ESPRIT Basic Research Action 6453

TYPES: Types for Proofs and Programs

Hugo Herbelin
INRIA, Paris

Formal description

The TYPES project officially started September 1st 1992 and has a planned duration of 3 years. It is the continuation of the LF project. The project LF ("Logical Frameworks") was concerned with the study of formalisms for the representation of proofs and the development of prototypes to support them. From LF to TYPES, the number of partners has increased. The current partners involved in the project TYPES are the following :

Institution	*Representative*	*Country*
INRIA Rocquencourt	Gérard Huet	France
ENS Lyon	Christine Paulin-Mohring	France
University of Cambridge	Larry Paulson	United Kingdom
University of Edinburgh	Rod Burstall	United Kingdom
Oxford University	Lincoln Wallen	United Kingdom
Manchester University	Peter Aczel	United Kingdom
Philips Laboratory Eindhoven	Leen Helmink	The Netherlands
Nijmegen University	Henk Barendregt	The Netherlands
INRIA Sophia Antipolis	Gilles Kahn	France
University of Torino	Mario Coppo	Italy
University of Udine	Furio Honsell	Italy
CNRS/Université Paris 7	Michel Parigot	France
Technical University Munich	Tobias Nipkow	Germany
Chalmers University (in Göteborg)	Bengt Nordström	Sweden

For any information, please contact:

Gérard Huet, Phone +33/1-39-63-54-60, Gerard.Huet@inria.fr

Keywords Type theory, Proof assistant, Lambda Calculus, Unification, Software Verification

Introduction

The aim of the TYPES action is to develop proof assistant systems with applications both to the formalisation of large pieces of mathematics and to proofs of correctness for software and hardware systems. This latter application is of increasing interest for the needs of the software and hardware industry.

However, the project is not only concerned with implementation but also with longer-term research, namely investigations in proof theory and logical formalisms applied to computer science,

software engineering (including rewriting, unification, design of user interfaces, design of specification languages) and experiments in industrial applications (especially in cooperation with our partner Philips). Trivially, each one of these activity fields reveals spontaneously opportunities for improvements and new questions in the other activity fields.

In a first section, objectives of the project are detailed. The second section is concerned with the description of the proof assistants developed by the project. There are four general proof assistant systems and one more specialized system and they are essentially built on the proof as program paradigm.

In the following sections, we explain successively the state-of-the-art of foundational aspects, of the implementations and, in the last section, we describe summarily the state-of-the-formalisation.

Training and dissemination

Our group is the originator of an annual summer school, alternatively on Proof Theory and Foundations of Programming and on Automated Deduction, started in 1993 at Chambéry, France. Our group organized also, in 1993, a summer school on Proof Development Systems, at Baastad, Sweden, and a course on theorem proving using Isabelle. In September 1994, it organized a school on Typed Lambda Calculi and Programming Languages at Udine, Italy. The project supported also the ninth Annual IEEE symposium on Logic in Computer Science, hold from July 3-7, 1994, at Paris, France.

An annual workshop gathers the people of our action together. In 1993, the workshop held at Nijmegen and the refereed proceedings are published as a volume of the Lecture Notes in Comuter Science, Springer Verlag.

Several small workshops are regularly held on special topics. These small workshops gave the opportunity to collaborate with people from the CLICS action (Categorical Logic In Computer Science) and from the CONFER action (Concurrency and Functions: Evaluation and Reduction).

1 Objectives

Applications to software engineering

The problem of correct software development is studied from different points of view. Several examples are investigated, but we attempt also to raise some methodologies for program development that could be easily transferred to other computer scientists. For this purpose, we stress on the design of a general-purpose specification language, usable both by the mathematicians and the computer scientists.

Implementations

Most of our systems are based on an interactive development of proofs using tactics. It is essential to provide powerful tactics. Higher-order unification or matching procedures are such tools upon which we can write tactics for automatic proof search. Tactics based on rewriting systems are also studied and integrated to our tools.

Developing bigger examples will raise the problem of handling large databases of definitions and lemmas.

Formalisms

Industrial exploitation

We have an industrial partner, Philips, who help us by providing industrially relevant case studies. The interactions with people outside of our group allows us to improve our tools too.

In the domain of software and hardware correctness, our main competitor are the HOL project and the Boyer-Moore work, both based on an earlier generation of technical foundation.

