System Description: Twelf — A Meta-Logical Framework for Deductive Systems

Frank Pfenning and Carsten Schürmann*

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

Abstract. Twelf is a meta-logical framework for the specification, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. It relies on the LF type theory and the judgments-as-types methodology for specification [HHP93], a constraint logic programming interpreter for implementation [Pfe91], and the meta-logic $\mathcal{M}_2$ for reasoning about object languages encoded in LF [SP98]. It is a significant extension and complete reimplementation of the Elf system [Pfe94].

Twelf is written in Standard ML and runs under SML of New Jersey and MLWorks on Unix and Window platforms. The current version (1.2) is distributed with a complete manual, example suites, a tutorial in the form of on-line lecture notes [Pfe], and an Emacs interface. Source and binary distributions are accessible via the Twelf home page `http://www.cs.cmu.edu/~twelf`.

1 The Twelf System

The Twelf system is a tool for experimentation in the theory of programming languages and logics. It supports a variety of tasks which we explain in this section: *specification* of object languages and their semantics, implementation of *algorithms* manipulating object-language expressions and deductions, and formal development of the *meta-theory* of an object language. Several extensive experiments have been conducted with Twelf, such as the formal development of the theory of logic and functional programming languages and various logics.

Specification. Twelf employs the representation methodology and underlying type theory of the LF logical framework. Expressions are represented as LF objects using the technique of *higher-order abstract syntax* whereby variables of an object language are mapped to variables in the meta-language. This means that common operations, such as renaming of bound variables or capture-avoiding substitution are directly supported by the framework and do not need to be programmed anew for each object language.

* This work was supported by NSF Grant CCR-9619584.

For semantic specification LF uses the *judgments-as-types* representation technique. This means that a derivation is coded as an object whose type represents the judgment it establishes. Checking the correctness of a derivation is thereby reduced to type-checking its representation in the logical framework (which is efficiently decidable).

Algorithms. Generally, specification is followed by implementation of algorithms manipulating expressions or derivations. Twelf supports the implementation of such algorithms by a constraint logic programming interpretation of LF signatures, a slight variant of the one originally proposed in [Pfe91] and implemented in Elf [Pfe94]. The operational semantics is based on goal-directed, backtracking search for an object of a given type.

Meta-Theory. Twelf provides two related means to express the meta-theory of deductive systems: higher-level judgments and the meta-logic $\mathcal{M}_2$.

A higher-level judgment describes a relation between derivations inherent in a (constructive) meta-theoretic proof. Using the operational semantics for LF signatures sketched above, we can then execute a meta-theoretic proof. While this method is very general and has been used in many of the experiments mentioned below, type-checking a higher-level judgment does not by itself guarantee that it correctly implements a proof.

Alternatively, one can use an experimental automatic meta-theorem proving component based on the meta-logic $\mathcal{M}_2$ for LF [SP98]. It expects as input a Π_2 statement about closed LF objects over a fixed signature and a termination ordering and searches for an inductive proof. If one is found, its representation as a higher-level judgment is generated and can then be executed.

Even though a number of the theorems in the example suites described below can be proven automatically, we consider the meta-theorem prover to be in a preliminary state. Its main current limitations are the lack of automatic appeal to lemmas and the restriction to reasoning only about closed expressions. We are presently extending both the meta-logic $\mathcal{M}_2$ and its implementation to overcome these limitations.

Example Suites. Twelf has been employed for a number of experiments in the area of programming languages and logics [Pfe96]. Many of these are contained in the example suite which is distributed with the Twelf system. Some of the examples contain fully automated proofs, others only their implementations as higher-level judgments.

One of the most well-developed case studies is Mini-ML. We prove value soundness, type preservation, and compiler correctness with respect to different abstract machines. This case study is fully explained and developed in [Pfe], as is a related development of pure logic programming. Other examples include natural deduction, axiomatic logical systems, sequent calculi for classical and intuitionistic logic, proofs of cut-elimination, Cartesian closed categories, and a proof of the Church-Rosser theorem.

2 Implementation

The implementation of Twelf comprises three major parts which we sketch in this section. At the heart is the *core type theory* which provides the necessary infrastructure for the representation of specifications, algorithms, and meta-theory. Algorithms can be executed by the *constraint logic programming* engine, and the development of the meta-theory is supported by the *meta-theorem proving* component.

Core Type Theory. The core of the implementation consists of a dependently typed λ-calculus λ^{Π} extended with notational definitions and existential variables. In contrast to earlier implementations, terms are represented in spine notation [PS98] with explicit substitutions, where we take advantage of normal forms in the $\lambda\sigma$-calculus. Composition of substitutions is a defined function rather than a constructor subject to rewrite rules. Presently, we have not yet undertaken the empirical analysis necessary for optimization, but we found explicit substitutions to be a useful organizing force in the structure of the implementation.

