LOGICAL ENVIRONMENTS

Edited by

Gérard Huet
Directeur de Recherche, INRIA-Rocquencourt

Gordon Plotkin
Professor of Theoretical Computer Science
University of Edinburgh

CAMBRIDGE
UNIVERSITY PRESS

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First Published 1993

Printed in Great Britain at the University Press, Cambridge

Library of Congress cataloguing in publication data available
British Library cataloguing in publication data available

ISBN 0 521 43312 6

LOGICAL ENVIRONMENTS

Preface

This book is a selection of papers presented at the second annual Workshop held under the auspices of the ESPRIT Basic Research Action 3245, "Logical Frameworks: Design, Implementation and Experiment." It took place in Edinburgh, Scotland from the 19th to the 24th of May, 1991. Ninety-three people attended the Workshop.

We thank the European Community for the funding which made the Workshop possible. We also thank George Cleland and Monika Lekuse who looked after local arrangements, and Claire Jones who put together the preliminary proceedings as an electronic document. Finally, we thank the following researchers who acted as referees: G. Bellin, R. Constable, S. Berardi, Y. Bertot, L. Danos, G. Dowek, H. Geuvers, R. Harper, C. Jones, D. Kesner, J. W. Klop, Y. Lafont, Z. Luo, D. Miller, C. Murthy, T. Nipkow, M. Parigot, C. Paulin-Mohring, R. Pollack, D. Pym, D. Sangiorgi, D. Sannella, P. Scott, I. Smith, and B. Werner.

This volume is a follow-up to the book "Logical Frameworks," which we previously edited, and which was published by Cambridge University Press in 1991.

G. Huet, INRIA-Rocquencourt
G. Plotkin, Edinburgh University

Introduction

The topic of this book lies at the frontier of Mathematical Logic and Computer Science. That Computer Science needs Logic has been well-understood from the very beginning of the subject: we speak of the "logic" of a circuit, alluding to propositional logic as the foundation of one level of hardware description. Software needs more expressive logics in order to describe recursive programs, for they are descriptions of potentially infinite state transition systems. More sophisticated descriptive formalisms, *i.e.* high-level programming languages, need still more sophisticated logics, able to formalise much of constructive mathematics. One then speaks of specification languages, of mathematical semantics, of logics of concurrency, and so on. Thus in the last twenty years the software revolution has resulted in the creation of a major application area for Mathematical Logic.

That Logic needs Computer Science became apparent as a consequence of the heavy use of logic in computer science. The need for software verification generated a need for the automation of large formal proofs. Sophisticated algorithms for deciding propositional calculus, or semi-deciding predicate logic, had to be designed and implemented. A whole new field emerged, called variously Automated Deduction or Computational Logic. The search for complete and efficient algorithms for pattern-matching, for equation solving (unification), for proof synthesis (resolution), for reasoning with equalities (rewriting), for organizing libraries of mathematical knowledge (modules), gave new direction and applicability to the field of Proof Theory.

The convergence of efforts between Logic and Computer Science was particularly motivated by the discovery that a fundamental structure of proof theory, namely that of natural deduction trees, is isomorphic to a fundamental structure of functional programming languages, namely the typed λ-calculus. This isomorphism — called the Curry-Howard correspondence — associates types to logical propositions, and expressions of the typed λ-calculus to constructive proofs. It suggests that the foundations of software design fall within a very general programme of constructive mathematics.

A flurry of new logics arose from the modelling of computational paradigms — dynamic logics, modal logics, temporal logics, linear logics, logics of computable functions and other logics of partial objects. These, in their turn, provoked a flurry of inference systems, proof search algorithms, and theorem-proving implementations. However, many basic problems are common to all such formalisms, and it was judged a waste of energy to duplicate efforts which

could be treated generically. It must be understood here that the implementation of a robust theorem prover such as HOL (the Cambridge Higher Order Logic system) or the Boyer-Moore theorem prover is a titanic software endeavour, requiring years of development by skilled programmers with a deep understanding of the underlying theoretical issues.