There is already considerable interest in industry in proof for hardware design, and in proof for concurrent software. Hardware problems are more tractable, being less sophisticated in their inductive structure, and concurrent programs are almost impossible to debug satisfactorily. It is the latter on which we place most emphasis in this project, but the tools which we are devising are unbiased in this respect. They provide a complementary alternative to bisimulation based approaches such as the Concurrency Workbench, being able to express a wide variety of formalisms in a single framework, including temporal logic ones. Most industrial research groups are active in concurrency.

Another area of considerable industrial concern is specification. Specification languages are an active field of research in industry. Type theory provides a common language in which one can express specifications and programs and proofs of relative correctness. When an inhomogeneous system is used with different representations of these the systems become too big for most research groups to handle, so the compactness of the type theory approach should make it more acceptable to industrial groups once they have come to understand the rather sophisticated ideas underlying it.

Most of the participants have experience of interacting with industry and of communicating their results, through spin off companies (INRIA filiales) or through research initiatives with emphasis on industrial participation (Edinburgh with Alvey programme). INRIA Rocquencourt have an active collaboration with Dassault on applications of proof systems, and Nijmegen have collaborated in a similar area with Philips. In addition many researchers have acted as industrial consultants (for example Barendregt with Philips, Plotkin with DEC SRC and Burstall with ICL and DEC SRC).

2 Description of the proof assistant systems

The proof assistants developed by the group are based on type theory and on the "proofs-as-programs/propositions-as-types paradigm". There is one exception, the system Isabelle is a logical framework. logics.

Type theory and the Curry-Howard isomorphism

A type theory, in the logical sense, is a formal system with a typed (and thus stratified) notion of functions and sets. This is in contrast with, for instance, the set theory in which all sets and functions are in the same boat. In type theory, a function waits for an argument of a certain type and cannot be applied to arguments of a wrong type. Similarly for the sets.

The "proofs-as-programs/propositions-as-types paradigm" is a consequence of the Curry-Howard isomorphism, in virtue of which, type theories are nicely expressed as typed λ-calculus.

The Curry-Howard isomorphism starts with the remark that the simply typed λ-calculus corresponds to the minimal propositional logic. More precisely, the typed λ-calculus is *isomorphic* to

the formalisation of minimal propositional logic in natural deduction. This means a very strong link between proofs and programs.

The isomorphism can be extended to stronger type systems for λ-calculus to which corresponds more powerful logics. AF_2, the system upon which is based the system Propre, is such a formal system (precisely the second order predicate calculus) defined as a typed λ-calculus. So is the Calculus of Constructions of which the logical strength is the one of the higher order predicate calculus, but which allows in addition the definition of elaborated data types and functions over these data types.

The Calculus of Constructions

Two of our general systems, namely Coq and LEGO, implement the Calculus of Constructions enriched with powerful recursion schemes. This extension of the Calculus of Constructions has been proved consistent. This means that we cannot prove false statements in it Up to the confidence in the consistency of the meta-system used to formalize the argument of consistency. This meta-system, in harmony with the second incompleteness theorem from Gödel, is stronger than the system of which the consistency is proved.

ALF is a framework for handling inductive concepts. It is also based on the Calculus of Constructions but provides a unrestricted mechanism to define inductive types and to do proofs by induction. In particular, it is possible to defined inductive notions leading to inconsistency. Then, the user has to check himself the consistency of the notions he defines.

The calculus of constructions is logically powerful but not only that. It allows easy and powerful definitions of sets, functions and predicates as well.

- Structures for computer science such as lists, trees, regular expressions or finite state machines can be easily defined by means of a general scheme of definition by induction.
- "Annotated" structures, such as lists of a given length or natural numbers which are even can also be considered.
- Mathematical structures such as natural numbers, integers, reals or abstract sets (in the sense of set theory) can also be easily defined inductively or using higher order abstraction schemes.
- Functions can be defined by recursion or pattern-matching over inductive types and by abstraction as in ML-like languages.
- Predicates can be defined inductively or by abstraction on first-order properties. In particular, it is direct to define specifications in a Prolog-like way.

