The Practice of Logical Frameworks

Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

fp@cs.cmu.edu

1 Introduction

Deductive systems, given via axioms and rules of inference, are a common conceptual tool in mathematical logic and computer science. They are used to specify many varieties of logics and logical theories as well as aspects of programming languages such as type systems or operational semantics. A *logical framework* is a meta-language for the specification of deductive systems. Research on logical frameworks is still in its infancy. Nonetheless, different frameworks have been proposed, implemented, and applied to a variety of problems. In addition, some general reasoning systems have been used to study deductions as mathematical objects, without specific support for the domain of deductive systems. This short survey cannot be complete, but we will try to highlight the major themes, concepts, and design choices for logical frameworks and provide some pointers to the literature. We concentrate on systems designed specifically as frameworks and among them on those that have the most mature and most heavily exercised implementations: hereditary Harrop formulas (implemented in λProlog and Isabelle) and the LF type theory (implemented in Elf). The interested reader is referred to the homepage on logical frameworks at `http://www.cs.cmu.edu/~fp/lfs.html` for a more complete bibliography, and pointers to papers, implementations and researchers in the area.

Logical frameworks are subject to the same general design principles as other programming or specification languages. They should be as simple and uniform as possible, yet they should provide concise means to express the concepts and methods of the intended application domain. Meaningless expressions should be detected statically, yet the language must remain effectively implementable. It should be possible to structure large specifications. There are also concerns specific to logical frameworks. Perhaps most importantly, an implementation must be able to check deductions for their validity with respect to the specification of a deductive system. Secondly, it should be feasible to prove (informally) that the representations of deductive systems in the framework are correct. Our task will be to exhibit some of the central principles underlying logical frameworks and discuss them in terms of their practical consequences. This will help us to motivate what we consider the major challenges faced by the field at present.

Historically, the first logical framework was Automath [NGdV94] and its various languages, developed during the late 60's and early 70's. The goal of the

To appear in the proceedings of CAAP'96, Linköping, Sweden, April 1996

Automath project was to provide a tool for the formalization of mathematics without foundational prejudice. Therefore, the logic underlying a particular mathematical development was an integral part of its formalization. Many of the ideas underlying the Automath languages have found their way into modern systems. The main experiment conducted within Automath was the formalization of Landau's *Foundations of Analysis* [Jut77].

In the early 80's the importance of *constructive type theories* for computer science was being recognized through the pioneering work of Martin-Löf [ML80, ML85a, ML85b]. On the one hand, this led to a number of systems for constructive mathematics and the extraction of functional programs from constructive proofs (for example, Nuprl [C$^+$86, CH90], LEGO [Pol94], and Coq [DFH$^+$93]). On the other hand, it strongly influenced the design of LF [HHP93], also called the Edinburgh Logical Framework (ELF). Concurrent with the development of LF, frameworks based on higher-order logic and resolution were designed in the form of generic theorem provers [Pau86, Pau89, NP92] and logic programming languages [NM88, MNPS91]. The type-theoretic and logic programming approaches were later combined in the Elf language [Pfe89, Pfe91a, Pfe92a, Pfe94]. At this point around 1988, there seems to have been in a pause in the development of new frameworks, while the potential and limitations of existing systems were explored in numerous experiments. This hiatus has recently come to a close with implementations of frameworks based on inductive definitions such as FS0 [Fef88, MSB93] and ALF [Mag95], partial inductive definitions [Hal91, Eri93, Eri94] and substructural logics [SH91, Gir93, Mil94, Cer96]. There is a different approach to logical frameworks based on equational rather than deductive reasoning. While one can be interpreted in the other without much difficulty, the meta-languages based on equational reasoning take a rather different form and we will not discuss them here. The interested reader is referred to [MOM93, KKV93, Hab94]. Another approach derives from the notion of *labelled deductive system* due to Gabbay [Gab94]. Here the meta-logic is a classical logic with equality [Gab93].

Two further notes on terminology. Some researchers distinguish logical frameworks from *meta-logical frameworks* [BC93], the latter being intended as a meta-language for reasoning *about* deductive systems rather than within them. Clearly, the latter is more general, since meta-logical frameworks must first provide means for specifying deductive systems before one can reason about them. We view this as just another application which may or may not be supported in a given framework. Secondly, some researchers prefer the term *general logic* for systems not based on type theory. We do not make this distinction, but we will point out the differences between the methodologies based on logical and type-theoretic frameworks.

