# HISTORY OF INTERACTIVE THEOREM PROVING

John Harrison, Josef Urban and Freek Wiedijk

## 1 INTRODUCTION

By interactive theorem proving, we mean some arrangement where the machine and a human user work *together* interactively to produce a formal proof. There is a wide spectrum of possibilities. At one extreme, the computer may act merely as a checker on a detailed formal proof produced by a human; at the other the prover may be highly automated and powerful, while nevertheless being subject to some degree of human guidance. In view of the practical limitations of pure automation, it seems today that, whether one likes it or not, interactive proof is likely to be the only way to formalize most non-trivial theorems in mathematics or computer system correctness.

Almost all the earliest work on computer-assisted proof in the 1950s [Davis, 1957; Gilmore, 1960; Davis and Putnam, 1960; Wang, 1960; Prawitz *et al.*, 1960] and 1960s [Robinson, 1965; Maslov, 1964; Loveland, 1968] was devoted to truly *automated* theorem proving, in the sense that the machine was supposed to prove assertions fully automatically. It is true that there was still a considerable diversity of methods, with some researchers pursuing AI-style approaches [Newell and Simon, 1956; Gelerntner, 1959; Bledsoe, 1984] rather than the dominant theme of automated proof search, and that the proof search programs were often highly tunable by setting a complicated array of parameters. As described by Dick [2011], the designers of automated systems would often study the details of runs and tune the systems accordingly, leading to a continuous process of improvement and understanding that could in a very general sense be considered interactive. Nevertheless, this is not quite what we understand by interactive theorem proving today.

Serious interest in a more interactive arrangement where the human actively guides the proof started somewhat later. On the face of it, this is surprising, as full automation seems a much more difficult problem than supporting human-guided proof. But in an age when excitement about the potential of artificial intelligence was widespread, mere proof-checking might have seemed dull. In any case it's not so clear that it is really so much easier as a research agenda, especially in the

context of the technology of the time. In order to guide a machine proof, there needs to be a language for the user to communicate that proof to the machine, and designing an effective and convenient language is non-trivial, still a topic of active research to this day. Moreover, early computers were typically batch-oriented, often with very limited facilities for interaction. In the worst case one might submit a job to be executed overnight on a mainframe, only to find the next day that it failed because of a trivial syntactic error.

The increasing availability of interactive time-sharing computer operating systems in the 1960s, and later the rise of minicomputers and personal workstations was surely a valuable enabler for the development of interactive theorem proving. However, we use the phrase *interactive theorem proving* to distinguish it from purely automated theorem proving, without supposing any particular style of human-computer interaction. Indeed the influential proof-checking system Mizar, described later, maintains to this day a batch-oriented style where proof scripts are checked in their entirety per run. In any case, perhaps the most powerful driver of interactive theorem proving was not so much technology, but simply the recognition that after a flurry of activity in automated proving, with waves of new ideas like unification that greatly increased their power, the capabilities of purely automated systems were beginning to plateau. Indeed, at least one pioneer clearly had automated proving in mind only as a way of filling in the details of a human-provided proof outline, not as a way of proving substantial theorems unaided [Wang, 1960]:

> The original aim of the writer was to take mathematical textbooks such as Landau on the number system, Hardy-Wright on number theory, Hardy on the calculus, Veblen-Young on projective geometry, the volumes by Bourbaki, as outlines and make the machine formalize all the proofs (fill in the gaps).

and the idea of proof *checking* was also emphasized by McCarthy [1961]:

> Checking mathematical proofs is potentially one of the most interesting and useful applications of automatic computers. Computers can check not only the proofs of new mathematical theorems but also proofs that complex engineering systems and computer programs meet their specifications. Proofs to be checked by computer may be briefer and easier to write than the informal proofs acceptable to mathematicians. This is because the computer can be asked to do much more work to check each step than a human is willing to do, and this permits longer and fewer steps. [. . . ] The combination of proof-checking techniques with proof-finding heuristics will permit mathematicians to try out ideas for proofs that are still quite vague and may speed up mathematical research.

McCarthy's emphasis on the potential importance of applications to program verification may well have helped to shift the emphasis away from purely auto-

Figure 1: Proof-checking project for Morse's 'Set Theory'

matic theorem proving programs to interactive arrangements that could be of more immediate help in such work. A pioneering implementation of an interactive theorem prover in the modern sense was the `Proofchecker` program developed by Paul Abrahams [1963]. While Abrahams hardly succeeded in the ambitious goal of 'verification of textbook proofs, i.e. proofs resembling those that normally appear in mathematical textbooks and journals', he was able to prove a number of theorems from Principia Mathematica [Whitehead and Russell, 1910]. He also introduced in embryonic form many ideas that became significant later: a kind of macro facility for derived inference rules, and the integration of calculational derivations as well as natural deduction rules. Another interesting early proof checking effort [Bledsoe and Gilbert, 1967] was inspired by Bledsoe's interest in formalizing the already unusually formal proofs in his PhD adviser A.P. Morse's 'Set Theory' [Morse, 1965]; a flyer for a conference devoted to this research agenda is shown in Figure 1. We shall have more to say about Bledsoe's influence on our field later.

Perhaps the earliest sustained research program in interactive theorem proving was the development of the SAM (Semi-Automated Mathematics) family of

provers. This evolved over several years starting with SAM I, a relatively simple prover for natural deduction proofs in propositional logic. Subsequent members of the family supported more general logical formulas, had increasingly powerful reasoning systems and made the input-output process ever more convenient and accessible, with SAM V first making use of the then-modern CRT (cathode ray tube) displays. The provers were applied in a number of fields, and SAM V was used in 1966 to construct a proof of a hitherto unproven conjecture in lattice theory [Bumcrot, 1965], now called 'SAM's Lemma'. The description of SAM explicitly describes interactive theorem proving in the modern sense [Guard *et al.*, 1969]:

> Semi-automated mathematics is an approach to theorem-proving which seeks to combine automatic logic routines with ordinary proof procedures in such a manner that the resulting procedure is both efficient and subject to human intervention in the form of control and guidance. Because it makes the mathematician an essential factor in the quest to establish theorems, this approach is a departure from the usual theorem-proving attempts in which the computer *unaided* seeks to establish proofs.

Since the pioneering SAM work, there has been an explosion of activity in the area of interactive theorem proving, with the development of innumerable different systems; a few of the more significant contemporary ones are surveyed by Wiedijk [2006]. Despite this, it is difficult to find a general overview of the field, and one of the goals of this chapter is to present clearly some of the most influential threads of work that have led to the systems of today. It should be said at the outset that we focus on the systems we consider to have been seminal in the introduction or first systematic exploitation of certain key ideas, regardless of those systems' present-day status. The relative space allocated to particular provers should not be taken as indicative of any opinions about their present value *as systems*. After our survey of these different provers, we then present a more thematic discussion of some of the key ideas that were developed, and the topics that animate research in the field today.

Needless to say, the development of *automated* theorem provers has continued apace in parallel. The traditional ideas of first-order proof search and equational reasoning [Knuth and Bendix, 1970] have been developed and refined into powerful tools that have achieved notable successes in some areas [McCune, 1997; McCune and Padmanabhan, 1996]. The formerly neglected area of propositional tautology and satisfiability checking (SAT) underwent a dramatic revival, with systems in the established Davis-Putnam tradition making great strides in efficiency [Moskewicz *et al.*, 2001; Goldberg and Novikov, 2002; Eén and Sörensson, 2003], other algorithms being developed [Bryant, 1986; Stålmarck and Säflund, 1990], and applications to new and sometimes surprising areas appearing. For verification applications in particular, a quantifier-free combination of first-order theories [Nelson and Oppen, 1979; Shostak, 1984] has proven to be especially valuable and has led to the current SMT (satisfiability modulo theories) solvers. Some

more domain-specific automated algorithms have proven to be highly effective in areas like geometry and ideal theory [Wu, 1978; Chou, 1988; Buchberger, 1965], hypergeometric summation [Petkovšek *et al.*, 1996] and the analysis of finite-state systems [Clarke and Emerson, 1981; Queille and Sifakis, 1982; Burch *et al.*, 1992; Seger and Bryant, 1995], the last-mentioned (model checking) being of great value in many system verification applications. Indeed, some researchers reacted to the limitations of automation not by redirecting their energy away from the area, but by attempting to combine different techniques into more powerful AI-inspired frameworks like MKRP [Eisinger and Ohlbach, 1986] and Ωmega [Huang *et al.*, 1994].