Proofs as programs

Since the Curry-Howard correspondence tells that a proof is a λ-term, a proof may be executed. But in fact a proof is more than a program: it is a program together with a purely logical proof that the program meets the expected specification. Thus, we have to define method to forget the purely logical, computationally irrelevant part. For instance, a proof of $\exists y.Q(y)$ gives explicitly an object n such that $Q(n)$. A proof of $\forall x.\exists y.Q(x,y)$ gives a functional program f such that $\forall x.Q(x,f(x))$. This interpretation makes our systems directly usable as frameworks for the development of correct programs. However, at the present, it is difficult to control the complexity of the extracted program. Thus, research is done in order to optimize the extracted programs. Another approach is to give a program and to try to reconstruct the correctness proof of which it is extracted.

The Curry-Howard isomorphism was first established for the intuitionistic logic, a logic which does not allow the reasoning by contradiction (= the axiom $\neg\neg A \Rightarrow A$), in contrast with classical logic, the logic which allows this kind of reasoning. Therefore, at the time being, it is not possible to get a program from a proof in classical logic. It is not really damaging because reasoning by contradiction may always be circumvented in specification proofs. However, it is more common to reason in classical logic. Since a few years ago, we know now that the reasoning by contradiction has also a computational counterpart, but it is not yet a completely clear one. It is work in progress and part of our research to clarify the relation between classical logic and computation in order to extend our systems to classical logic.

Our systems allow a modular construction of library of theories expressing, for instance, basic results of arithmetics, basic properties of groups or fields or metric spaces, basic properties of lists or trees, or automata. We are now working at the possibility of parameterized theories.

The narrow link between proofs and programs is the main feature of our systems which distinguishes them from the Boyer-Moore and HOL systems, built on an older technology. This allows to execute proofs as if they were programs or/and to extract automatically certified programs from the proofs of their correctness.

The standard way to build proofs in our systems is to do interactive step by step goal-directed refining. *Tactics* are applied to statements in order to split them in more simple one. This feature is shared with the HOL system, but this latter has more powerful tactics than our systems. In particular, like the Boyer-Moore system (but this one does not allow refining), HOL provides with sophisticated tools for proof search automation. However, the proof engineering technology of these systems (e.g. rewriting, induction, automatic proof search) is progressively integrated to some of ours systems.

In our system, a special attention is also paid to user interfaces.

As for Propre, it is a more specific theorem prover built on the system AF_2, a formalisation of the second order predicate calculus.

Here is a consequence of this kind of formalism : proof checking is reduced to type checking which is decidable in our systems. A proof can be deep or complex, but once it is found, checking its correctness is easy.

ALF implements Martin-Löf's monomorphic type theory and is developed at Göteborg. Inductive definitions and proofs by induction are the natural ways to use ALF. The window interface plays a major role in the construction of proofs, allowing proof editing as with "scratch paper and pencil". Some mechanisms of very recent conception allows to automatically eliminate the contradictory cases in a proof.f

ALF and its documentation are available by anonymous ftp at `cs.chalmers.se` in the directory `pub/provers/alf`.

Coq is an implementation of the Calculus of Inductive Constructions and is developed at INRIA Rocquencourt and ENS Lyon. It is a kind of higher order predicate calculus with inductive types and powerful recursion schemes. It permits the extraction of ML programs iterating on inductively defined types, which correspond naturally to the usual algorithms. It permits also the reverse approach, proof reconstruction from a program and its specification.

There are several user interfaces using the X window system, one of them is built upon the system Centaur. In this approach, the front-end and engine are two separate entities communicating by message passing. The previous implementation of the Calculus of Constructions was distributed to more than 10 sites.

Coq and its documentation are available by anonymous ftp at `ftp.inria.fr` in the directory `INRIA/coq/V5.8.3`. Last year, a user's club, the Coq Club, has been started as an electronic mail forum `coq-club@margaux.inria.fr`.

LEGO is an implementation, developed at Edinburgh, of the Extended Calculus of Constructions (ECC) and the Edinburgh Logical Framework (LF). ECC does not include primitively inductive types, but a facility is provided in order to define convenient and efficient inductive types as in Martin-Löf's monomorphic type theory.

LEGO and its documentation are available by anonymous ftp at `ftp.dcs.ed.ac.uk` in the directory `pub/lego` and on the World Wide Web (the relevant URL is `http://www.dcs.ed.ac.uk:80/packages/lego/`)

Logical Framework

Proofs are done in formal systems. The latter may be described by a language of primitive objects (such as connectives, atomic formulas, variables, inference rules, ...) together with a set of rules allowing to build complex objects from the primitive ones (rules to build well-formed formulas, to form correct proofs, ...). A "logical framework" is such a framework in which it is possible to define primitive objects and construction rules, and thus to specify particular formal systems.