The most frequently used operations are weak head reduction and unification; they are employed for type reconstruction, logic program execution, and theorem proving. The simply-typed case of our unification algorithm is described in [DHKP96]; the dependently typed case is very similar. In short, equations which fall within the fragment of higher-order patterns are solved eagerly, while all other equations are postponed as constraints. Such constraints are indexed on their head variable and are reawakened when this variable is instantiated, which means no overhead for dormant constraints. Instantiation of variables during unification is implemented as an effect which is trailed so it can be undone during backtracking.

Twelf supports notational, non-recursive definitions. They are typically used as syntactic sugar for expressions or to abbreviate derivations of lemmas and theorems. Definitions are checked for strictness [PS98], which means we can often avoid expanding them during unification without sacrificing soundness or completeness. Briefly, a definition is strict if its expansion cannot discard any of its arguments. This guarantees that during unification definitions must only be expanded in the case that the heads of two expressions clash.

A separately implemented type checker is designed to verify the validity of terms in the core type theory without relying on complex algorithms for unification or search. It has been internally employed during the development of the Twelf system, and can be activated upon request when using Twelf.

Constraint Logic Programming. The logic programming interpreter for LF signatures works with a rudimentary compiled form. A compiled signature may be executed on an abstract machine based on a continuation-passing interpreter which maintains higher-order equational constraints. The implementation of this interpreter is presently straightforward and less efficient then the previous Elf implementation.

Twelf provides two tools to verify that the operational reading of a signature as a logic program is consistent: a mode checker and a termination checker. Both are helpful in the implementation of algorithms, since they allow static detection of common programming errors which escape the type-checker, such as misspelled variable names or incorrect subgoal ordering.

The mode checker verifies that the roles of input and output arguments to a predicate are respected throughout the program. The termination checker is given an extension of the subterm ordering on higher-order terms [RP96] and verifies that each well-moded query eventually either fails or yields a solution. Extensions can either be *lexicographic* or *simultaneous*. The implementation also allows termination orderings across several mutually recursive predicates.

A third tool to check that all cases for a set of input variables to a predicate are covered is currently in preparation. Together with mode and termination checking this can verify that a predicate implements a possibly non-deterministic function.

Meta-Theorem Proving. The meta-theorem proving module implements a special-purpose inductive theorem prover for deductive systems [SP98]. It requires the meta-theorem and a termination ordering which expresses the induction ordering, but is otherwise completely automatic. In particular, it does not support tactic-style or interactive theorem proving.

The prover chains together three basic operations: *filling*, *splitting*, and *recursion*. Filling uses an iterative-deepening variant of the constraint logic programming interpreter to perform direct search. Splitting performs complete case analysis based on the (possibly dependent) type of a variable. It requires unification and exploits a static subordination relation on type families to remove spurious parameter dependencies. Finally, there is recursion which appeals to the available induction hypotheses according to the given induction ordering.

The theorem prover generates explicit representations of the proofs it finds as higher-order judgments in LF. These are guaranteed to satisfy mode, termination, and coverage properties and can be safely executed as logic programs.

3 Environment

While Twelf is implemented in ML it is executed as a stand-alone program rather than within the ML top-level loop. This is feasible, since meta-programming is carried out in type theory itself via a logic programming interpretation, rather than in ML as in many other proof development environments. The most effective way to interact with Twelf is as an inferior process to Emacs. The Emacs interface, which has been tested under XEmacs, FSF Emacs, and NT Emacs, provides an editing mode for Twelf source files and commands for incremental type checking, logic program execution, and theorem proving. Moreover it provides utilities for jumping to error locations and tagging and maintaining configurations of source files.

The most sophisticated feature of the front end is type reconstruction, which is based on a duality between implicit quantification and implicit arguments, thereby reducing a type reconstruction problem to unification. With the unification algorithm sketched above, this means our system will always either report a principal type, a type error, or, in the case of unresolved constraints, an indication that the source term contained insufficient type annotations. This simple technique has shown itself to be surprisingly effective and often leads to precise error location information.

Acknowledgements. We would like to thank Iliano Cervesato for his contributions to the logic programming component of Twelf.

References

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[HHP93] 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.

[Pfe] Frank Pfenning. *Computation and Deduction*. Cambridge University Press. In preparation. Draft from April 1997 available electronically.

[Pfe91] 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.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*. Springer-Verlag LNCS, 1998. To appear.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.