Elf: A Meta-Language for Deductive Systems

(System Description)

Frank Pfenning[⋆]

Department of Computer Science,
Carnegie Mellon University,
Pittsburgh, PA 15213, U.S.A.

1 Overview

Elf is a uniform meta-language for the formalization of the theory of programming languages and logics. It provides means for

1. specifying the abstract syntax and semantics of an object language in a natural and direct way;
2. implementing related algorithms (*e.g.*, for type inference, evaluation, or proof search); and
3. representing proofs of meta-theorems about an object language, its semantics, and its implementation.

Its conceptual basis are *deductive systems* which are used pervasively in the study of logic and the theory of programming languages. Logics and type systems for programming languages, for example, are often specified via inference rules. Structured operational semantics and natural semantics also employ deductive systems, and other means for semantic specification (for example, by rewrite rules) can be easily cast into this framework. Many meta-theorems can be proved by induction over the structure of derivations.

Elf's formal foundation is the logical framework LF [5] in which systems of natural deduction can be concisely represented. LF employs the *judgments-as-types* encoding technique for the representation of derivations in a type theory with dependent types. In addition, Elf provides sophisticated type reconstruction and a constraint logic programming interpretation for LF. The latter allows the execution of algorithms when expressed as deductive systems. Proofs of meta-theorems can be represented concisely as higher-level judgments relating derivations.

The most complete reference describing the Elf language is [10]. Gentler introductions can be found in [12] and [6]. Elf has also been used in a graduate course on the theory of programming languages. A draft of the course notes may be available from the author upon request. Below we provide a brief overview of how specification, implementation, and meta-theory tasks are supported in the Elf language. The subsequent sections list some case studies and describe the implementation of Elf.

Object Language Specification. LF generalizes first-order terms by allowing objects from a dependently typed λ-calculus to represent object language expressions. This allows variables in the object language to be represented by variables in the meta-language, using the technique of *higher-order abstract syntax.* Common operations

[⋆] Internet address: fp@cs.cmu.edu

(*e.g.*, renaming of bound variables or substitution) and side-conditions on inference rules (*e.g.*, occurrence restrictions on variables) are thus built into the framework and do not need to be coded up anew for each object language. Another important advantage of this technique is that it greatly simplifies the representation of meta-theoretic proofs.

For semantic specification LF uses the *judgments-as-types* representation technique: a judgment J is represented by a type A and a derivation $\mathcal{D}$ of J by an object M of type A. Such an encoding is *adequate* if there is a bijection between canonical LF objects of type A and derivations $\mathcal{D}$ of J. For adequate encodings, checking the validity of a proposed derivation $\mathcal{D}$ for a judgment J is reduced to type-checking the representation M of $\mathcal{D}$ in the LF type theory. As type-checking for LF is decidable, this yields an effective procedure for checking derivations.

In combination with higher-order abstract syntax, this technique also allows direct representation of *parametric* (sometimes called *generic*) and *hypothetical judgments* without cumbersome, explicit side-conditions on inference rules.

Object Language Implementation. Once a language and its semantics have been represented in LF, one would often like to program and execute related algorithms. In other framework implementations, these algorithms are typically written in a different (meta-)meta-language such as ML, for example using tactics and tacticals. This has the disadvantage that it is difficult, if not impossible, to reason formally about these algorithms.

Thus we pursue another approach in which deductive systems are given an operational interpretation. This has been inspired by the logic programming language λProlog [8] which gives an operational interpretation to hereditary Harrop formulas. Similarly, Elf gives an operational interpretation to types. While λProlog aspires to be a general purpose programming language and thus includes non-logical features such as cut, primitives for input and output, *etc.*, Elf remains pure. Consequently, not all algorithms can be expressed faithfully or implemented efficiently, and Elf should be considered a prototyping and experimentation tool. On the other hand the purity and simplicity of the language enables us to represent proofs of some properties of Elf programs within Elf itself.

Our experience with Elf has also shown that in many cases specifications themselves are executable. A good example of this phenomenon is natural semantics, which is usually structured so that goal-oriented search for a derivation as performed by the Elf interpreter corresponds directly to evaluation in the object language. We would like to emphasize, however, that this is usually not the case for logics: an encoding of natural deduction for a logic does not automatically give rise to a search procedure. Theorem provers for object logics have to be programmed explicitly. Standard techniques (*e.g.* iterative deepening, or tactics and tacticals) are readily implementable in Elf. For truly interactive theorem proving, some ML programming is also necessary due to the lack of input and output primitives in Elf.

Meta-Theory. Many proofs in the theory of programming languages proceed by induction over the structure of terms or derivations. It is well-known that such proofs give rise to primitive recursive functions. Unfortunately, the presence of induction principles or primitive recursive function objects conflicts with higher-order abstract syntax and the central encoding techniques for parametric and hypothetical judgments.

Instead, we continue to follow ideas from logic programming by representing the functions which could be extracted from meta-proofs as higher-level judgments relating derivations. This captures the computational contents of meta-proofs, *i.e.*, they can be executed. While type-checking in Elf guarantees local consistency of this kind of representation, it cannot guarantee that a judgment represents a total function. Thus, while we can implement, partially verify, and execute the meta-theory of deductive systems, at present Elf cannot guarantee the validity of a meta-proof. We are currently implementing a *schema-checker* that would verify that higher-level judgments follow a schema akin to primitive recursion, thus, in combination with the type-checker, verifying meta-proofs. Some preliminary ideas related to schema-checking can be found in [13].