One of the general systems developed by our group, namely Isabelle, implements a logical framework. This means that this system, through an encoding, support formal systems of various kinds. The standard library of logics supported by Isabelle includes the predicate calculus, Zermelo-Fraenkel set theory, but also some type theories (Church's Higher Order Logic and Martin-Löf's Constructive Type Theory). Then, using Isabelle, one can reason freely in any one of these formal systems.

The fact that Isabelle supports – indirectly – type theories, implies that we can do with Isabelle as much things as in the systems directly supporting a type theory.

Isabelle is the fourth proof tool involved in the project. Isabelle uses intuitionistic higher-order logic as a meta-level framework for the representation of various logics. Terms and propositions of the object-logic are represented by simply typed λ-terms; the deduction rules are specified using meta-level implication and quantification. Isabelle supports existential variables and provides powerful tools for automated reasoning: higher-order unification, rewriting, and first-order theorem proving. Isabelle also supports a system for correct-program development, based on program extraction. This system uses classical reasoning for proving assertions, and yields a realistic functional program containing no proof objects. The system is based on well-founded recursion; programs are not restricted to primitive recursion.

Three of the most important object-logics currently supported by Isabelle are Zermelo-Fraenkel set theory, Church's higher-order logic (also implemented by Gordon's HOL system), and a version of Martin-Löf's Constructive Type Theory. This latter shows that Isabelle may be used also to do proofs in type theory as the other systems.

Isabelle and its documentation are available by anonymous ftp at `ftp.cl.cam.ac.uk` in the directory `ml`.

Propre is an experimental programming language under design at Paris 7. It is based on the "proofs as programs" paradigm for the second order type theory AF2 introduced by Leivant and Krivine and its extension to recursive types by Parigot. The idea is to specify a program by a set of equations, and prove the termination of this program in AF2. The theory tells how to extract a pure λ-term from the proof that it satisfies the original specification.

The implementation provides a proof strategy, based on recursion, for the automatic derivation of programs from equational specifications. and an abstract machine with special features for the execution of these programs.

3 Foundational aspects

Development of type theory A new type theory, UTT has been developed at Edimburgh. UTT combines the *full* Martin-Löf's intensional type theory and the calculus of constructions; compared with ECC, the system upon which LEGO is based, it still has a distinguished internal higher-order logic and good mechanism for modularisation and polymorphism, but supports new (and better) styles of development of proofs and programs with inductive data types. Many mathematical structures can now be directly described as *types* rather than 'sets' considered at the logical level (a notion unclear how to treat in type theory); this improves in great deal the adequacy of formalisation and the efficiency of proof development. Programs are now (in a general form of) primitive recursive functions rather than the (problematic) impredicative codings and both programming and proving program correctness become easier and more effective.

Design of logical frameworks. Work has been done on logical frameworks to allow a better representation of logics. R. Constable and D. Basin introduced the notion of Metalogical Frameworks. This method of definition explains especially well the relationship between a logic and its implementation. P. Gardner has developed a refinement of the Edinburgh Logical Framework that allows uniform understanding of the correctness of representation of logics. A. Simpson and G. Plotkin worked on an extension of the Edinburgh Logical Framework that permits a natural encoding of modal logics.

New deduction systems for classical logic. The possibility of extending the Curry-Howard isomorphism to classical logic is a major discovery of the decade. It has repercussions both from the point of view of mathematical logic and from the point of view of computer science.

We get classical logic from intuitionistic logic by adding the axiom $\neg\neg A \Rightarrow A$ for any proposition A. The computational counterpart of this axiom has been very recently understood. It corresponds to a control operator such as the operator C defined by M. Felleisen in 1986. This kind of operator, with appropriated reduction rules, allows to add imperative features (for instance *goto*, exception handling, ...) to purely functional languages.

A good candidate for implementation is probably the $\lambda\mu$-calculus of M. Parigot. It is a variant of the λ-calculus $+ C$ but more elegant reduction rules. Now, if we consider classical logic without implication, it becomes possible to identify the formula $\neg\neg A \Rightarrow A$ with A. This has led to the λ_{sym}-calculus, a nice restriction of λ-calculus defined with non-deterministic computational rules.