This problem of genericity may be attacked from two different angles. The first one is to choose an extremely general formal system, such as Set Theory, and to axiomatise in it the relevant theory. This is basically the approach chosen in the Automath project, in Martin-Löf's type theory (or such variants as in the systems Nuprl or ALF), and in the Calculus of Constructions (or variants such as those in the systems Coq or LEGO). The second one is to build-in the specific logic not as axioms in a very general language, but rather as rules in a general formalism called a "Logical Framework." Typical examples of this second approach are the Edinburgh Logical Framework (ELF), the Isabelle proof assistant, and the λ-PROLOG language. Both approaches have their advantages. The second permits a finer control of the encoding of the specific logic by the general metalogic, allowing, for example, the proof of conservativity results.

In 1989 the main European research teams working on generic proof systems formed a common consortium in the ESPRIT Basic Research program, entitled "Logical Frameworks." The first general workshop of this action was organised in May 1990 in Sophia-Antipolis, in the south of France. A selection of the papers presented at this workshop were edited as a book, also called Logical Frameworks. The second general workshop was organised in Edinburgh in May 1991; the present book offers a selection of the papers presented there. The title "Logical Environments" reflects a general awareness that, despite minor differences, the community now agrees on the general form of logical frameworks: most proposed such frameworks can be presented uniformly as a Generalised Type System, or GTS for short, *i.e.* as a kind of typed λ-calculus. But there remains much to do as regards the general design of a proof assistant helping the user to manage large proofs interactively. For instance, we need a flexible facility for manipulating definitions. But when definitions should be expanded is largely a research problem. Similarly, we need a general language in which to describe proof developments in a modular fashion. We also need to develop large libraries of mathematical facts, such as basic theorems concerning the usual data structures of programming. Many such issues are tackled in the papers constituting this volume. These have been arranged according to four themes.

The first theme concerns the general problem of representing formal systems in logical frameworks. In the first paper, "Metalogical Frameworks," David A. Basin and Robert L. Constable present their view of the general architecture of a proof assistant, and in particular of the respective positions of the *implementation language* of the prover, the *meta-language* in which proof tactics are written, and the *extraction language* which underlies the constructive nature of proofs. They also discuss the respective role of the *object logic*, imple-

mented by the prover, and of the *programming logic*, used for reasoning about the correctness of the implementation. They describe a coherent architecture in which the metalogic is primary, and is sufficiently powerful to encode all the relevant meta-reasoning, such as proving that an internal simplification routine is correct and always terminates. In the second paper, "Encoding of data types in Pure Construction Calculus: a semantic justification," Stefano Berardi presents a uniform encoding of data types in the pure Construction Calculus, together with a semantic justification. This work gives a general solution to the problem of axiomatising mathematical structures using the Berarducci-Böhm encoding of data types in the polymorphic λ-calculus. S. Berardi shows that first-order program properties do not introduce paradoxes. The semantic justification uses a model for the Calculus of Constructions, built using partial equivalence relations.

In the third paper of this section, "Experience with FS_0 as a Framework Theory," Sean Matthews, Alan Smaill and David Basin present experiments in using FS_0 as a Framework Theory. The formalism FS_0 is due to Solomon Feferman, and provides general means for the description of primitive recursive algorithms on recursively defined data structures. As such, it is strongly related to LISP-like computations on S-expressions. The authors describe an experiment using FS_0 to axiomatise operations on formulas. They give a solution to the thorny problem of manipulating expressions containing bound variables, without resorting to codings (such as de Bruijn's indices), and without the need to quotient all operations by a complicated renaming congruence (α-conversion). The last paper of this section, "Logical Support for Modularisation," is by Răzvan Diaconescu, Joseph Goguen and Petros Stefaneas. This paper studies some properties of logical systems that support the definition, combination, parameterisation and reuse of modules. Rather than using a type theory to represent the *syntactic* structure of logics, the authors make use of the framework of institutions. These are an abstraction of Tarski's *semantic* definition of truth based on a relation of satisfaction between models and sentences.