Opinions on the relative values of automation and interaction differ greatly. To those familiar with highly efficient automated approaches, the painstaking use of interactive provers can seem lamentably clumsy and impractical by comparison. On the other hand, attacking problems that are barely within reach of automated methods (typically for reasons of time and space complexity) often requires prodigious runtime and/or heroic efforts of tuning and optimization, time and effort that might more productively be spent by simple problem reduction using an interactive prover. Despite important exceptions, the clear intellectual center of gravity of automated theorem proving has been the USA while for interactive theorem proving it has been Europe. It is therefore tempting to fit such preferences into stereotypical national characteristics, in particular the relative importance attached to efficiently automatable industrial processes versus the painstaking labor of the artisan. Such speculations aside, in recent years, we have seen something of a rapprochement: automated tools have been equipped with more sophisticated control languages [de Moura and Passmore, 2013], while interactive provers are incorporating many of the ideas behind automated systems or even using the tools themselves as components — we will later describe some of the methodological issues that arise from such combinations. Even today, we are still striving towards the optimal combination of human and machine that the pioneers anticipated 50 years ago.

## 2   AUTOMATH AND SUCCESSORS

Automath might be the earliest interactive theorem prover that started a tradition of systems which continues until today. It was the first program that used the *Curry-Howard isomorphism* for the encoding of proofs. There are actually two variants of the Curry-Howard approach [Geuvers and Barendsen, 1999], one in which a formula is represented by a type, and one in which the formulas are *not* types, but where with each formula a type of proof objects of that formula is associated. (The popular slogan 'formulas as types' only applies to the first variant, while the better slogan 'proofs as objects' applies to both.) The first approach is used by modern systems like Coq, Agda and NuPRL. The second approach is used in the LF framework [Harper *et al.*, 1987], and was also the one used in the Automath systems. The idea of the Curry-Howard isomorphism in

either style is that the type of 'proof objects' associated with a formula is non empty exactly in the case that that formula is true.

As an example, here is an Automath text that introduces implication and the two natural deduction rules for this connective (this text appears almost verbatim on pp. 23–24 of [de Bruijn, 1968b]). The other connectives of first order logic are handled analogously.

```
        * bool   := PN  : TYPE
        * b      := --- : bool
b       * TRUE   := PN  : TYPE
b       * c      := --- : bool
c       * impl   := PN  : bool
c       * asp1   := --- : TRUE(b)
asp1 * asp2      := --- : TRUE(impl)
asp2 * modpon := PN  : TRUE(c)
c       * asp4   := --- : [x,TRUE(b)]TRUE(c)
asp4 * axiom  := PN  : TRUE(impl)
```

This code first introduces (axiomatically: `PN` abbreviates 'Primitive Notion') a type for the formulas of the logic called `bool`[1], and for every such formula `b` a type of the 'proof objects' of that formula `TRUE(b)`. The `---` notation extends a context with a variable, where contexts are named by the last variable, and are indicated before the `*` in each line. Next, it introduces a function `impl(b,c)` that represents the implication $b \Rightarrow c$. Furthermore, it encodes the Modus Ponens rule

$$\frac{b \qquad b \Rightarrow c}{c}$$

using the function `modpon`. If `asp1` is a 'proof object' of type `TRUE(b)` and `asp2` is a 'proof object' of type `TRUE(impl(b,c))`, then the 'proof term' `modpon(b,c,asp1, asp2)` denotes a 'proof object' of type `TRUE(c)`. This term represents the syntax of the proof in first order logic using *higher order abstract syntax*. Finally, the rule

$$\frac{\begin{matrix} b \\ \vdots \\ c \end{matrix}}{b \Rightarrow c}$$

is encoded by the function `axiom`. If `asp4` is a 'function' that maps 'proof objects' of type `TRUE(b)` to those of type `TRUE(c)`, then `axiom(b,c,asp4)` is a 'proof object' of type `TRUE(impl(b,c))`.

---

[1]For a modern type theorist `bool` will be a strange choice for this name. However, in HOL the same name is used for the type of formulas (which shows that HOL is a classical system).

This Automath code corresponds directly to the modern typing judgments:

$$\texttt{bool} : *$$
$$\texttt{TRUE} : \texttt{bool} \rightarrow *$$
$$\texttt{impl} : \texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{bool}$$
$$\texttt{modpon} : \Pi b : \texttt{bool}.\,\Pi c : \texttt{bool}.\,\texttt{TRUE}\,b \rightarrow \texttt{TRUE}\,(\texttt{impl}\,b\,c) \rightarrow \texttt{TRUE}\,c$$
$$\texttt{axiom} : \Pi b : \texttt{bool}.\,\Pi c : \texttt{bool}.\,(\texttt{TRUE}\,b \rightarrow \texttt{TRUE}\,c) \rightarrow \texttt{TRUE}\,(\texttt{impl}\,b\,c)$$

The way one codes logic in LF style today is still exactly the same as it was in the sixties when Automath was first designed.

Note that in this example, the proof of $p \Rightarrow p$ is encoded by the term

$$\texttt{axiom}\,p\,p\,(\lambda H : \texttt{TRUE}(p).\,H)$$

which has type $\texttt{TRUE}\,(\texttt{impl}\,p\,p)$. In the 'direct' Curry-Howard style of Coq, Agda and NuPRL, $p$ is itself a type, and the term encoding the proof of $p \Rightarrow p$ becomes simply

$$\lambda H : p.\,p$$

which has type $p \rightarrow p$. Another difference between Automath and the modern type theoretical systems is that in Automath the logic and basic axioms have to be introduced axiomatically (as PN lines), while in Coq, Agda and NuPRL these are given by an 'inductive types' definitional package, and as such are defined using the type theory of the system.

The earliest publication about Automath is technical report number 68-WSK-05 from the Technical University in Eindhoven, dated November 1968 [de Bruijn, 1968a]. At that time de Bruijn already was a very successful mathematician, had been full professor of mathematics for sixteen years (first in Amsterdam and then in Eindhoven), and was fifty years old. The report states that Automath had been developed in the years 1967–1968. Two other people already were involved at that time: Jutting as a first 'user', and both Jutting and van Bree as programmers that helped with the first implementations of the language. These implementations were written in a variant of the Algol programming language (probably Algol 60, although Algol W was used at some point for Automath implementations too, and already existed by that time).

Automath was presented in December 1968 at the *Symposium on Automatic Demonstration*, held at INRIA Rocquencourt in Paris. The paper presented there was later published in 1970 in the proceedings of that conference [de Bruijn, 1968c].

The Automath system that is described in those two publications is very similar to the Twelf system that implements the LF logical framework. A formalization in this system essentially consists of a long list of definitions, in which a sequence of constants are defined as abbreviations of typed lambda terms. Through the Curry-Howard isomorphism this allows one to encode arbitrary logical reasoning. It appears that de Bruijn was not aware of the work by Curry and Howard at the time. Both publications mentioned contain no references to the literature,

and the notations used are very unlike what one would expect from someone who knew about lambda calculus. For example, function application is written with the argument *in front* of the function that is applied to it (which is indeed a more natural order), i.e., instead of MN one writes $\langle N \rangle M$, and lambda abstraction is not written as $\lambda x{:}A.M$ but as $[x{:}A]M$ (this notation was later inherited by the Coq system, although in modern Coq it has been changed.) Also, the type theory of Automath is quite different from the current type theories. In modern type theory, if we have the typings $M : B$ and $B : s$, then we have $(\lambda x{:}A.M) : (\Pi x{:}A.B)$ and $(\Pi x{:}A.B) : s$. However, in Automath one would have $([x{:}A]M) : ([x{:}A]B)$ and $([x{:}A]B) : ([x{:}A]s)$. In other words, in Automath there was no difference between $\lambda$ and $\Pi$, and while in modern type theory binders 'evaporate' after two steps when calculating types of types, in Automath they never will. This means that the typed terms in Automath do not have a natural set theoretic interpretation (probably the reason that this variant of type theory has been largely forgotten). However, this does *not* mean that this is not perfectly usable as a logical framework.