Another approach is to interpret proofs as winning strategies for a two-players game of which the rules depend on the disputed formula. This approach subtends a dynamic approach of computation, strongly related to weak-head-style reduction. Especially, in this framework, we can define computation algorithms radically different from the usual computation algorithms based on λ-calculus, but with a good behaviour on some examples of classical proof.

Relation between frameworks and standard logics. Berardi and Geuvers invented the notion *Pure Type Systems* (PTS, but also called Generalized Type Systems). This notion allows a uniform description of at once the simply-typed λ-calculus, the LF system of Edinburgh and the Calculus of Constructions. Stefano Berardi also showed that higher-order logic can not be conservatively embedded in the Calculus of Constructions. However, he obtained a partial conservativity result for CC with respect to higher-order arithmetic, showing that provability in CC coincides with provability in higher-order arithmetic for all "interesting mathematical statements" (in particular first-order statements). On the other hand, Z. Luo, working on the problem of adequacy of formalization of mathematical problems in impredicative type theory , showed that higher-order logic can be conservatively embedded in ECC. This result supports his point of view that arbitrary sets should be formalized predicatively. Further analysis of these results from the point of view of PTSs allowed Geuvers and Pollack to point out that Luo's conservativity result is an extension of C. Paulin's conservativity of CC over $F\omega$.

Foundations of inductive types. A semantical justification of inductive types (see below) allows to know in which manner they can be considered without leading to paradoxes. For instance, Alf does not check if the inductive definitions it allows may lead or not to contradictions. We have to investigate the cases where an inductive definition may be used safely or not in ALF. Also, there are open questions about coinductive types, these one being of interest in order to handle notions like streams.

Recently, it has been proved by B. Werner that the Calculus of Inductive Constructions enjoys strong normalization.

Investigations in λ-calculus. A notable result is that Ralph Loader disproved the long-standing Plotkin-Statman conjecture concerning lambda-definability. On another side, works has been done from elementary ideas from game theory. This kind of ideas is fruitful to provide an intensional or dynamic semantics for both proofs and computation. One application is to use a notion of finite fair games to provide a fully complete semantics for proofs of multiplicative linear logic without the MIX-rule, thus achieving a sharp characterization of proofs. Game semantics have also been used for modelling computations. A notion of innocent strategies has been developed which leads to a (intensional) solution of the full abstraction problem for PCF, a long-standing open problem in semantics of programming languages.

Verification and Synthesis of programs in Isabelle. Martin Coen investigated formal program development as a theorem proving task. Results obtained from the earlier work using ZF set theory have been abstracted to produce a type theory, based on classical first-order logic and incorporating dependent types. Well-founded recursion is expressible and all typable terms represent terminating programs. Tactics have been developed for type-checking, yielding a set of proof obligations, and for rewriting. In particular, rewriting terms using weak head-normal form reduction provides an operational semantics, incorporating well-founded recursion, with which to execute synthesized programs.

4 Implementation of frameworks and pragmatics

Enhancement of the proof assistants useability

Extension of systems to inductive types. ML-like inductive types have been implemented in almost all of the proof assistants. They allow very convenient and powerful definitions. It is direct to define ML-like data-types and functions defined by recursion on these data-types. The definitions of mathematical properties by induction appear clearer and more natural than definitions in a language such as the usual predicate calculus.

A related problem is the problem of "inductive types inversion": how to know which clause of the definition of an inductive predicate has given such or such instance of the predicate. Several solution have been proposed. On one side, ALF provides the user with an automatic analysis (based on unification) of how to invert an inductively defined predicate. On the other side, Coq supplies a set of tactics solving the problem, while Isabelle provides tools for generating instances of rule inversion.

Possibility of defining functions by equations. The heuristics of Propre have been integrated to Coq. This allows to define a function by a set of rewriting rules and the system automatically turns it into a primitive recursive-like definition.

User interfaces. Window interfaces for Coq and Isabelle have been written using the Centaur environment. This kind of work suggests conditions that a system has to satisfy in order to support a window interface. The Sophia site is also working on the translation of formal proofs into proofs written as in mathematics books.

Design of a proof methodology. Works are done in order to develop a specification language, sufficiently close of the logic to be understood by a machine but sufficiently intuitive to be understood by both a mathematician and a computer scientist. Works are done also to develop window interfaces with the same aim. An ideal point of view is the "proof by pointing" paradigm : the successive steps of a proof are directed by mouse clicks.