The second theme concerns basic algorithms of general use in proof assistants. The first paper, "Algorithmic definition of lambda-typed lambda calculus," is by N. G. de Bruijn, the pioneer of Automath, which may justly qualify as the first logical framework. He presents an algorithmic definition of the λ-typed λ-calculus $\Delta\Lambda$, which is the underlying formal structure of Automath terms. A complete type-checking algorithm is presented in detail for the first time in the literature.The correctness and complexity of the various algorithms presented in the paper are discussed. In the next paper, "A Canonical Calculus of Residuals," Yves Bertot gives a general methodology for tracing, for every substructure of a proof term, the justification for its construction. This method is based on a generalisation of the notion of residual, from λ-calculus theory. This mechanism gives a uniform way to explain to the user why a particular proof succeeds or fails. It can thus be considered an essential tool for debugging proof developments.

In the last paper of this section, "Order-Sorted Polymorphism in Isabelle," Tobias Nipkow describes the algorithm actually implemented in the Isabelle proof assistant for type-checking. This combines the kind of genericity present in polymorphic programming languages, such as ML, with the kind of overloading present in systems describing subclasses of objects in a hierarchy, in the tradition of object-oriented formalisms. It uses the theory of order-sorted algebras. The author discusses why this generic mechanism is essential in using Isabelle as a meta-logic.

The third theme is concerned with various logical issues. In the first paper, "An Interpretation of Kleene's Slash in Type Theory," Jan Smith transfers Kleene's slash operation from intuitionistic logic to type theory — particularly Martin-Löf's intensional type theory. This gives conditions for a typeable term containing free variables to have a normal form beginning with a constructor; for closed terms one essentially obtains Tait's computability method. In the second paper, "Inductive Data Types: Well-orderings Revisited," Healfdene Goguen and Zhaohui Luo discuss inductive data types in the framework of intensional type theory. It is shown that a large class of inductive data types may be faithfully represented in the system, completed by a type of well-ordered trees (W-types), provided a number of so-called "filling-up" equality rules are added. The consequences for the meta-theoretic analysis of such systems are discussed.

The last three papers consider how to extract constructive content from classical proofs. Well-known *indirect* methods make use of logical results translating classical proofs to constructive ones. The challenge is to extract the constructive content *directly* from the classical proofs, and obtain practically executable algorithms. In the first paper, "Witness Extraction in Classical Logic through Normalization," Franco Barbanera and Stefano Berardi carry out this programme for Kreisel's result that any Σ^0_1-formula provable in classical logic is also provable in intuitionistic logic. The authors' method is to make use of (their generalisation of) Prawitz' work on strong normalisation for reduction rules for certain systems of classical logic. These rules can be applied to extract the needed witnesses of existential statements. It is particularly noteworthy that the system of rules is non-confluent, unlike the case of intuitionistic logic.

The second paper, "Finding the Answers in Classical Proofs: A Unifying Framework," is by Chetan Murthy. The paper uses continuations to provide a uniform treatment of various methods for extracting computational information from classical proofs. In particular, Murthy treats in this way the work of Barbanera and Berardi, peviously described. In the third paper, "Church-Rosser Property in Classical Free Deduction," Michel Parigot presents free deduction, which is a deduction system for classical logic. This possesses a global cut-elimination procedure. By imposing a choice of "input" for each formula (on the left or the right) cut elimination becomes confluent, and strongly normalising. This can be considered as a first step towards using free deduction as a mechanism for computing with classical proofs.

The fourth and final theme concerns large-scale experiments with proof assistants. In the first paper, "Completing the Rationals and Metric Spaces in LEGO," Claire Jones presents the proof development of the completion of rationals and metric spaces in the LEGO proof assistant. This large proof development, which is motivated by the development of constructive analysis in the Bishop-Stolzenberg style, uncovered certain difficulties with the management of a large library of mathematical definitions and lemmas. The development discusses interesting issues of representation of mathematical notions such as collections of intervals. The second paper, "A Machine Checked Proof that Ackermann's Function is not Primitive Recursive," is by Nora Szasz. The proof was written in Martin-Löf's type theory and carried out using the Göteborg system ALF (Another Logical Framework). As often happens, the process of formalisation necessitates choice of representations of mathematical structures, and decisions on proof structure. Here N. Szasz presents a careful discussion of such matters as the treatment of inductive definitions and abstract specification as a way of modularising proofs.