Apparently de Bruijn rediscovered the Curry-Howard isomorphism mostly independently (although he had heard from Heyting about the intuitionistic interpretation of the logical connectives). One of the inspirations for the Automath language was a manual check by de Bruijn of a very involved proof, where he wrote all the reasoning steps on a large sheet of paper [de Bruijn, 1990]. The scoping of the variables and assumptions were indicated by drawing lines in the proof with the variables and assumptions. written in a 'flag' at the top of this line. This is very similar to Jaśkowski-Fitch style natural deduction, but in Automath (just like in LF) this is not tied to a specific logic.

There essentially have been four groups of Automath languages, of which only the first two have ever been properly implemented:

**AUT-68** This was the first variant of Automath, a simple and clean system, which was explained in the early papers through various weaker and less practical systems, with names like PAL, LONGPAL, LONGAL, SEMIPAL and SEMIPAL 2, where 'PAL' abbreviates 'Primitive Automath Language' [de Bruijn, 1969; de Bruijn, 1970]. Recently there has been a revival of interest in these systems from people investigating weak logical frameworks [Luo, 2003].

**AUT-QE** This was the second version of the Automath language. 'QE' stands for 'Quasi-Expressions'. With this Automath evolved towards the current type theories (although it still was quite different), one now *both* had $([x{:}A]B) : ([x{:}A]s)$ as well as $([x{:}A]B) : s$. This was called type inclusion. AUT-QE is the dialect of Automath in which the biggest formalization, Jutting's translation of Landau's *Grundlagen*, was written [van Benthem Jutting, 1979]. It has much later been re-implemented in C by one of the authors of this chapter [Wiedijk, 2002].

**Later AUT languages** These are later variants of Automath, like AUT-QE-NTI (a subset of AUT-QE in which subtyping was removed, the 'NTI' standing

for 'no type inclusion'), and the AUT-Π and AUT-SYNTH extensions of AUT-QE. These languages were modifications of the AUT-QE framework, but although implementations were worked on, it seems none of them was really finished.

**AUT-SL** This was a very elegant version of Automath developed by de Bruijn (with variants of the same idea also developed by others). In this language the distinction between definitions and redexes is removed, and the formalization, including the definitions, becomes a single very large lambda term. The 'SL' stands for 'single line' (Section B.2 of [Nederpelt *et al.*, 1994]). The system also was called $\Delta\Lambda$, and sometimes $\Lambda\Delta$ (Section B.7 of [Nederpelt *et al.*, 1994]). A more recent variant of this system was De Groote's $\lambda^\lambda$ type theory [de Groote, 1993]. The system AUT-QE-NTI can be seen as a step towards the AUT-SL language.

There were later languages, by de Bruijn and by others, that were more loosely related to the Automath languages. One of these was WOT, the abbreviation of 'wiskundige omgangstaal', Dutch for *mathematical vernacular* [de Bruijn, 1979]. Unlike Trybulec with Mizar, de Bruijn only felt the need to have this 'vernacular' be *structurally* similar to actual mathematical prose, and never tried to make it natural language-like.

In 1975, the Dutch science foundation ZWO (nowadays called NWO) gave a large five year grant for the project *Wiskundige Taal AUTOMATH* to further develop the Automath ideas. From this grant five researchers, two programmers and a secretary were financed [de Bruijn, 1978]. During the duration of this project many students of the Technical University Eindhoven did formalization projects. Examples of the subjects that were formalized were:

- two treatments of set theory

- a basic development of group theory

- an axiomatic development of linear algebra

- the König-Hall theorem in graph theory

- automatic generation of Automath texts that prove arithmetic identities

- the sine and cosine functions

- irrationality of $\pi$

- real numbers as infinite sequences of the symbols 0 and 1

We do not know how complete these formalizations were, nor whether they were written using an Automath dialect that actually was implemented.

In 1984 De Bruijn retired (although he stayed scientifically active), and the Automath project was effectively discontinued. In 1994 a volume containing the

most important Automath papers was published [Nederpelt *et al.*, 1994], and in
2003 most other relevant documents were scanned, resulting in the Automath
Archive, which is freely available on the web [Scheffer, 2003].

Automath has been one of the precursors of a development of type systems
called *type theory*:

- On the predicative side of things there were the type theories by Martin-
  Löf, developed from 1971 on (after discovery of the Girard paradox, in 1972
  replaced by an apparently consistent system), which among other things
  introduced the notion of inductive types [Nordström *et al.*, 1990].

- On the impredicative side there were the polymorphic lambda calculi by
  Girard (1972) and Reynolds (1974). This was combined with the depen-
  dent types from Martin-Löf's type theory in Coquand's CC (the Calculus
  of Constructions), described in his PhD thesis [Coquand and Huet, 1988].
  CC was structured by Barendregt into the eight systems of the lambda cube
  [Barendregt, 1992], which was then generalised into the framework of *pure
  type systems* (PTSs) by Berardi (1988) and Terlouw (1989).

Both the Martin-Löf theories and the Calculus of Constructions were further de-
veloped and merged in various systems, like in ECC (Extended Calculus of Con-
structions) [Luo, 1989] and UTT (Universal Type Theory) [Luo, 1992], and by
Paulin in CIC (the Calculus of Inductive Constructions) [Coquand and Paulin,
1990], later further developed into pCIC (the predicative version of CIC, which
also was extended with coinductive types), the current type theory behind the
Coq system [Coq development team, 2012].

All these type theories are similar and different (and have grown away from
the Automath type theory). Two of the axes on which one might compare them
are *predicativity* ('objects are not allowed to be defined using quantification over
the domain to which the object belongs') and *intensionality* ('equality between
functions is not just determined by whether the values of the functions coincide').
For example, the type theory of Coq is not yet predicative but it is intensional,
the type theory of NuPRL is predicative but it is not intensional, and the type
theory of Agda is both predicative and intensional.

Recently, there has been another development in type theory with the introduc-
tion of *univalent type theory* or *homotopy type theory* (HoTT) by Voevodsky in
2009 [Univalent Foundations Program, 2013]. Here type theory is extended with
an interpretation of equality as homotopy, which gives rise to the *axiom of uni-
valence*. This means that representation independence now is hardwired into the
type theory. For this reason, some people consider HoTT to be a new foundation
for mathematics. Also, in this system the inductive types machinery is extended to
*higher inductive types* (inductive types where equalities can be added inductively
as well). Together, this compensates for several problems when using Coq's type
theory for mathematics: one gets functional extensionality and can have proper
definitions of subtypes and quotient types. This field is still young and in very
active development.

We now list some important current systems that are based on type theory and the Curry-Howard isomorphism, and as such can be considered successors to Automath. We leave out the history of important historical systems like LEGO [Pollack, 1994], and focus on systems that still have an active user community. For each we give a brief overview of their development.

## NuPRL

In 1979 Martin-Löf introduced an extensional version of his type theory. In the same year Bates and Constable, after earlier work on the PL/CV verification framework [Constable and O'Donnell, 1978] had founded the PRL research group at Cornell University to develop a program development system where programs are created in a mathematical fashion by interactive refinement (PRL at that time stood for Program Refinement Logic). In this group various systems were developed: the AVID system (Aid Verification through the techniques of Interactive program Development) [Krafft, 1981], the Micro-PRL system, also in 1981, and the λPRL system [Constable and Bates, 1983].