Modularity It is necessary to have the possibility of doing modular proofs, as soon as there are a great number of development. There is work on modularity of proof in the Extended Calculus of Constructions, using the notion of Σ-type internal to the logic. Examples in LEGO have been developed. But there is also implementation of tools for using modules.

Enhancement of the proof search automation

In addition to implementation of standard automatic proof search techniques (for instance, implementation of a tautology checker and refiner in Coq), basic research is done on unification.

Unification. Unification is an important tool for the automation of proof assistants. Following ideas by D. Miller, restrictions of unification to efficiently solvable problems have been investigated. Miller isolated a weak class of unification problems for λ-terms that are efficiently decidable, yet powerful enough to declaratively express the full (undecidable) higher-order unification problem. This idea has been extended to the dependent types theories, including the Calculus of Constructions. Nipkow has given a short formalization of Miller's ideas by a system of transformation rules and developed this into an ML-implementation.

Gilles Amiot is studying unification and its relation with proof search in linear logic. He proved the undecidability of the second order predicate unification problem. He has deduced from this result the undecidability of second order linear logic, without exponentials.

G. Dowek formulated a proof synthesis method which is complete for all type systems inside Barendregt's cube, as well as the Calculus of Constructions with universes. The originality of this method is that it does not distinguish between the levels of unification and resolution, but instead builds on the correspondence between proofs and objects to integrate these two processes. He also obtained a result of undecidability of pattern matching at the 3rd order in the calculus with dependent types $\lambda\Pi$.

The decidability of matching for simply typed lambda-calculus is still open. However the third order case (Dowek) and the fourth order case (Padovani) have been proved to be decidable. Third order matching is also decidable in lambda-calculus with type constructors (Springintveld).

Existential variables. Existential variables means "open proofs", i.e. proofs containing unproved parts or unspecified witnesses, waiting to be proved or instantiated. Providing existential variables for the representation of incomplete proofs or unspecified variables is a promising way to integrate proof verification and proof search. As a consequence, some of ours systems can be used in a Prolog-like way. For instance, the proof of a statement $\forall x.\exists y.Q(x,y)$ can be used in a reverse way: you give y and the system gives, if possible, the x which satisfy(ies) $Q(x,y)$. Suppose now that $Q(x,y)$ means "y is the compiled code of the program x". Then you give the compiled code and the system tells you from which program x it comes.

Existential variables facilitate the automatic inference of implicit data in statements. For instance, in a statement $x = 3 + y$, it becomes unnecessary to specify that x and $3 + y$ are natural numbers.

Development of certifed programs

Program extraction. As explained before, it is possible to extract a program from a proof of its specification. But this technology raises several questions: how to be sure that a proof reflects an intended algorithm, how to reflect at the proof level some efficient programming structures. Problably proof reconstruction is a more promising approach.

Proof reconstruction. The structure of the proof and the one of the extracted program are strongly related. C. Parent studied the possibility to automatically build a proof using the expected program as an input for a sophisticated tactic. Given a specification and a program which is supposed to meet the specification, the tactic generates logical lemmas which have to be checked in order to insure the correctness of the algorithm. A prototype of this tactic is now available in Coq.

Proving properties of Program The implementation of Logics in LF and Coq has inspired the development of proof systems in Natural Deduction style for proving properties of programs. M.Miculan at the subsite of Udine has carried out the representation in ND-style of Structural Operational Semantics (initiated by Kahn and Despeyroux at Sophia, and later developed by Burstall and Honsell) and has developed and adequately formalized ND-style proof systems for Hoare Logic and Dynamic Logic. This work addresses the issue of finitary vs. infinitary rules/proof systems, and their representation within Logical Frameworks.

5 Formalisation

Numerous examples are formalized in the proof assistants. These examples include several theorems of our research field but also standard theorems of arithmetic, algebra and calculus. Results in computer science are also formalized. The largest concrete examples are concerned with the verification of communication protocols. The correctness proofs for several simple algorithms have also been formalised in order to test the feasibility of the program extraction paradigm.

A selection from the proof development libraries of ALF, Coq, Isabelle and LEGO is available by anonymous ftp at `cs.chalmers.se`, under the name `pub/provers/TYPES/typeslib.dvi`.