Contents

Metalogical Frameworks

David A. Basin
Max-Planck-Institut für Informatik, Saarbrücken, Germany

Robert L. Constable
Cornell University, Ithaca, NY, USA

Abstract

In computer science we speak of *implementing* a logic; this is done in a programming language, such as Lisp, called here the *implementation language*. We also reason about the logic, as in understanding how to search for proofs; these arguments are expressed in the *metalanguage* and conducted in the *metalogic* of the *object language* being implemented. We also reason about the implementation itself, say to know it is correct; this is done in a *programming logic*. How do all these logics relate? This paper considers that question and more.

We show that by taking the view that the metalogic is primary, these other parts are related in standard ways. The metalogic should be suitably rich so that the object logic can be presented as an abstract data type, and it must be suitably computational (or constructive) so that an instance of that type is an implementation. The data type abstractly encodes all that is relevant for metareasoning, i.e., not only the term constructing functions but also the principles for reasoning about terms and computing with them.

Our work can also be seen as an approach to the task of finding a generic way to present logics and their implementations, which is for example the goal of the Edinburgh Logical Frameworks (ELF) effort. This approach extends well beyond proof-construction and includes computational metatheory as well.

1 Introduction

1.1 Role of Logical Frameworks and Formalized Metamathematics

At one time logic seemed to be a *finished* subject, and computers, like radio telescopes and linear accelerators, seemed to be the tools of *big science*. Now computers are ubiquitous and their software systems have brought logics to life. We see limitless expansion in both domains with computers destined to

play a part in nearly every aspect of life and the logics of their systems needed to bring order and sense to this activity and provide a basis for realizing ever bolder dreams. This is a new role for logic and with it come new tasks and problems; this paper is about some of them, especially those that fall to computer scientists to address. Let us be more specific about this new role for logic and put these tasks in context, starting with the most familiar.

The first generation of software systems were brought under control with the development of automata theory, formal languages, and programming logics. Modern compilers are built with tools and techniques based on a deep understanding of parsing and compilation, and the methods of rigorous program development make it possible to achieve high levels of reliability for a range of specifiable programs.

Now the field seeks to do more and to carry the techniques forward into more imaginative systems which have formal logics as components. For instance there will be programming languages with type systems that are equivalent to logics, and the type checkers will be little theorem provers (maybe even big ones). There will be formal specification languages and program verifiers both based on a formal logic of some kind. There will be systems that support program synthesis and hardware synthesis. There will be more experimental systems such as programmer's assistants. In all of these systems formal logic is playing its new role. Moreover, there will be many different logics and many different implementations of each. So as in the case of compiler construction, there will be an industry for building and modifying the implemented logics.

Thus far, there are few tools designed specifically to support the activity of implementing logics. So systems are built from scratch using tools intended for other purposes. What is wanting is a framework for designing these logics and supplying the generic tools. This is one of the main goals of research in *logical frameworks.*

Another way to approach the topic of this paper is to consider the activity of generating *formal mathematics*, as pioneered by the project of Automath [11]. This has been taken up with considerable energy in computer science because it is seen as an integral part of some of the new kinds of systems mentioned above and because it is related to concerns in AI and programming. One of the subjects that is most promising to formalize is metamathematics, in part because the results of that work feed back directly to the task of generating formal theorems. In the context of formalizing metamathematics, we are confronted with many of the same questions facing the designer of a logical framework. This paper considers why this is so. Moreover we claim that the vantage point of formalized metamathematics is a good way to look at the task of logical frameworks, and we advocate its primacy, hence the title of the paper.