In 1984 the PRL group implemented a variant of Martin-Löf's extensional type theory in a system called NuPRL (also written as νPRL, to be read as 'new PRL'; PRL now was taken to stand for Proof Refinement Logic). This system [Constable, 1986] has had five versions, where NuPRL 5, the latest version, is also called NuPRL LPE (Logical Programming Environment). In 2003, a new architecture for NuPRL was implemented called MetaPRL (after first having been called NuPRL-Light) [Hickey *et al.*, 2003]. The NuPRL type theory always had a very large number of typing rules, and in the MetaPRL system this is handled through a logical framework. In that sense this system resembles Isabelle.

Part of the NuPRL/MetaPRL project is the development of a library of formal results called the FDL (Formal Digital Library).

## Coq

In 1984 Huet and Coquand at INRIA started implementing the Calculus of Constructions in the CAML dialect of ML. Originally this was just a type checker for this type theory, but with version 4.10 in 1989 the system was extended in the style of the LCF system, with tactics that operate on goals. Subsequently many people have worked on the system. Many parts have been redesigned and re-implemented several times, including the syntax of the proof language and the kernel of the system. The Coq manual [Coq development team, 2012] gives an extensive history of the development of the system, which involved dozens of researchers. Of these a majority made contributions to the system that have all turned out to be essential for its efficient use. Examples of features that were added are: coercions, canonical structures, type classes, coinductive types, universe polymorphism, various decision procedures (e.g., for equalities in rings and fields, and for linear arithmetic), various tactics (e.g., for induction and inversion, and for rewriting with a congru-

ence, in type theory called 'setoid equality'), mechanisms for customizing term syntax, the `coqdoc` documentation system, the `Ltac` scripting language and the `Program` command (which defines yet another functional programming language within the Coq system). The system nowadays is a very feature rich environment, which makes it the currently most popular interactive theorem prover based on type theory.

The latest released version of Coq is 8.4. This system can be seen as a theorem prover, but also as an implementation of a functional programming language with an execution speed comparable to functional languages like OCaml. A byte code machine similar to the one of OCaml was implemented by Grégoire and Leroy, but there now also is native code compilation, implemented by Denes and Grégoire. Also, computations on small integers can be done on the machine level, due to Spiwack.

Another feature of Coq is that Coq programs can be exported in the syntax of other functional programming languages, like OCaml and Haskell.

Coq has more than one user interface, of which Proof General [Aspinall, 2000] and CoqIDE [Bertot and Théry, 1998] are currently the most popular.

There are two important extensions of Coq. First there is the SSReflect proof language and associated mathematical components library by Gonthier and others [Gonthier *et al.*, 2008; Gonthier and Mahboubi, 2010], which was developed for the formalization of the proofs of the 4-color theorem (2005) and the Feit-Thompson theorem (2012). This is a compact and powerful proof language, which has not been merged in the mainstream version of Coq. Second there are implementations of Coq adapted for homotopy type theory.

Finally there is the Matita system from Bologna by Asperti and others [Asperti *et al.*, 2006]. This started out in 2004 as an independent implementation of a type checker of the Coq type theory. It was developed in the HELM project (Hypertextual Electronic Library of Mathematics), which was about presenting Coq libraries on the web [Asperti *et al.*, 2003], and therefore at the time was quite similar to Coq, but has in the meantime diverged significantly, with many improvements of its own.

## Twelf

In 1987, Harper, Honsell and Plotkin introduced the Edinburgh Logical Framework, generally abbreviated as Edinburgh LF, or just LF [Harper *et al.*, 1987]. This is a quite simple predicative type theory, inspired by and similar to Automath, in which one can define logical systems in order to reason about them. An important property of an encoded logic, which has to be proved on the meta level, is *adequacy*, the property that the beta-eta long normal forms of the terms that encode proofs in the system are in one-to-one correspondence with the proofs of the logic themselves.

A first implementation of LF was EFS (Environment for Formal Systems) [Griffin, 1988]. Soon after, in 1989, Pfenning implemented the Elf system, which added

a meta-programming layer [Pfenning, 1994]. In 1999 a new version of this system was implemented by Pfenning and Schürmann, called Twelf [Pfenning and Schürmann, 1999]. In the meta layer of Twelf, one can execute Twelf specifications as logic programs, and it also contains a theorem prover that can establish properties of the Twelf encoding automatically, given the right definitions and annotations.

## *Agda*

In 1990 a first implementation of a type checker for Martin-Löf's type theory was created by Coquand and Nordström. In 1992 this turned into the ALF (Another Logical Framework) system, implemented by Magnusson in SML [Magnusson and Nordström, 1993]. Subsequently a Haskell implementation of the same system was worked on by Coquand and Synek, called Half (Haskell Alf). A C version of this system called CHalf, also by Synek, was used for a significant formalization by Cederquist of the Hahn-Banach theorem in 1997 [Cederquist *et al.*, 1998]. Synek developed for this system an innovative Emacs interface that allows one to work in a procedural style on a proof that essentially is declarative [Coquand *et al.*, 2005].

A new version of this system called Agda was written by Catarina Coquand in 1996, for which a graphical user interface was developed around 2000 by Hallgren. Finally Norell implemented a new version of this system in 2007 under the name of Agda2 [Bove *et al.*, 2009]. For a while there were two different versions of this system, a stable version and a more experimental version, but by now there is just one version left.

## 3  LCF AND SUCCESSORS

The LCF approach to interactive theorem proving has its origins in the work of Robin Milner, who from early in his career in David Cooper's research group in Swansea was interested specifically in *interactive* proof:

> I wrote an automatic theorem prover in Swansea for myself and became shattered with the difficulty of doing anything interesting in that direction and I still am. I greatly admired Robinson's resolution principle, a wonderful breakthrough; but in fact the amount of stuff you can prove with fully automatic theorem proving is still very small. So I was always more interested in amplifying human intelligence than I am in artificial intelligence.[2]

Milner subsequently moved to Stanford where he worked in 1971–2 in John McCarthy's AI lab. There he, together with Whitfield Diffie, Richard Weyhrauch and Malcolm Newey, designed an interactive proof assistant for what Milner called the *Logic of Computable Functions* (LCF). This formalism, devised by Dana Scott in

---

[2]`http://www.sussex.ac.uk/Users/mfb21/interviews/milner/`

1969, though only published much later [Scott, 1993], was intended for reasoning about recursively defined functions on complete partial orders (CPOs), such as typically occur in the Scott-Strachey approach to denotational semantics. The proof assistant, known as Stanford LCF [Milner, 1972], was intended more for applications in computer science rather than mainstream pure mathematics. Although it was a proof checker rather than an automated theorem prover, it did provide a powerful automatic simplification mechanism and convenient support for backward, goal-directed proof.

There were at least two major problems with Stanford LCF. First, the storage of proofs tended to fill up memory very quickly. Second, the repertoire of proof commands was fixed and could not be customized. When he moved to Edinburgh, Milner set about fixing these defects. With the aid of his research assistants, Lockwood Morris, Malcolm Newey, Chris Wadsworth and Mike Gordon, he designed a new system called Edinburgh LCF [Gordon *et al.*, 1979]. To allow full customizability, Edinburgh LCF was embedded in a general programming language, ML.[3] ML was a higher-order functional programming language, featuring a novel polymorphic type system [Milner, 1978] and a simple but useful exception mechanism as well as imperative features. Although the ML language was invented as part of the LCF project specifically for the purpose of writing proof procedures, it has in itself been seminal: many contemporary functional languages such as CAML Light and OCaml [Cousineau and Mauny, 1998; Weis and Leroy, 1993] are directly descended from it or at least heavily influenced by it, and their applications go far beyond just theorem proving.

