System Description: Teyjus—A Compiler and Abstract Machine Based Implementation of λProlog

Gopalan Nadathur and Dustin J. Mitchell

Department of Computer Science
University of Chicago
Ryerson Hall, 1100 E 58th Street
Chicago, IL 60637
gopalan@cs.uchicago.edu, djmitche@cs.uchicago.edu

Abstract. The logic programming language λProlog is based on the intuitionistic theory of *higher-order hereditary Harrop formulas*, a logic that significantly extends the theory of Horn clauses. A systematic exploitation of features in the richer logic endows λProlog with capabilities at the programming level that are not present in traditional logic programming languages. Several studies have established the value of λProlog as a language for implementing systems that manipulate formal objects such as formulas, programs, proofs and types. Towards harnessing these benefits, methods have been developed for realizing this language efficiently. This work has culminated in the description of an abstract machine and compiler based implementation scheme. An actual implementation of λProlog based on these ideas has recently been completed. The planned presentation will exhibit this system—called *Teyjus*—and will also illuminate the metalanguage capabilities of λProlog.

1. Introduction. In work going back over a decade, Miller, Nadathur and colleagues have studied the proof-theoretic foundations of logic programming. A result of these investigations is the establishment of the intuitionistic theory of *higher-order hereditary Harrop formulas* as a suitable basis for this paradigm of programming [8]. This class of formulas enriches that of Horn clauses—the traditional basis for logic programming—with the possibilities of quantifying over function and (certain occurrences of) predicate variables, of explicitly representing binding in terms and of using a fuller complement of connectives and quantifiers. The λProlog language [13] is based on the logic of higher-order hereditary Harrop formulas. By systematically exploiting the new features in the underlying logic, λProlog provides support at the programming level for capabilities such as higher-order programming, polymorphic typing, scoping over names and procedures, modular programming, abstract data types and the use of lambda terms as data structures. Much research has been conducted in recent years towards fully understanding the usefulness of these additions to logic programming. Two aspects that have received special attention are the availability of lambda terms for representing objects and of a suitable set of primitives for manipulating such representations. These features enable λProlog to support the notion of *higher-order abstract syntax* [18] that is a profitable way to view the syntax of objects whose structure involves binding. Several detailed studies (e.g. [3], [5], [7]) have indicated the utility of λProlog in building systems that manipulate formal objects such as formulas, programs, proofs and types, making it comparable to other recently proposed metalanguages and logical frameworks such as Coq [1], Elf [17] and Isabelle [16]. Fuelled by these applications, Nadathur and colleagues have investigated methods for providing an efficient and robust realization of the language. This work has culminated recently in an actual implementation called *Teyjus*. We provide some insight here into the overall implementation scheme and its realization.

2. The λProlog Abstract Machine. An integral part of the implementation scheme is an abstract machine that is capable of realizing the operations that arise in typical λProlog programs efficiently. As with other logic programming languages, unification and backtracking are intrinsic to λProlog and the Warren Abstract Machine [19] provides a basic structure for treating these aspects well. However, an extensive embellishment of this framework is needed for realizing the following additional features satisfactorily:

- The language contains primitives that can alter the name space and the definitions of procedures in the course of execution. This means, in particular, that unification has to pay attention to changing signatures and that the solution to each (sub)goal has to be relativized to a specific program context.
- Lambda terms are used in λProlog as *data structures*. A representation must therefore be provided for these terms that permits their structures to be examined and compared in addition to supporting reduction operations efficiently.
- Higher-order unification is used in an intrinsic way in the language. This operation has a branching character that must be supported. Further, it is sometimes preferable to delay the solution to unification problems and so a good method is needed for transmitting these across computation steps.
- In addition to having a role in determining program correctness, types are relevant to the dynamic behavior of programs. A sensible scheme must therefore be included for carrying these along at run-time.
- Programming in the language is done relative to modules. In realizing this feature, it is necessary to support certain operations for composing modules. Moreover, a mechanism must be provided for periodically adding and removing code depending on which modules are in use.

The abstract machine that has been developed includes devices for treating all these aspects well. The solution to the problem of changing signatures is based on an elegant scheme for tagging constants and variables and using these tags in unification [9]. To realize changing program contexts, a fast method has been designed for adding and removing code that is capable also of dealing with backtracking [11]. The code that needs to be added may sometimes contain global variables and this possibility has been dealt with by an adaptation to logic programming of the idea of a closure. To facilitate a sensible representation of lambda terms, a new notation has been designed for these terms that utilizes the scheme of de Bruijn for eliminating names and that additionally supports an incremental calculation of reduction substitutions [15]. This notation has then been deployed systematically in the low-level steps contained in the abstract machine [10]. A method has been devised for the purpose of representing suspended unification problems [12] that has several interesting features. For example, based on the observation that new such problems arise out of incremental changes to old ones, the scheme is designed to support sharing while still making it possible to rapidly reinstate a previous unification problem upon backtracking. The treatment of higher-order unification itself includes compilation and prioritizes deterministic computations while delaying the truly non-deterministic parts [12]. This approach makes it possible, for instance, to realize a generalization of first-order unification in a completely deterministic fashion. The technique that is adopted for propagating types at run-time uses information already present during compilation to reduce substantially the effort to be expended dynamically [6]. Using this approach, virtually *no* new computation is required relative to a monomorphic subset of the language that is similar to Prolog. Finally, towards supporting modular programming, a method has been designed for realizing module interactions that permits separate compilation [14]. The actual addition and removal of modules of code can be achieved by methods developed for handling scoping over program clauses. However, these methods have been embellished by efficient devices for determining if a block of code will be redundant in a given context and also by mechanisms for realizing information hiding.