Scientists actually doing formal mathematics have come to understand the importance of metareasoning from experience. They quickly saw that nearly all informal mathematics is a mixture of object theory and metatheory. The work of Howe [27], comes to our minds in this connection. Also the style of

automating reasoning via tactics involves programming in a *metalanguage*, and if one applies the methodology of programming systems based on the proofs-as-programs principle, such as Nuprl [15] and the Calculus of Constructions (CoC) [17], then many of these tactics should be extracted from metatheorems. A great deal of work has been done in this direction at Cornell [30, 26, 3]. Also even if the proofs-as-programs principle is not underlying the programming style, it has been widely noticed that substantial economies in proof generation are possible if tactics are replaced by the metatheorems that they implement.

The key component in an implemented logic is usually the *inference engine*, and our starting point is the observation that the systems will need ways to modify and progressively enhance it. Our response to the situation is to conjecture that the problems of understanding, specifying and modifying the inference engine are best dealt with in the framework of a *formalized metalogic*. It will be important to reason about the object logic (the one being implemented) in a very expressive metalogic.

1.2 Conceptual Issues

Logicians have been concerned for years with treating their subject more abstractly and generally. The elementary formal systems of Smullyan were an attempt to characterize the deductive machinery of *Principia Mathematica*, and lately Feferman has proposed more usable formalisms [22, 21] such as FS_0. There are abstract semantic characterizations of logic [5, 6] and the theorem of Lindström in abstract model theory characterizing first-order logics. There are other efforts along these lines such as [2].

The work of logicians deals with many of the issues central to the problem of logical frameworks such as what is a logic and what are the different ways to present them. Their concern for the most general setting for major results such as Gödel's theorems or Löb's theorem is directly germane to the task of formalizing metamathematics. But some of the issues faced in defining a logical framework and formalizing mathematics on a machine are new. For example, how should bound variables be represented? How can a sequent based proof economize on storage? What is the best way to realize computational content? What is the essential difference between sequent style proofs and natural deduction style, and does it apply to all logics?

Inevitably discoveries about logical frameworks will contribute to a conceptual understanding of logic. From the vantage point of this paper we see that the efforts to formalize metamathematics will contribute as well. This might not be so obvious *a priori*.

1.3 Results

Beyond advocating the primacy of metalogic this paper contributes some specific technical results. We propose a particular way to specify logics, namely using types that one might call *higher-order abstract data types* or ADT's for

this paper. We show that an implementation of a logic can be seen then as an instance of the abstract data type. There are many interesting connections between these types and other approaches to data abstraction in type theory and programming languages.

We also look at *constructive metatheory in action* and illustrate by example why a framework for automating logic will want access to the results of formalized metamathematics. This practical need is remarkably congruent to the philosophical stance adopted by the first metamathematicians who used finitist or constructivist metatheories. This historical accident means that there is a deep literature of informal implicitly computational metatheory to draw on as well the explicitly computational results of modern computer scientists such as unification, resolution, term rewriting, and the like, and moreover that formalizing these results makes them clearer and more general, thus fulfilling the promise of formalized mathematics. Indeed given the deep literature in proof theory, there is much to be done in this vein to bring it to life and make it available to applied logicians. But also new kinds of metamathematical result are needed in reasoning about implemented logics, and we mention some of them as well.

We show that by taking a rich constructive metalogic as the basis for a logical framework, the context becomes simpler and the relationship between the concerns of logical frameworks and those of the formal metamathematics stands out. Finally, rather than choosing a specific metalogic which is adequate, such as Nuprl or CoC, we talk about the requirements for such a theory.

2 Data Abstraction in Type Theory

In this section, we provide a brief account of data abstraction within type theory. Our notion of data abstraction has similarities to two standard techniques used to define data within type theories; these are the techniques of defining data-types within minimal type theories such as CoC, and the definition of ADTs as Σ-types within predicative type theories.

Within type theories such as the CoC standard data-types (e.g., pairing, lists, etc.) are encoded by inductive definitions. These data-types are specified by:

- naming a set (type) that members will inhabit;
- giving function constants (and their types) for constructing members of the set;
- providing rules for reasoning about type members and functions defined over them.