In LCF, recursive (tree-structured) types are defined in the ML metalanguage to represent the (object) logical entities such as types, terms, formulas and theorems. For illustration, we will use `thm` for the ML type of theorems, though the exact name is not important. Logical inference rules are then realized concretely as functions that return an object of type `thm`. For example, a classic logical inference rule is *Modus Ponens* or $\Rightarrow$-elimination, which might conventionally be represented in a logic textbook or paper as a rule asserting that *if* $p \Rightarrow q$ and $p$ are both provable (from respective assumptions $\Gamma$ and $\Delta$) *then* $q$ is also provable (from the combined assumptions):[4]

$$\frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

The LCF approach puts a concrete and computational twist on this by turning each such rule into a *function* in the metalanguage. In this case the function, say `MP`, takes two theorems as input, $\Gamma \vdash p \Rightarrow q$ and $\Delta \vdash p$, and *returns* the theorem $\Gamma \cup \Delta \vdash q$; it therefore has a function type `thm->thm->thm` in ML (assuming

---

[3]ML for metalanguage; following Tarski [1936] and Carnap [1937], it has become usual in logic and linguistics to to distinguish carefully the object logic and the metalogic (which is used to reason about the object logic).

[4]We show it in a sequent context where we also take the union of assumption lists. In a Hilbert-style proof system these assumptions would be absent; in other presentations we might assume that the set of hypotheses are the same in both cases and have a separate weakening rule. Such fine details of the logical system are orthogonal to the ideas we are explaining here.

curried functions). When logical systems are presented, it's common to make some terminological distinctions among the components of the foundation, and all these get reflected in the types in the metalanguage when implemented in LCF style:

- An *axiom* is simply an element of type `thm`

- An *axiom schema* (for example a first-order induction principle with an instance for each formula) becomes a function that takes some argument(s) like a term indicating which instance is required, and returns something of type `thm`.

- A true inference rule becomes an ML object like the `MP` example above that takes objects, at least one of which is of type `thm`, as arguments and returns something of type `thm`.

The traditional idea of logical systems is to use them as a foundation, by choosing once and for all some relatively small and simple set of rules, axioms and axiom schemas, which we will call the *primitive inference rules*, and thereafter perform all proof using just those primitives. In an LCF prover one can, if one wishes, create arbitrary proofs using these logical inference rules, simply by composing the ML functions appropriately. Although a proof is always *performed*, the proof itself exists only ephemerally as part of ML's (strict) evaluation process, and therefore no longer fills up memory. Gordon [2000] makes a nice analogy with writing out a proof on a blackboard, and rubbing out early steps to make room for more. In order to retain a guarantee that objects of type `thm` really were created by application of the primitive rules, Milner had the ingenious idea of making `thm` an abstract type, with the primitive rules as its only constructors. After this, one simply needs to have confidence in the fairly small amount of code underlying the primitive inference rules to be quite sure that all theorems must have been properly deduced simply because of their type.

But even for the somewhat general meta-arguments in logic textbooks, and certainly for concretely performing proofs by computer, the idea of proving something non-trivial by decomposing it to primitive inference rules is usually daunting in the extreme. In practice one needs some other *derived rules* embodying convenient inference patterns that are not part of the axiomatic basis but can be derived from them. A derived inference rule too has a concrete realization in LCF systems as a function whose definition composes other inference rules, and using parametrization by the function arguments can work in a general and schematic way just like the metatheorems of logic textbooks. For example, if we also have a primitive axiom schema called `ASSUME` returning a theorem of the form $p \vdash p$:

$$\frac{}{\{p\} \vdash p}$$

then we can implement the following *derived inference rule*, which we will call
`UNDISCH`:

$$\frac{\Gamma \vdash p \Rightarrow q}{\Gamma \cup \{p\} \vdash q}$$

as a simple function in the metalanguage. For example, the code might look some-
thing like the following. It starts by breaking apart the implication of the input
theorem to determine the appropriate $p$ and $q$. (Although objects of type `thm`
can only be *constructed* by the primitive rules, they can be examined and decon-
structed freely.) Based on this, the appropriate instance of the `ASSUME` schema is
used and the two inference rules plugged together.

```
let UNDISCH th =
  let Imp(p,q) = concl th in
  MP th (ASSUME p);;
```

This is just a very simple example, but because a full programming language
is available, one can implement much more complex derived rules that perform
sophisticated reasoning and automated proof search but still ultimately reduce
to the primitive rules. Indeed, although LCF and most of its successors use a
traditional forward presentation of logic, it is easy to use a layer of programming
to support goal-directed proof in the style of Stanford LCF, via so-called *tactics*.

This flexibility gives LCF an appealing combination of reliability and exten-
sibility. In most theorem proving systems, in order to install new facilities it is
necessary to modify the basic code of the prover. But in LCF an ordinary user
can write an arbitrary ML program to automate a useful inference pattern, while
all the time being assured that even if the program has bugs, no false theorems
will arise (though the program may fail in this case, or produce a valid theorem
other than the one that was hoped for). As Slind [1991] puts it 'the user controls
the means of (theorem) production'.

LCF was employed in several applications at Edinburgh, and this motivated
certain developments in the system. By now, the system had attracted attention
elsewhere. Edinburgh LCF was ported to LeLisp and MacLisp by Gérard Huet,
and this formed the basis for a rationalization and redevelopment of the system by
Paulson [1987] at Cambridge, resulting in Cambridge LCF. First, Huet and Paul-
son modified the ML system to generate Lisp code that could be compiled rather
than interpreted, which greatly improved performance. Among many other im-
provements Paulson [1983] replaced Edinburgh LCF's complicated and monolithic
simplifier with an elegant scheme based on on *conversions*.

A conversion is a particular kind of derived rule (of ML type `:term->thm`) that
given a term $t$ returns a *theorem* of the form $\Gamma \vdash t = t'$ for some other term $t'$.
(For example, a conversion for addition of numbers might map the term $2 + 3$
to the theorem $\vdash 2 + 3 = 5$.) This gives a uniform framework for converting
ad hoc simplification routines into those that are justified by inference: instead
of simply taking $t$ and asserting its equality to $t'$, we actually carry *theorems*
asserting such equivalences through the procedure. Via convenient higher-order

functions, conversions can be combined in various ways, applied recursively in a depth-first fashion etc., with all the appropriate inference to plug the steps together (transitivity and congruences and so on) happening automatically.

## 3.1 HOL

As emphasized by Gordon [1982], despite the name 'LCF', nothing in the Edinburgh LCF methodology is tied to the Logic of Computable Functions. In the early 1980s Gordon, now in Cambridge, as well as supervising Paulson's development of LCF, was interested in the formal verification of hardware. For this purpose, classical higher order logic seemed a natural vehicle, since it allows a rather direct rendering of notions like signals as functions from time to values. The case was first put by Hanna and Daeche [1986] and, after a brief experiment with an ad hoc formalism 'LSM' based on Milner's Calculus of Communicating Systems, Gordon [1985] also became a strong advocate.

Gordon modified Cambridge LCF to support classical higher order logic, and so HOL (for Higher Order Logic) was born. Following Church [1940], the system is based on simply typed $\lambda$-calculus, so all terms are either variables, constants, applications or abstractions; there is no distinguished class of formulas, merely terms of boolean type. The main difference from Church's system is that polymorphism is an object-level, rather than a meta-level, notion; essentially the same Hindley-Milner automated typechecking algorithm used in ML [Milner, 1978] is used in the interface so that most general types for terms can be deduced automatically. Using defined constants and a layer of parser and pretty-printer support, many standard syntactic conventions are broken down to $\lambda$-calculus. For example, the universal quantifier, following Church, is simply a higher order function, but the conventional notation $\forall x.P[x]$ is supported, mapping down to $\forall(\lambda x.P[x])$. Similarly there is a constant LET, which is semantically the identity and is used only as a tag for the pretty-printer, and following Landin [1966], the construct 'let $x = t$ in $s$' is broken down to 'LET $(\lambda x.s)$ $t$'.[5] The advantage of keeping the internal representation simple is that the underlying proof procedures, e.g. those that do term traversal during simplification, need only consider a few cases.