Case studies

Properties of λ-calculi Both the confluence of the λ-calculus and the confluence of the Hardin-Lévy $\lambda\sigma_{\Uparrow}$-calculus have been proved in the system Coq. Numerous proofs of normalisation for various typed λ-calculi have also been done in Alf.

Recursive functions N. Szasz has implemented a proof of Ackermann's result on the growth of primitive recursive functions in ALF. This is an extensive example, including more than 200 theorems. Using Isabelle, Paulson has done a similar proof, which is available electronically.

Other examples Various simple examples have been now developed : subject reduction for lazy reduction in PCF, Girard's paradox for system U, Gilbreath card trick, Eratosthene's sieve (using streams encoded as coinductive types), intuitionistic versions of Ramsey's theorem, Higman's lemma, correctness of a tiny expression compiler into polish form, the Church-Rosser Theorem for combinators.

Programs extracted from proofs in Coq Various programs have been developed from a proof of their specification: Warshall's transitive closure algorithm, Quicksort, Heapsort, insertion in AVL trees, Euclid's division, text formatting, Dijkstra closest path algorithm, unification algorithm for first-order terms, search of embeddings of sequences over a two-letter alphabet. All these programs are executable in ML language.

Formalisation of mathematics

Algebra and arithmetic

A formalisation of a few notions from abstract algebra (Setoids, Categories, Monoids, Groups, Rings, etc,...) has been done in Coq.

The main arithmetical results proved are the theorem of Bezout and the Chinese remaindering theorem. A formalisation of the proof of unicity and existence of the decomposition into prime factors for integers has been done in Alf.

Calculus

A large piece of constructive mathematics has been implemented in LEGO. This includes all the usual axioms for the basic functions and relations about the rationals and the construction of reals from the rationals. A proof of the Lagrange theorem has also been done.

Computer science

Automata theory A library of notions of formal language theory (finite automata and rational languages, context-free grammars and push-down automata) has been done by the site of Lyon.

Basic routines and properties on data types On the basis of the implementation of inductive types, a standard library of basic inductive types, the associated functions and their properties and proofs, containing about 400 theorems and many datatype definitions, has been developed in LEGO.

Semantics An axiomatization of Domain Theory has been done in Coq by the site of Sophia. This has allowed to check a few initial pages of a paper by Kahn and Plotkin. F. Regensburger at TU Munich has produced a complete semantic embedding of LCF in Isabelle/HOL, including a construction of the natural numbers, lists and streams. This formalization has enabled him to derive admissibility statements, and structural induction and co-induction schemata not derivable in Classic LCF.

Compiler correctness A correctness proof for a compiler from mini-ML towards the Categorical Abstract Machine (CAM) has been done in Coq. Questions have appeared concerning the formalisation of pointers. This has led to a theoretical study of inductive types quotiented by equations. At TU Munich two equivalence proofs between a denotational and an operational semantics were carried out: one in Isabelle/LCF for a small functional language, and one in Isabelle/ZF for a while-language.

Formalisation of PTSs The theory of Pure Type Systems (PTSs) together with their basic meta-theoretic properties has been formalized in LEGO. In short, the proofs are concerned with the basic syntactic lemmas on type systems about renaming, substitution, reductions, which are a background for the complete formalization of most syntactic proofs. A proof of the strong normalization theorem for Girard's system F has also been formalised in LEGO.

Z and VDM in LEGO. The library of the Z specification language has been reproduced in LEGO. This has been applied to a small Z specification from Hayes' book on case studies. A similar exercise has been done for VDM.

Proofs of software correctness

Correctness of Communication Protocols The Alternating Bit Protocol has been checked in the system Coq by the sites of Nijmegen and Eindhoven. A certainly more advanced protocol is the I/O-automata, a model for distributed processes, formalized in Isabelle/HOL by Nipkow and Slind. Based on this formalization they have proved the correctness of a protocol which achieves reliable transmission on channels which may lose and reorder (but not duplicate) messages, using only a single bit header. This is a fairly subtle protocol with a non-trivial proof. Another work on the verification of an industrial telecommunication protocol from Philips is planned for 1994.

About this note

G. Huet suggested to write this note. Most of the information is extracted from the technical annex of the initial proposal, from the periodic progress report of September 1993 and from recent contributions of the site leaders. The whole has been structured by H. Herbelin. Numerous improvements have also been made by J. H. McKinna.