These declarations can be seen as capturing the rules for type formation, introduction and elimination (right/left rules), and computation. In theorem

$$
\begin{array}{l}
LIST \hat{=} A{:}Type \rightarrow \\
\quad List{:}Type \\
\quad \# \ Nil{:}List \\
\quad \# \ Cons : A \rightarrow List \rightarrow List \\
\quad \# \ Lind{:}\ P{:}(List \rightarrow Type) \rightarrow l{:}List \rightarrow P(Nil) \\
\qquad \rightarrow (h{:}A \rightarrow t{:}List \rightarrow P(t) \rightarrow P(Cons(h,t))) \rightarrow P(l) \\
\quad \# \ P{:}(List \rightarrow Type) \rightarrow l{:}List \rightarrow g{:}P(Nil) \rightarrow i{:}(h{:}A \rightarrow t{:}List \rightarrow P(t) \rightarrow \\
\qquad\quad P(Cons(h,t))) \rightarrow Lind(P, Nil, g, i) = g \\
\qquad\quad \wedge \forall h{:}A.\forall t{:}List.\, Lind(P, Cons(h,t), g, i) = i(h, t, Lind(P, t, g, i))
\end{array}
$$

$$
\begin{array}{l}
MSET \hat{=} A{:}Type \rightarrow \\
\quad MSet{:}Type \\
\quad \# \ Empty{:}MSet \\
\quad \# \ Add : A \rightarrow MSet \rightarrow MSet \\
\quad \# \ \forall a,b{:}A.\forall s{:}MSet.\, Add(a, Add(b,s)) = Add(b, Add(a,s)) \\
\quad \# \ Mind{:}\ P{:}(MSet \rightarrow Type) \rightarrow l{:}MSet \rightarrow P(Empty) \\
\qquad \rightarrow \{i{:}(a{:}A \rightarrow s{:}MSet \rightarrow P(s) \rightarrow P(Add(a,s))) \mid \\
\qquad\quad \forall a,b{:}A.\forall s{:}MSet.\forall p{:}P(s).\, i(a, Add(b,s), i(b,s,p)) = i(b, Add(a,s), i(a,s,p))\} \\
\qquad \rightarrow P(l) \\
\quad \# \ P{:}(MSet \rightarrow Type) \rightarrow l{:}MSet \rightarrow g{:}P(Empty) \\
\qquad \rightarrow i{:}\{i{:}(a{:}A \rightarrow s{:}MSet \rightarrow P(s) \rightarrow P(Add(a,s))) \mid \\
\qquad\quad \forall a,b{:}A.\forall s{:}MSet.\forall p{:}P(s).\, i(a, Add(b,s), i(b,s,p)) = i(b, Add(a,s), i(a,s,p))\} \\
\qquad\quad \rightarrow Mind(P, Empty, g, i) = g \\
\qquad\quad \wedge \forall a{:}A.\forall s{:}MSet.\, Mind(P, Add(a,s), g, i) = i(a, s, Mind(P, s, g, i))
\end{array}
$$

Figure 1: List and Multi-set ADT

provers for CoC such as *LEGO* [12] these declarations are made within a *context*, and one reasons within the scope of these declarations.

When a type theory is enriched with Σ-types, a second approach is possible whereby the declarations of the ADT are packaged together with a (iterated) Σ-type; theorems are then proved within the context of these types. This has the conceptual advantage of unifying the data-type definition into a single type and the practical advantage of allowing parameterization of one ADT by another. It can also be seen as a natural generalization of signatures in ML, Extended ML [44], and other languages implementing data-abstraction using dependent types and is also related to the ideas of Bauer and his group at Munich on nonfree algebraic types [9]. This is the approach we shall take. One complication is that such modularization requires predicative quantification and hence a type hierarchy. For simplicity, we shall use the first universe of types (which we call *Type*) as the type of ADT parameters and set carriers.

Figure 1 contains two examples of ADTs: parameterized lists and finite multi-sets (sometimes called "bags"). Members of the LIST type are functions: when applied to a parameter type A, they return a tuple in which the first projection, the *carrier* of the ADT, is the type of lists over A whose members are built from the functions inhabiting the *Nil* and *Cons* projections. The multi-set ADT is similar except for the defined carrier equality and the