The exact axiomatization of the logic was partly influenced by Church, partly by the way things were done in LCF, and partly through consultation with the logicians Mike Fourman, Martin Hyland and Andy Pitts in Cambridge. HOL originally included a simple constant definitional mechanism, allowing new equational axioms of the form $\vdash c = t$ to be added, where $t$ is a closed term and $c$ a new constant symbol. A mechanism for defining new types, due to Mike Fourman, was also included. Roughly speaking one may introduce a new type in bijection with any nonempty subset of an existing type (identified by its characteristic predicate). An important feature of these definitional mechanisms bears emphasizing: they are not metalogical translations, but means of extending the signature of the object

---

[5]Landin, by the way, is credited with inventing the term 'syntactic sugar', as well as this notable example of it.

logic, while guaranteeing that such extension preserves consistency. In fact, the definitional principles are *conservative*, meaning roughly that no new statements not involving the defined concept become provable as a result of the extension.

HOL emphasized the systematic development of theories by these principles of conservative extension to the point where it became the norm, purely axiomatic extensions becoming frowned on. Such an approach is obviously a very natural fit with the LCF philosophy, since it entails pushing back the burden of consistency proofs or whatever to the beginning, once and for all, such that all extensions, whether of the theory hierarchy or proof mechanisms, are correct by construction. (Or at least *consistent* by construction. Of course, it is perfectly possible to introduce definitions that do not correspond to the intuitive notion being formalized, but no computable process can resolve such difficulties.) This contrasts with LCF, where there was no distinction between definitions and axioms, and new types were often simply characterized by axioms without any formal consistency proof. Though there was usually a feeling that such a proof would be routine, it is easy to make mistakes in such a situation. It can be much harder to produce useful structures by definitional extension than simply to assert suitable axioms — the advantages were likened by Russell [1919] to those of theft over honest toil.

For example, Melham's derived definitional principle [Melham, 1989] for recursive data types was perhaps at the time the most sophisticated LCF-style derived rule ever written, and introduced important techniques for maintaining efficiency in complex rules that are still used today — we discuss the issues around efficient implementations of decision procedures later. This was the first of a wave of derived definitional principles in LCF-like systems for defining inductive or coinductive sets or relations [Andersen and Petersen, 1991; Camilleri and Melham, 1992; Roxas, 1993; Paulson, 1994a], general recursive functions [Ploegaerts *et al.*, 1991; van der Voort, 1992; Slind, 1996; Krauss, 2010], quotient types with automated lifting of definitions and theorems [Harrison, 1998; Homeier, 2005], more general forms of recursive datatypes with infinite branching, nested and mutual recursion or dual codatatypes [Gunter, 1993; Harrison, 1995a; Berghofer and Wenzel, 1999; Blanchette *et al.*, 2014] as well as special *nominal* datatypes to formalize variable binding in a natural way [Urban, 2008]. Supporting such complex definitions as a primitive aspect of the logic, done to some extent in systems as different as ACL2 and Coq, is a complex, intricate and error-prone activity, and there is a lot said for how the derived approach maintains foundational simplicity and security. In fact general wellfounded recursive functions in Coq are also supported using derived definitional principles [Balaa and Bertot, 2000], while quotients in the current foundations of Coq are problematic for deeper foundational reasons too.

The HOL system was consolidated and rationalized in a major release in late 1988, which was called, accordingly, HOL88. It became fairly popular, acquired good documentation, and attracted many users around the world. Nevertheless, despite its growing polish and popularity, HOL88 was open to criticism. In particular, though the higher-level parts were coded directly in ML, most of the term operations below were ancient and obscure Lisp code (much of it probably written

HOL88

hol90                ProofPower    Isabelle/HOL
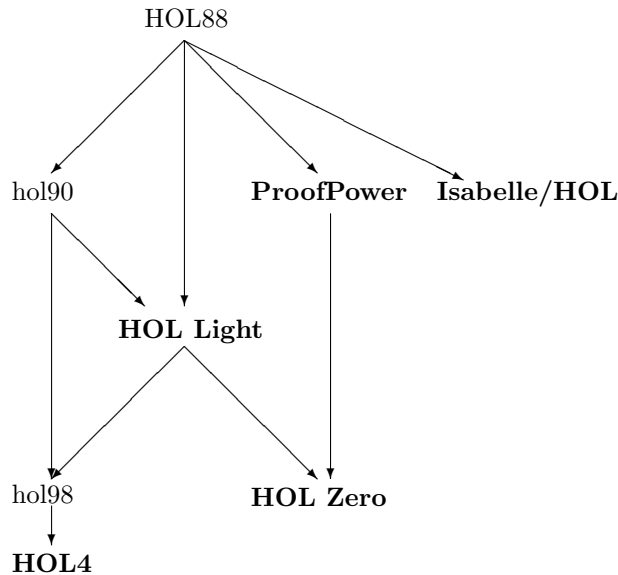
HOL Light

hol98                HOL Zero

HOL4

Figure 2: The HOL family tree

by Milner in the 1970s). Moreover, ML had since been standardized, and the new Standard ML seemed a more promising vehicle for the future than Lisp, especially with several new compilers appearing at the time. These considerations motivated two new versions of HOL in Standard ML. One was developed by Roger Jones, Rob Arthan and others at ICL Secure Systems and called ProofPower. This was intended as a commercial product and has been mainly used for high-assurance applications, though the current version is freely available and has also been used in other areas like the formalization of mathematics.[6] The other, called hol90, was written by Konrad Slind [1991], under the supervision of Graham Birtwistle at the University of Calgary. The entire system was coded in Standard ML, which made all the pre-logic operations such as substitution accessible. Subsequently several other versions of HOL were written, including HOL Light, a version with a simplified axiomatization and rationalized structure written in CAML Light by one of the present authors and subsequently ported to OCaml [Harrison, 2006a], and Isabelle/HOL, described in more detail in the next section. The 'family DAG' in Figure 2 gives an approximate idea of some of the influences. While HOL88, hol90 and hol98 are little used today (though HOL88 is available as a Debian package), all the other provers in this picture are under active development and/or have significant user communities.

---

[6]See http://www.lemma-one.com/ProofPower/index/.

## 3.2   Isabelle

We will discuss one more LCF-style system in a little more depth because it has some distinguishing features compared to others in the family, and is also perhaps currently the most widely used member of the LCF family. This is the Isabelle system, originally developed by Paulson [1990]. The initial vision of Isabelle was as an LCF-style logical framework in which to embed other logics. Indeed, the subtitle 'The Next 700 Theorem Provers' in Paulson's paper (with its nod to Landin's 'The Next 700 Programming Languages') calls attention to the proliferation of different theorem proving systems already existing at the time. Many researchers, especially in computer science, were (and still are) interested in proof support for particular logics (classical, constructive, many-valued, temporal etc.) While these all have their distinctive features, they also have many common characteristics, making the appeal of a re-usable generic framework obvious.

Isabelle effectively introduces yet another layer into the meta-object distinction, with the logic directly implemented in the LCF style itself being considered a *meta-logic* for the embedding of other logics. The Isabelle metalogic is a simple form of higher-order logic. It is intentionally weak (for example, having no induction principles) so that it does not in itself introduce foundational assumptions that some might find questionable or cause incompatibilities with the way object logics are embedded. But it serves its purpose well as a framework for embedding object logics and providing a common infrastructure across them. The inference rules in the object logic are then given in a declarative fashion as meta-implications (implications in the meta-logic). For example, our earlier example of Modus Ponens can be represented as the following (meta) theorem. The variables starting with '?' are *metavariables*, i.e. variables in the metalogic; $\rightarrow$ denotes object-level implication while $\Rightarrow$ denotes meta-level implication.

$$[?P \rightarrow ?Q; ?P] \Rightarrow ?Q$$

By representing object-level inference rules in this fashion, the actual implementations often just need to perform forward or backward chaining with matching and/or unification. Isabelle supports the effective automation of this process with a powerful higher-order unification algorithm [Huet, 1975] giving a kind of higher-order resolution principle.