This survey is organized along the specific tasks that are carried out within logical frameworks, from the representation of expressions (Section 2) and derivations (Section 3) to methods for search and meta-programming (Section 4) and development of the meta-theory of encoded systems (Section 5). We specifically do not discuss the meta-theory of the logical frameworks themselves.

2 Representing Syntax

The specification of a deductive system usually proceeds in two stages: first we define the *syntax* of the object language and then its *judgments* via *axioms* and *rules of inference*. In order to concentrate on the meanings of expressions we ignore issues of concrete syntax and parsing and concentrate on specifying *abstract syntax*. Different framework implementations provide different means for modifying or customizing the parser in order to embed the desired object-language syntax.

One of the simplest meta-languages is the language of Horn clauses. In such a framework the expressions of the object language are represented by ordinary first-order terms. This uni-typed approach (with one universal type of individuals) requires that we axiomatize explicitly, via a set of Horn clauses, when a given first-order term actually represents an expression of the object language. Similarly, in a functional meta-language like Scheme, we would have to write a function to check if a given s-expression denotes an expression of the object language. Such definitions are *inductive*, which has advantages for the development of the meta-theory as discussed in Section 5, but they are also external to the representation itself.

A standard method for transforming an *external* validity condition into an *internal* property of the representation is to introduce *types*. By designing the type system so that type checking is decidable, we turn a dynamic invocation of a predicate or function into a static check. In the functional and logic programming communities, many type systems have been proposed, but few of them interact well with higher-order features needed later. We begin with Church's *simply typed λ-calculus* $\lambda^{\to}$. The representation introduces a new type a for each syntactic category of the object language. The *adequacy theorem* states that the representation function $\ulcorner\cdot\urcorner$ is a bijection between expressions of the object language and canonical meta-language objects M of type a. In the realm of logical frameworks, the appropriate notion of canonical form is usually the long $\beta\eta$-normal form.

A critical issue for meta-languages in general is the representation of variables. In informal practice we pervasively use the so-called *variable convention*, that is, we identify expressions that differ only in the names assigned to their bound variables. This can be achieved in the meta-language by using *de Bruijn indices* [dB72] where a variable occurrence is replaced by a pointer to the corresponding binder.

Another approach, which remains closer to informal practice, reduces all binding operators to a single one, namely λ-abstraction in the meta-language. This entails that object language variables are represented by variables in the meta-language, and variables bound in the object language are bound with corresponding scope in the meta-language. This is the fundamental idea of *higher-order abstract syntax* [HHP93, PE88] which goes back to Martin-Löf's system of arities [NPS90]. Higher-order abstract syntax identifies expressions that differ only in the names assigned to their bound variables through α-conversion in the meta-language. It also supports substitution through β-reduction in the

meta-language: Since the representation expresses the scope of all variables, capture-avoiding substitution is automatically available and does not need to be implemented on a language-by-language basis.

While representation of syntax is relatively well understood, current approaches are limited in that they do not permit subsorting. With larger examples it is frequently the case that syntactic categories are not completely disjoint, but that some are subclasses of others. In a first-order meta-language this problem can be addressed by using an *order-sorted* type system, but in higher-order languages these interact, sometimes in undesirable ways, with other features. Preliminary theoretical work to extend order-sorted techniques to logical frameworks is reported in [Pfe93, KP93], but to our knowledge none has been implemented on a realistic scale.

The variable convention is just one example where syntax is considered modulo a certain equivalence. In classical logic, for example, it is sometimes convenient to think of $\neg(A \wedge B)$ as the *same* formula as $\neg A \vee \neg B$. Building equations into the representation of syntax has been recognized as a significant challenge and investigated in the context of logical frameworks by [Nip91] with a rapidly increasing literature regarding its operational properties [ALS94, Kah95, LP95, Pre95].

Finally, *substitutions* are often used in informal developments. Some work on incorporating them directly into frameworks has been done [Dug94, Mag95], but further theoretical and practical issues regarding explicit substitutions remain to be explored.