3. Realizing the Implementation Scheme. An implementation of λProlog on stock hardware based on the above ideas envisages four software subsystems: a compiler, a loader, an emulator for the abstract machine and a user interface. The function of the compiler is to process any given module of λProlog code, to certify its internal consistency and to ensure that it satisfies a promise determined by an associated signature and, finally, to translate it into a byte code form consisting of a 'header' part relevant to realizing module interactions and a 'body' containing sequences of instructions that can be run on the abstract machine. The purpose of the loader is to read in byte code files for modules that need to be used, to resolve names and absolute addresses using the information in the header parts of these files and to eventually produce a structure consisting of a block of code together with information for linking this code into a program context when needed. The emulator provides the capability of executing such code after it has been linked. Finally, the user interface allows for a flexibility in the compilation, loading and use of modules in an interactive session.

The Teyjus system embodies all the above components and comprises about 50,000 lines of C code. The functionality outlined above is realized in its entirety in a development environment. Also supported is the use of the compiler on the one hand and the loader and emulator on the other in standalone mode. The system architecture actually makes byte code files fully portable. Thus, λProlog modules can be distributed in byte code form, to be executed later using only the loader/emulator. Finally, the system includes a disassembler for the purpose of viewing the results of compilation.

4. Related Languages and Implementations. Logic and functional programming languages have been used often in the role of metalanguages. However, λProlog and Elf are, to our knowledge, the only languages that systematically support the higher-order abstract syntax approach. The language Qu-Prolog [4] incorporates a partial understanding of binding: in particular, it permits binding operators to be identified and recognizes equality modulo α-conversion but does not support unification relative to the full set of λ-conversion rules and does not also include primitives for recursing over binding structure.

The λProlog language has received several implementations in the past. All but one of these implementations have been in the form of interpreters, the most recent one being the Terzo interpreter [20] that is written in Standard ML. The interpreter based implementations have not profited from an in-depth analysis of the issues that are peculiar to realizing λProlog and, consequently, it is not reasonable to compare their design with that of the Teyjus system. The only other implementation to adopt a compilation approach is Prolog/MALI [2]. The core of this implementation is a memory management system called MALI that offers functionalities relevant to realizing logic programming languages. To a first approximation, λProlog programs are compiled to C programs that execute MALI commands for creating and interpreting goal structure. Certain inefficiencies, such as the copying of goal structure, appear to be inherent to this approach as opposed to the abstract machine based approach used in Teyjus. There are also differences in the treatment of specific aspects such as the representation and runtime manipulation of types, the representation of terms, the realization of reduction, and the implementation of the scoping primitives. Finally, Prolog/MALI appears to support a different notion of modularity. It is of interest to analyze the impact of all these differences and also to quantify their effect on performance, but a treatment of this issue is beyond the scope of this paper.

5. The System Presentation. We will demonstrate the Teyjus system and will expose the λProlog language as embodied in this implementation. We will also discuss examples that indicate the metalanguage capabilities of λProlog.

6. Acknowledgements. B. Jayaraman, K. Kwon and D.S. Wilson have assisted in the design of the λProlog abstract machine. L. Headley, S.-M. Perng and G. Tong have participated in the implementation effort. Support for this work has been provided by NSF at different stages under the grants CCR-8905825, CCR-9208465, CCR-9596119 and CCR-9803849.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

2. Pascal Brisset and Olivier Ridoux. The compilation of λProlog and its execution with MALI. Publication Interne No 687, IRISA, Rennes, November 1992.
3. Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
4. Richard A. Hagen and Peter J. Robinson. Qu-Prolog 4.3 reference manual. Technical Report 99-03, Software Verification Research Centre, School of Information Technology, University of Queensland, 1999.
5. John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
6. Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.
7. Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388. IEEE Computer Society Press, September 1987.
8. 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.
9. Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.
10. Gopalan Nadathur. An explicit substitution notation in a λProlog implementation. Technical Report TR-98-01, Department of Computer Science, University of Chicago, January 1998.
11. Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.
12. Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.
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. MIT Press, August 1988.
14. Gopalan Nadathur and Guanshan Tong. Realizing modularity in λProlog. Technical Report TR-97-07, Department of Computer Science, University of Chicago, August 1997. To appear in *Journal of Functional and Logic Programming*.
15. Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
16. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
17. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
18. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
19. D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.
20. Philip Wickline and Dale Miller. The Terzo 1.1b implementation of λProlog. Distribution in NJ-SML source files. See http://www.cse.psu.edu/ dale/lProlog/., April 1997.