Many design decisions in Isabelle were based on Paulson's experience with Cambridge LCF and introduced a number of improvements. In particular, backward proof ('tactics') in LCF actually worked by iteratively creating function closures to eventually reverse the refinement process into a sequence of the primitive forward rules. This non-declarative formulation meant, for example, that it was a non-trivial change to add support in LCF for logic variables allowing the instantiation of existential goals to be deferred [Sokołowski, 1983]. Isabelle simply represented goals as theorems, with tactics effectively working forwards on their assumptions, making the whole framework much cleaner and giving metavariables with no extra effort. This variant of tactics was also adopted

in the ProofPower and LAMBDA systems (see next section). Isabelle's tactic mechanism also allowed backtracking search over lazy lists of possible outcomes in its tactic mechanism. Together with unification, this framework could be used to give very simple direct implementations of some classic first-order proof search algorithms like tableaux à la lean$T^AP$ [Beckert and Posegga, 1995] (`fast_tac` in Isabelle) and the Loveland-Stickel presentation [Loveland, 1978; Stickel, 1988] of model elimination (`meson_tac`). While nowadays largely superseded by much more powerful automation of the kind that we consider later, these simple tactics were at the time very convenient in making typical proofs less low-level.

It's customary to use appellations like 'Isabelle/X' to describe the particular instantiation of Isabelle with object logic X. Among the many object logics embedded in Isabelle are constructive type theory, classical higher order logic (Isabelle/HOL) and first-order Zermelo-Fraenkel set theory (Isabelle/ZF) [Paulson, 1994b]. Despite this diversity, only a few have been extensively used. Some axiom of choice equivalences have been formalized in Isabelle/ZF [Paulson and Grąbczewski, 1996], as has Gödel's hierarchy $L$ of *constructible* sets leading to a proof of the relative consistency of the Axiom of Choice [Paulson, 2003]. But by far the largest user community has developed around the Isabelle/HOL instantiation [Nipkow *et al.*, 2002]. This was originally developed by Tobias Nipkow as an instantiation of Isabelle with something very close to the logic of the various HOL systems described above, but with the addition (at the level of the metalogic) of a system of axiomatic type classes similar to those of Haskell. In this instantiation, the ties between the Isabelle object and metalogic are particularly intimate.

Since its inception, Isabelle/HOL has become another full-fledged HOL implementation. In fact, from the point of view of the typical user the existence of a separate metalogic can largely be ignored, so the effective common ground between Isabelle/HOL and other HOL implementations is closer than might be expected. However, one more recent departure (not limited to the HOL instantiation of Isabelle) takes it further from its LCF roots and other HOL implementations. This is the adoption of a structured proof language called Isar, inspired by Mizar [Wenzel, 1999]. For most users, this is the primary interaction language, so they no longer use the ML read-eval-print loop as the interface. The underlying LCF mechanisms still exist and can be accessed, but many facilities are mediated by Isar and use a sophisticated system of contexts. We describe some of the design decisions in proof languages later in this chapter.

## 3.3   *Other LCF systems*

There have been quite a few other theorem provers either directly implemented in the LCF style or at least heavily influenced by it. Some of them, such as NuPRL and Coq, we have discussed above because of their links to constructive type theory, and so we will not discuss them again here, but their LCF implementation pedigree is also worth noting. For example, Bates and Constable [1985] describe

the LCF approach in detail and discuss how NuPRL developed from an earlier system PL/CV. Another notable example is the LAMBDA (Logic And Mathematics Behind Design Automation) system, which was developed in a team led by Mike Fourman for use in hardware verification. Among other distinctive features, it uses a logic of partial functions, as did the original LCF system and as does the non-LCF system IMPS [Farmer *et al.*, 1990].

## 4 MIZAR

The history of Mizar [Matuszewski and Rudnicki, 2005] is a history of a team of Polish mathematicians, linguists and computer scientists analyzing mathematical texts and looking for a satisfactory human-oriented formal counterpart. One of the mottos of this effort has been Kreisel's *'ENOD: Experience, Not Only Doctrine'* [Rudnicki and Trybulec, 1999], which in the Mizar context was loosely understood as today's ideas on rapid/agile software development. There were Mizar prototypes with semantics that was only partially clear, and with only partial verification procedures. A lot of focus was for a long time on designing a suitable language and on testing it by translating real mathematical papers into the language. The emphasis has not been just on capturing common patterns of reasoning and theory development, but also on capturing common syntactic patterns of the language of mathematics. A Mizar text is not only supposed to be written by humans and then read by machines, but it is also supposed to be directly easily readable by humans, avoid too many parentheses, quotation marks, etc.

The idea of such a language and system was proposed in 1973 by Andrzej Trybulec, who was at that time finishing his PhD thesis in topology under Karol Borsuk, and also teaching at the Płock Branch of the Warsaw University of Technology. Trybulec had then already many interests: since 1967 he had been publishing on topics in topology and linguistics, and in Płock he was also running the Informatics Club. The name Mizar (after a star in Big Dipper)[7] was proposed by Trybulec's wife Zinaida, originally for a different project. The writing of his PhD thesis and incorporating of Borsuk's feedback prompted Trybulec to think about languages and computer systems that would help mathematicians with such tasks. He presented these ideas for the first time at a seminar at the Warsaw University on November 14, 1973, and was soon joined by a growing team of collaborators that were attracted by his vision[8] and personality, in many cases for their whole lives: Piotr Rudnicki, Czesław Byliński, Grzegorz Bancerek, Roman Matuszewski, Artur Kornilowicz, Adam Grabowski and Adam Naumowicz, to name just a few.

---

[7]Some backronyms related to Mathematics and Informatics have been proposed later.

[8]It was easy to get excited, for several reasons. Andrzej Trybulec and most of the Mizar team have been a showcase of the popularity of science fiction in Poland and its academia. A great selection of world sci-fi books has been shared by the Mizar team, by no means limited to famous Polish sci-fi authors such as Stanisław Lem. Another surprising and inspiring analogy appeared after the project moved in 1976 to Białystok: the city where Ludwik Zamenhof grew up and started to create Esperanto in 1873 [Zalewska, 2010] – 100 years before the development of Mizar started.

The total count of Mizar authors in May 2014 grew to 246. In his first note (June 1975 [Trybulec, 1977]) about Mizar, Trybulec called such languages *Logic-Information Languages* (LIL) and defined them as *facto-graphic languages that enable recording of both facts from a given domain as well as logical relationships among them.* He proposed several applications for such languages, such as:

- Input to information retrieval systems which use logical dependencies.

- Intermediate languages for machine translation (especially of science).

- Automation of the editorial process of scientific papers, where the input language is based on LIL and the output language is natural (translated into many languages).

- Developing verification procedures for such languages, not only in mathematics, but also in law and medicine, where such procedures would typically interact with a database of relevant facts depending on the domain.

- Education, especially remote learning.

- Artificial intelligence research.

For the concrete instantiation to mathematics, the 1975 note already specified the main features of what is today's Mizar, noting that although such a vision borders on science fiction, it is a proper research direction:

- The paper should be directly written in a LIL.

- The paper is checked automatically for syntactic and some semantic errors.

- There are procedures for checking the correctness of the reasoning, giving reports about the reasoning steps that could not be automatically justified.

- A large database of mathematics is built on top of the system and used to check if the established results are new, providing references, etc.

- When the paper is verified, its results are included in such database.

- The paper is automatically translated into natural languages, given to other information retrieval systems such as citation indexes, and printed.