3 Representing Derivations

The next step is to represent the judgments and the defining axioms and inference rules of the deductive system under consideration. These might be the type system and operational semantics of a programming language, or the inference rules of a logical system in natural deduction, sequent, or axiomatic formulation. We will generally think of axioms as inference rules with no premises, thus no formal distinction between them is required. Important recurring notions from informal practice are *parametric* and *hypothetical judgments*, that is, reasoning from hypotheses and reasoning with parameters as in the implication and universal introduction rules in natural deduction. We should therefore take care to support these notions directly in the framework.

The first choice that arises is if derivations should be modelled as objects, or if we are interested only in derivability. In the latter case we can follow an approach which is analogous to the first one pursued in the representation of syntax: We define derivability via axioms in a meta-logic. Using this technique, a judgment (such as A *is true*) is represented by a formula (such as true($\ulcorner A \urcorner$)). Each inference rule defining a judgment is turned into an axiom in the meta-language. The *adequacy theorem* states that we can prove true($\ulcorner A \urcorner$) in the meta-logic iff we can derive A *is true*.

An appropriate language that supports parametric and hypothetical judgments is the language of *hereditary Harrop formulas*, which forms the basis of the logic programming language λProlog and the generic theorem prover Isabelle. Variations of this approach to encoding derivability have been devised by Paulson [Pau86] and Felty and Miller [FM88]. Quantifiers in hereditary Harrop formulas are typed and range over simply-typed λ-terms, thus permitting the technique of higher-order abstract syntax.

With a general implementation of the meta-logic, we can now reason within the object language and interpret the results via the adequacy theorem. Many experiments have been carried out following this methodology, including type inference [Pfe88], equational reasoning [Nip89], theorem proving [Fel89], functional programming [HM90, Han93], specification languages [MM93], VLSI design [Ros92], set theory [Pau93, Noë93], interpreter verification [BHN$^+$94] and the Church-Rosser theorem [Nip95, Ras95].

However, in many applications we need to go a step further and design a representation of *derivations* themselves as objects in the meta-language. A natural first choice for this representation is also the simply-typed λ-calculus. We introduce a new type for derivations, where each inference rules becomes a constructor of objects of the new type. Functional constructors can be used to model parametric and hypothetical judgments. This technique has been investigated by Felty [Fel89] for representing derivations in first-order logic.

One drawback of this representation is that the validity of derivations must be axiomatized explicitly. This is because the system of simple types is not accurate enough to capture the expressions which are part of a judgment. Fortunately, it is possible to refine the simply-typed λ-calculus so that validity of the representation of derivations becomes an *internal* property, without destroying the decidability of the type system. This is achieved by introducing *type families* indexed by the syntactic constituents of the judgment. Simple function types must be generalized to *dependent function types* to capture the dependencies between an argument to a function and its role as an index object. This representation technique is often summarized with the phrases *judgments-as-types* and *derivations-as-objects*.

Dependent types create the need for a rule of *type conversion* which drastically alters the character of the type theory. The pure type system λ^{Π} preserves decidability of type-checking, which is very easily lost for language extensions. λ^{Π} is the basis of the LF logical framework [HHP93] which also systematizes the representation techniques for various judgment forms and the proofs of adequacy of these representations [Gar92]. Applications of LF have been numerous, first with pencil and paper [HHP93, AHMP92], then in the context of the Elf language which implements LF (see Section 5).

A significant challenge in the area of meta-representation are modal and other non-local side conditions in the formulation of deductive systems, as they occur, for example, in the presentation of linear logic. A higher-order classical linear meta-logic to address some of these problems has been proposed by Miller [Mil94], a conservative extension of LF by Cervesato [Cer96]. Implemen-

tation projects for these languages have just begun. Besides linear and related logics, these frameworks also enable a whole new class of languages to be represented concisely, namely those involving state and concurrency [Chi95]. Also relevant is the work on labelled deductive systems [Gab93, Gab94].

Another challenge is the development of appropriate structuring principles to achieve modular presentation of deductive systems. This has been studied in the abstract [HST94] and in the context of the Elf language [HP92b], but only prototype implementations exist. The ALF framework employs explicit substitutions in a similar structuring role [Mag95].

It is straightforward to encode systems of equational reasoning in any of the logical frameworks we have discussed [Nip89, Fel91]. The granularity and efficiency of reasoning in such explicit encodings is generally too low to allow complex developments—we must look for ways to incorporate equational theories directly into the underlying meta-logic or type theory without sacrificing decidability and other desirable properties. Some promising work in this direction includes reflection [Con94] and dependently typed rewriting [Vir95].