2 Case Studies

A number of case studies have been carried out using Elf, most of which are distributed with the Elf source. Each example deals with a language, some aspects of implementation, and some meta-theory. Among these examples are various logics and logical interpretations, following the ideas laid out in [5]. We have also investigated the theory of logic programming and uniform proofs in this context. We have further implemented a small functional language with polymorphism and recursion and proved various properties such as type preservation [6]. For a proof of compiler correctness for essentially the same language, see [4]. Penny Anderson [1] has implemented a constructive logic, an extraction procedure for functional programs, and some aspects of the correctness proof for this procedure. We have also implemented a proof of the Church-Rosser theorem for the untyped λ-calculus [9]. Other unpublished experiments include type reconstruction for the polymorphic λ-calculus, a proof of the equivalence of Cartesian Closed Categories and the simply-typed λ-calculus (A. Filinski), an implementation of Monads (W. Gehrke), and a correctness proof for CPS conversion (O. Danvy).

3 The Elf Implementation

Elf is implemented in Standard ML of New Jersey; only a few minor changes would be required for other implementations of SML. The implementation is highly modular, taking advantage of the module system of SML. The core of the implementation is a λ-calculus with *type* : *type* which is general enough to encompass the Calculus of Constructions and the LF type theory. Precisely which quantifications are allowed is specified in a separate module, giving rise to various concrete type theories.[1] Building upon this core, we have implemented a constraint solving algorithm [10, 11] for the full core calculus and a pre-unification algorithm [3] for LF. The main constraint solver simplifies equations between typed λ-terms. For reasons discussed in [7], the pre-unification algorithm is currently not in use, although with the ML module system it is easy to construct a system which employs it.

Based on the constraint solver, we have implemented an algorithm for type reconstruction, again for the full core calculus. Because of the undecidability of the general

[1] The current implementation is not general enough to encompass all Pure Type Systems.

reconstruction problem [2], the algorithm will either report a principal type, a type error, or an indication that the source term contained insufficient type information for unambiguous reconstruction. It is always possible to add enough types to the source so that the typing becomes unambiguous.

Type reconstruction for Elf is practical: parsing and checking the types of the meta-proof of the Church-Rosser theorem for the untyped λ-calculus, for example, takes about 5 seconds on a SparcStation IPX. This proof, described in [9], consists of 1852 words (374 lines) of Elf source code. After type reconstruction, the proof consists of 3922 words (439 dense lines). On many examples, the fully typed source expressions will be 3–5 times larger than the actual input.

The logic programming module is specific to Elf. It implements an interpreter using an interactive top-level similar to that of Prolog. Ignoring some of the more advanced features, one may think of it as a typed version of a Prolog interpreter which maintains derivations of queries in the form of λ-terms. Queries can be of the form

`?- M : A.` *or* `?- A.`

The type `A` represents a judgment, the object `M` its derivation. If `M` is given, this represents a type-checking query; if `M` is a free variable, the query triggers search for an object of type `A`, *i.e.*, an object representing a derivation of the judgment represented by `A`. In the second form, the derivation `M` need not be explicitly constructed, which can be significantly more efficient than the first form. Queries in either form may contain free variables that may be instantiated during constraint simplification or by resolution. Search proceeds in a depth-first fashion as in Prolog and is thus predictable though incomplete as a theorem prover. Unlike λProlog's higher-order unification, the constraint solver will never make non-deterministic choices; all choices are made during subgoal and clause selection. As a consequence, constraints may remain even after execution, where each solution to the constraints (possibly none) yields an answer to the original query.

Even though Elf does not compile programs or queries, it applies some standard optimizations known from logic programming interpreters and a few which are specific to Elf. Among the general optimizations are clause indexing, elimination of some unnecessary occurs-checks, and the use of efficient global symbol tables. Among the specific optimization are avoidance of redundant unifications and elimination of unnecessary proof objects. Rationales for the various implementation design decisions and an empirical analysis of the runtime behavior of Elf programs can be found in [7].

Together with the implementation we distribute some Emacs Lisp files that allow Elf to run as an inferior process to GNU Emacs. This includes a recently completed incremental type-checker which allows the programmer to check individual declarations, thus tightening the usual feedback loop of editing, type-checking a whole file, locating the source of a possible type error, *etc.* It can also be used to query types of subterms or to obtain a menu of possible constructors for objects valid in a given subterm location.

Elf can be retrieved by anonymous ftp from `alonzo.tip.cs.cmu.edu`, directory `/afs/cs/user/fp/public`. Copies of papers related to Elf can be found in the `elf-papers` subdirectory. Please see the `README` files for further information. There is also a mailing list with announcements regarding Elf; please send mail to `elf-request@cs.cmu.edu` to join the list.

Acknowledgments. The author would like to acknowledge the contributions of Spiro Michaylov, Ekkehard Rohwedder, Conal Elliott, and Ken Cline to the Elf implementation. This work was supported in part by NSF Grant CCR-9303383 and by the U.S. Air Force under Contract F33615-90-C-1465, ARPA Order No. 7597.

References

1. Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, October 1993. Available as Technical Report CMU-CS-93-206.
2. Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.
3. Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, April 1989. Springer-Verlag LNCS 355.
4. John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.
5. 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.
6. Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.
7. 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.
8. Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.
9. Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 199? To appear.
10. 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.
11. 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.
12. Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, Massachusetts, 1992.
13. Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.