science (LNCS) 914, pages 117–131. Springer-Verlag, 1995.
9. 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.
10. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
11. 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.
12. 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.
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. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. 2005.
16. 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.
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. Higher-order substitution tree indexing. In C. Palamidessi, editor, *19th International Conference on Logic Programming, Mumbai, India*, Lecture Notes in Computer Science (LNCS 2916), pages 377–391. Springer-Verlag, 2003.
24. Brigitte Pientka. Tabling for higher-order logic programming. In *20th International Conference on Automated Deduction (CADE), Talinn, Estonia*, volume 3632 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.
25. Brigitte Pientka. Eliminating redundancy in higher-order unification: a lightweight approach. In U. Furbach and N. Shankar, editors, *Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, USA*, Lecture Notes in Artificial Intelligence (LNAI), page to appear. Springer-Verlag, 2006.
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 Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 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. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.
29. Karl Crary Susmit Sarkar, Brigitte Pientka. Small proof witnesses for lf. In Maurizio Gabbrielli and Gopal Gupta, editors, *21th International Conference on Logic Programming, Sitges, Spain*, volume 3668 of *Lecture Notes in Computer Science (LNCS)*, pages 387–401. Springer-Verlag, 2005.