4 Search and Meta-Programming

The representation of a deductive system in a logical framework may be used for a variety of purposes. The obvious application is to construct derivations within a deductive system, with the support of the framework implementation. For example, after specifying a logic for reasoning about programs in a particular programming language, we may now want to prove the correctness of some program. This process typically involves a mixture of interactive and automatic deduction. A related, but qualitatively different task, is the implementation of specific algorithms for the deductive system at hand. For example, after specifying a type system for a programming language as a deductive systems, we may want to implement algorithms for type checking or reconstruction. Similarly, we may wish to implement an abstract machine and a compiler after specifying a high-level semantics for a programming language. A third application is the investigation of the meta-theory of the deductive systems we have encoded. In this section we consider search and meta-programming applications and postpone the meta-theory until Section 5.

It is beyond the reach of current implementations and even undesirable in many circumstances to conduct completely automatic search. We cannot expect to obtain an efficient and powerful automatic theorem prover merely from the specification of a logic as a deductive system, nor can we expect an automatic theorem prover to find good derivations, which is, of course, a subjective notion. Instead, we must look for methods that support interactive deduction while permitting heuristic searches to be programmed and automatic methods to be used when they exist. *Tactics* and *Tacticals* provide a popular mechanism to structure and program search. Tactics and tacticals arose out of the LCF theorem proving effort [GMW79, Pau83] and are used in such diverse systems as NuPrl [C$^+$86], Coq [DFH$^+$93], Isabelle [NP92], and λProlog [NM88, Fel93]. In all but the last

one, they are programmed in ML which was originally developed to support theorem proving for LCF.

Logic programming offers a different approach to meta-programming. Rather than meta-programming in a language in which the logical framework is implemented (typically ML), we endow the logical framework itself with an operational interpretation via goal-directed search in the spirit of Prolog. This means that we are working in a uniform language for specifications and implementations of algorithms, but it should be clear that a specification of a logic under this approach does not automatically give rise to a theorem prover. Two frameworks to date have pursued this approach: λProlog, which gives an operational interpretation to hereditary Harrop formulas, and Elf, which gives an operational interpretation to λ^{Π}.

Unification is a central and indispensable operation in traditional first-order theorem provers and logic programming languages. It plays a critical role in the implementations of tactics and tacticals in Isabelle and λProlog [Fel93, FH94]. Unification allows the search algorithm to postpone existential choices until more information becomes available as to which instances may be useful. Since most logical frameworks go beyond first-order terms, traditional first-order unification is insufficient.

Despite its undecidability, Huet [Hue75] devised a practical algorithm for *higher-order pre-unification*, a form of unification where solvable equations of a certain form are postponed as constraints. Huet's algorithm has been used extensively in λProlog and Isabelle and generally seems to have good computational properties. It also generalizes smoothly from the simply-typed to the dependently typed case, as discovered independently by Elliott [Ell89, Ell90] and Pym [Pym90, Pym92].

The practical success of Huet's algorithm seems to be in part due to the fact that difficult, higher-order unification problems rarely arise in practice. An analysis of this observation led Miller [Mil91] to discover *higher-order patterns*, a sublanguage of the simply-typed λ-calculus with restricted variable occurrences. For this fragment, most general unifiers exist. In fact, the theoretical complexity of this problem is linear [Qia93], just as for first-order unification. Miller proposed it as the basis for a lower-level language L_λ similar to λProlog, but one where unification does not branch, since only higher-order patterns are permitted as terms. An empirical study of this restriction [MP92, MP93] showed that most dynamically arising unification problems lie within this fragment, but that a syntactic restriction rules out some useful programming idioms, since the operation of substitution of terms for bound variables has to be reprogrammed for each syntactic category.

For this reason, the logic programming language Elf uses neither Huet's algorithm nor a static pattern restriction, but a general higher-order constraint simplification algorithm [Pfe91a, Pfe91b]. This algorithm directly solves problems within Miller's decidable fragment, while other equations are postponed as constraints. On the positive side, this can drastically reduce backtracking compared to higher-order unification and imposes no restrictions on variable oc-

currences. On the negative side, unsolvable constraints may remain until the end of the computation, in which case the answer must be considered a conditional solution.

From the considerable practical experience it seems that logic programming is often superior to implement specific algorithms such as for type inference, evaluation, or compilation, while tactics and tacticals work well for general reasoning and search within a specified logic.

Methods for general proof search for LF have been investigated [PW90], but a general and practically efficient theorem proving procedure for a logical framework remains an important area for further research.

Generality, as found in a logical framework, often comes at the price of efficiency. For example, compare the undecidability of higher-order unification with the efficiency of first-order unification. One way to recapture efficiency would be to compile or specialize the general search procedure to specific encoded logics. Only very preliminary work on this has been done [NJ89, MP93].

5 Representing Meta-Theory

Since logical frameworks are designed to express the language and inference rules of deductive systems at a very high level of abstraction, one rightly suspects that they should be amenable to an investigation of the meta-theory of deductive systems. By far the most common proof technique is induction, both over the structure of expressions and derivations. Thus one naturally looks towards frameworks that permit inductive definitions of judgments and support the corresponding induction principles. Unfortunately, induction conflicts with the representation technique of higher-order abstract syntax. For example, expressions of the untyped λ-calculus would be represented by constructors $\mathrm{lam} : (\mathrm{exp} \to \mathrm{exp}) \to \mathrm{exp}$ and $\mathrm{app} : \mathrm{exp} \to \mathrm{exp} \to \mathrm{exp}$. This cannot be considered as an inductive type, because of the negative occurrence of exp in the type of lam. An attempt to formulate a valid induction principle for the type exp fails.

Several options have been explored to escape this dilemma. The first, for example used in [BC93, Fef88, MN94, Pol95] is to reject the notion of higher-order abstract syntax and use inductive representations directly. This is engenders a complication of the encoding and consequently of the meta-theory, which now has to deal with many lemmas regarding variable naming. Using de Bruijn indices [dB72] alleviates this problem somewhat. In fact, this representation was designed in order to be able to give a completely rigorous proof of the Church-Rosser theorem for the untyped λ-calculus. It has subsequently been used in formalizations of this proof in NQTHM [Sha88], Coq [Hue94] and Isabelle [Nip95, Ras95].

Instead of completely rejecting higher-order abstract syntax, we can also relax the notion of inductive definition to obtain *partial inductive definitions* [Hal91]. These have been used as the basis for a logical framework [Eri93], implemented in the Pi derivation editor [Eri94], but its potential for formalizing meta-theory remains largely unexplored.

A third option is to *externalize* the induction. This reflects one of the ideas behind Gödel's system T in the context of type theory: Instead of proving a statement $\forall x.\ \exists y.\ A(x, y)$ explicitly by induction over x, we can exhibit a primitive recursive functional f such that $\forall x.\ A(x, f(x))$. Since all primitive recursive functionals are total (which we prove once and for all), the required y is thus guaranteed to exist. An extension of this idea beyond primitive recursion to general pattern matching (without the notion of higher-order abstract syntax) has been explored in the ALF system [Mag95, MN94, Coq92, CNSvS94]. The empirical evidence suggests that this shortens developments considerably [Coq92] and also allows the formulations of functions in a manner which is closer to functional programming practice.

Adding such functions to the simply-typed λ-calculus or LF still leads to inadequate encodings. To eliminate the paradoxes we can formulate functions of this sort as *higher-level judgments* relating derivations so that they cannot interfere with the encodings themselves. They can still be executed due to the computational interpretation of meta-language Elf via logic programming search. This technique has been applied in a number of case studies such as program derivation [And93, And94a, And94b], type preservation [MP91], compiler verification [HP92a], CPS conversion [DP95], partial evaluation [Hat95], theorem proving [Pfe92b], the Church-Rosser theorem [Pfe92c], and cut elimination [Pfe95].

With this technique we can implement and execute meta-theoretic proofs, but LF type checking alone cannot guarantee that a higher-level type family actually represents a meta-theoretic proof. The design of an appropriate external validity condition for these relations and its implementation is the subject of current research described in [PR92, RP96, Roh96]. Presently, the external argument guaranteeing the meta-theorem has been carried out mechanically only for some of these above-mentioned experiments.

An important challenge for logical frameworks is to reconcile induction principles with higher-order abstract syntax. Two approaches, using existing inductive calculi, are presented in [DH94, DFH95]. Another approach, pursued by the author in joint work with Joëlle Despeyroux and Carsten Schürmann, employs modal restrictions to separate closed from arbitrary expressions, thereby recovering adequacy of encodings in conjunction with a system of primitive recursive functionals for higher-order data representations.

Termination orderings and higher-order, dependently typed rewriting provide tools which should significantly extend the scope of the methods sketched here. Some work along these lines can be found in [Geh95, Kah95, LP95, vdPS95].

High-level representations of deductive systems allow proofs of their properties to be implemented quickly and efficiently. Yet the current degree of automation is not satisfactory. We should look for ways to apply techniques from inductive theorem proving in the realm of logical frameworks to automate some of these proofs.

References

[AHMP92] Arnon Avron, Furio A. Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992. A preliminary version appeared as University of Edinburgh Report ECS-LFCS-87-31.

[ALS94] Jürgen Avenhaus and Carlos Loría-Sáenz. Higher order conditional rewriting and narrowing. In J.-P. Jouannaud, editor, *Proceedings of the First International Conference on Constraints in Computational Logics*, pages 269–284, Munich, Germany, September 1994. Springer-Verlag LNCS 845.

[And93] Penny Anderson. *Program Derivation by Proof Transformation*. PhD thesis, Carnegie Mellon University, October 1993. Available as Technical Report CMU-CS-93-206.

[And94a] Penny Anderson. Program extraction in a logical framework setting. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 144–158, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[And94b] Penny Anderson. Representing proof transformations for program optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, pages 575–589, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[BC93] David A. Basin and Robert L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[BHN+94] Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, and Birgit Schieder. Interpreter verification for a functional language. In P.S. Thiagarajan, editor, *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 77–88. Springer-Verlag LNCS 880, 1994.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, 1996. Forthcoming.

[CH90] Robert Constable and Douglas Howe. NuPrl as a general logic. In P. Odifreddi, editor, *Logic and Computation*. Academic Press, 1990.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[Con94] Robert L. Constable. Using reflection to explain and enhance type theory. In *Proof and Computation*, NATO ASI Series. Springer-Verlag, 1994.

[Coq92] Catarina Coquand. A proof of normalization for simply typed lambda calculus written in ALF. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 85–92, Båstad, Sweden, 1992.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The

Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[DP95] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical Report CMU-CS-95-121, Department of Computer Science, Carnegie Mellon University, February 1995.

[Dug94] Dominic Duggan. Logical closures. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 114–129, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[Ell89] 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.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[Eri93] Lars-Henrik Eriksson. *Finitary Partial Inductive Definitions and General Logic*. PhD thesis, Department of Computer and System Sciences, Royal Institute of Technology, Stockholm, 1993.

[Eri94] Lars-Henrik Eriksson. Pi: An interactive derivation editor for the calculus of partial inductive definitions. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 821–825, Nancy, France, June 1994. Springer Verlag LNAI 814.

[Fef88] Solomon Feferman. Finitary inductive systems. In R. Ferro, editor, *Proceedings of Logic Colloquium '88*, pages 191–220, Padova, Italy, August 1988. North-Holland.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, August 1989. Available as Technical Report MS-CIS-89-53.

[Fel91] Amy Felty. A logic programming approach to implementing higher-order term rewriting. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 135–161, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[FH94] Amy Felty and Douglas Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 605–619, Nancy, France, June 1994. Springer-Verlag LNAI 596.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, Illinois, May 1988. Springer-Verlag LNCS 310.

[Gab93] Dov M. Gabbay. Classical vs non-classical logics: The universality of classical logic. Technical Report MPI-I-93-230, Max-Planck-Institut für Informatik, Saarbrücken, Germany, August 1993.

[Gab94] Dov M Gabbay. Labelled deductive systems, volume 1 — foundations. Technical Report 465, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1994.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Geh95] Wolfgang Gehrke. Problems in rewriting applied to categorical concepts by the example of a computational comonad. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications*, pages 210–224, Kaiserslautern, Germany, April 1995. Springer-Verlag LNCS 914.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[Hab94] Marianne Haberstrau. ECOLOG: An environment for constraint logics. In J.-P. Jouannaud, editor, *Proceedings of the First International Conference on Constraints in Computational Logics*, pages 237–252, Munich, Germany, September 1994. Springer-Verlag LNCS 845.

[Hal91] Lars Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, September 1991.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[Hat95] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 279–298, Utrecht, The Netherlands, September 1995. Springer-Verlag LNCS 982.

[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. A preliminary version appeared in the *Proceedings of the Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

[HP92a] 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.

[HP92b] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*. To appear. A preliminary version is available as Technical Report CMU-CS-92-191.

[HST94] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*,

67:113–160, 1994.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue94] Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. Preliminary version available as INRIA Technical Report 2009, August 1993.

[Jut77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System*. PhD thesis, Eindhoven University of Technology, 1977.

[Kah95] Stefan Kahrs. Towards a domain theory for termination proofs. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications*, pages 241–255, Kaiserslautern, Germany, April 1995. Springer-Verlag LNCS 914.

[KKV93] Claude Kircher, Hélène Kirchner, and Marian Vittek. Implementing computational systems with constraints. In P. van Hentenryck and V. Saraswat, editors, *Proceedings of the First Workshop on Principles and Practice of Constraints Programming*, Newport, Rhode Island, April 1993. MIT Press.

[KP93] Michael Kohlhase and Frank Pfenning. Unification in a λ-calculus with intersection types. In Dale Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 488–505, Vancouver, Canada, October 1993. MIT Press.

[LP95] Olav Lysne and Javier Piris. A termination ordering for higher order rewrite systems. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications*, pages 26–40, Kaiserslautern, Germany, April 1995. Springer-Verlag LNCS 914.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML85a] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML85b] Per Martin-Löf. Truth of a propositions, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, June 1985.

[MM93] D. Méry and A. Mokkedem. Crocos: an integrated environment for interactive verification of SDL specifications. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification: Fourth International Workshop, CAV '92*. Springer-Verlag LNCS 663, 1993.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs*

and Programs, pages 213–237. Springer-Verlag LNCS 806, 1994.

[MNPS91] 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.

[MOM93] Narciso Martì-Oliet and Jose Meseguer. Rewriting logic as a logical and semantical framework. Technical Report SRI-CSL-93-05, SRI International, August 1993.

[MP91] 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.

[MP92] 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.

[MP93] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[MSB93] Seán Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[NGdV94] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[Nip89] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, The Netherlands, July 1991.

[Nip95] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). Unpublished manuscript, July 1995.

[NJ89] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for λProlog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1180–1198, Cleveland, Ohio, October 1989.

[NM88] 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.

[Noë93] Philippe Noël. Experimenting with Isabelle in ZF set theory. *Journal of Automated Reasoning*, 10(1):15–58, 1993.

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag LNAI 607. System abstract.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[Pau83] Lawrence Paulson. Tactics and tacticals in Cambridge LCF. Technical Report 39, University of Cambridge, Computer Laboratory, July 1983.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau93] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[PE88] 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.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe91a] 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.

[Pfe91b] 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.

[Pfe92a] Frank Pfenning. Computation and deduction. Unpublished lecture notes, revised May 1994, May 1992.

[Pfe92b] 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.

[Pfe92c] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[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.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pol95] Robert Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 365–380, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[PR92] 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.

[Pre95] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, Technische Universität München, March 1995.

[PW90] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 236–250, Kaiserslautern, Germany, July 1990. Springer-Verlag LNCS 449.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[Pym92] David Pym. A unification algorithm for the $\lambda\Pi$-calculus. *International Journal of Foundations of Computer Science*, 3(3):333–378, September 1992.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 391–405, Orsay, France, April 1993. Springer-Verlag LNCS 668.

[Ras95] Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, March 1995.

[Roh96] Ekkehard Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[Ros92] Lars Rossen. *A Relational Approach to Sequential VLSI Design*. PhD thesis, Department of Computer Science, Technical University of Denmark, 1992.

[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*, Linköping, Sweden, April 1996. Springer-Verlag LNCS. To appear.

[SH91] Peter Schroeder-Heister. Structural frameworks, substructural logics, and the role of elimination inferences. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 385–403. Cambridge University Press, 1991.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, July 1988.

[vdPS95] J. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 350–364, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[Vir95] Roberto Virga. Higher-order superposition for dependent types. Technical Report CMU-CS-95-150, Carnegie Mellon University, 1995.