The proposal further suggested the use of classical first-order logic with a rich set of reasoning rules, and to include support for arithmetics and set theory. The language should be rich and closely resemble natural languages, but on the other hand it should not be too complicated to learn, and at the lexical and syntactic level it should resemble programming languages. In particular, the Algol 60 and later the Pascal language and compiler, which appeared in 1970, became sources of inspiration for Mizar and its implementation languages. Various experiments with Pascal program verification were done later with Mizar [Rudnicki and Drabent, 1985].
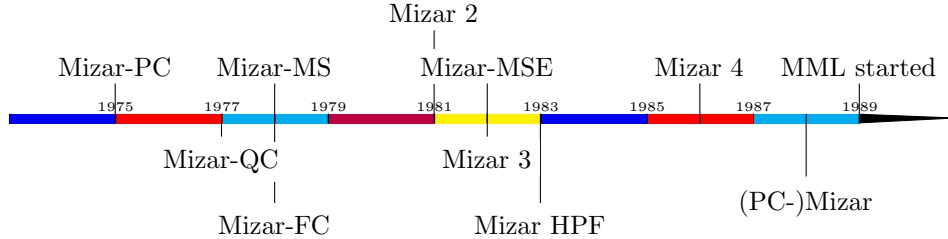
Figure 3: The Mizar timeline

It is likely that in the beginning Trybulec did not know about Automath, LCF, SAM, and other Western efforts, and despite his good contacts with Russian mathematicians and linguists, probably neither about the work of Glushkov and his team in Kiev [Letichevsky *et al.*, 2013] on the Evidence Algorithm and the SAD system. However, the team learned about these projects quite quickly, and apart from citing them, the 1978 paper on Mizar-QC/6000 [Trybulec, 1978] also makes interesting references to Kaluzhnin's 1962 paper on 'information language for mathematics' [Kaluzhnin, 1962], and even earlier (1959, 1961) related papers by Paducheva and others on such languages for geometry. As in some other scientific fields, the wealth of early research done by these Soviet groups has been largely unknown in the western world, see [Lyaletski and Verchinine, 2010; Verchinine *et al.*, 2008] for more details about them.

## 4.1   Development of Mizar

The construction of the Mizar language was started by looking at the paper by H. Patkowska[9] *A homotopy extension theorem for fundamental sequences* [Patkowska, 1969], and trying to express it in the designed language. During the course of the following forty years, a number of versions of Mizar have been developed (see the timeline in Figure 3), starting with bare propositional calculus and rule-based proof checking in Mizar-PC, and ending with today's version of Mizar (Mizar 8 as of 2014) in which a library of 1200 interconnected formal articles is written and verified.

*Mizar-PC 1975-1976*

While a sufficiently expressive mathematical language was a clear target of the Mizar language from the very beginning, the first implementation [Trybulec, 1977]

---

[9]Another PhD student of Karol Borsuk.

of Mizar (written in Algol 1204) was limited to propositional calculus (Mizar-PC).
A number of features particular to Mizar were however introduced already in this
first version, especially the Mizar vocabulary and grammar motivated by Algol
60, and the Mizar suppositional proof style which was later found[10] to correspond
to Jaśkowski-Fitch natural deduction [Jaśkowski, 1934]. An example proof taken
from the June 1975 description of Mizar-PC is as follows:

```
begin
((p ⊃ q) ∧ (q ⊃ r)) ⊃ (p ⊃ r)
proof
      let    A: (p ⊃ q) ∧ (q ⊃ r) ;
      then   B: p ⊃ q ;
             C: q ⊃ r by A ;
      let       p ;
      then      q by B ;
      hence     r by C
end
end
```

The *thesis* (contents, meaning) of the proof in Mizar-PC is constructed from
*assumptions* introduced by the keyword `let` (later changed to `assume`) by placing
an implication after them, and from *conclusions* introduced by keywords `thus` and
`hence` by placing a conjunction after them (with the exception of the last one).
The `by` keyword denotes inference steps where the formula on the left is justified
by the conjunction of formulas whose labels are on the right. The immediately
preceding formula can be used for justification without referring to its label by
using the *linkage* mechanism invoked by keywords `then` for normal inference steps
and `hence` for conclusions. The proof checker verifies that the formula to be proved
is the same as the thesis constructed from the proof, and that all inference steps
are instances of about five hundred inference rules available in the database of im-
plemented schematic proof-checking rules. This rule-based approach was changed
in later Mizar versions to an approach based on model checking (in a general sense,
not in connection with temporal logic model checking). Mizar-PC already allowed
the construction of a so-called *compound statement* (later renamed to *diffuse state-
ment*), i.e., a statement that is constructed implicitly from its suppositional proof
given inside the `begin` ... `end` brackets (later changed to `now` ... `end`) and can be
given a label and referred to in the same way as normal statements. An actual
use of Mizar-PC was for teaching propositional logic in Płock and Warsaw.

### Mizar-QC 1977-1978

Mizar-QC added quantifiers to the language and proof rules for them. An example
proof (taken from [Matuszewski and Rudnicki, 2005]) is:

---

[10]Indeed, the Mizar team found only later that they re-invented Jaśkowski-Fitch proof style.
Andrzej Trybulec was never completely sure if he had not heard about it earlier, for example at
Roman Suszko's seminars.

```
BEGIN
((EX X ST (FOR Y HOLDS P2[X,Y])) > (FOR X HOLDS (EX Y ST P2[Y,X])))
  PROOF
    ASSUME THAT A: (EX X ST (FOR Y HOLDS P2[X,Y]));
    LET X BE ANYTHING;
    CONSIDER Z SUCH THAT C: (FOR Y HOLDS P2[Z,Y]) BY A;
    SET Y = Z;
    THUS D: P2[Y,X] BY C;
  END
END
```

The `FOR` and `EX` keywords started to be used as quantifiers in formulas, and the `LET` statement started to be used for introducing a local constant in the proof corresponding to the universal quantifier of the thesis. The keyword `ASSUME` replaced the use of `LET` for introducing assumptions. The `CONSIDER` statement also introduces a local constant with a proposition justified by an existential statement. The `SET` statement (replaced by `TAKE` later) chooses an object that will correspond to an existential quantifier in the current thesis. While the `LET X BE ANYTHING` statement suggests that a sort/type system was already in place, in the QC version the only allowed sort was just `ANYTHING`.

The `BY` justification proceeds by transforming the set of formulas into a standard form that uses only conjunction, negation and universal quantification and then applying a set of rewrite rules restricted by a bound on a sum of complexity coefficients assigned to the rules. The verifier was implemented in Pascal/6000 for the CDC CYBER-70, and run in a similar way to the Pascal compiler itself, i.e., producing a list of error messages for the lines of the Mizar text. The error messages are inspected by the author, who modifies the text and runs the verifier again. This batch/compilation style of processing of the whole text is also similar to TeX, which was born at the same time. It has become one of the distinctive features of Mizar when compared to interpreter-like ITPs implemented in Lisp and ML.

### Mizar Development in 1978-1988

The development of Mizar-QC was followed by a decade of additions leading to the first version of PC-Mizar[11] in 1989. In 1989 the building of the Mizar Mathematical Library (MML) started, using PC-Mizar.

**Mizar MS (Multi Sorted) (1978)** added predicate definitions and syntax for schemes (such as the Induction and Replacement schemes), i.e., patterns of theorems parameterized by arbitrary predicates and functors. Type declarations were added, the logic became many-sorted, and equality was built in.

**Mizar FC (1978-1979)** added function (usually called functor in Mizar) definitions. The syntax allowed both equational definitions and definitions by conditions that guarantee existence and uniqueness of the function. The `BY` justification

---

[11]PC stands here for Personal Computer.

procedure based on rewrite rules was replaced by a procedure based on 'model checking': the formula to be proved is negated, conjoined with all its premises, and the procedure tries to refute every possible model of this conjunction. This procedure [Wiedijk, 2000] has been subject to a number of additions and experiments throughout the history of Mizar development, focusing on the balance between speed, strength, and obviousness to the human reader [Davis, 1981; Rudnicki, 1987a]. They were mainly concerned with various restricted versions of matching and unification, algorithms such as congruence closure for handling equality efficiently, handling of the type system and various built-in constructs [Naumowicz and Bylinski, 2002; Naumowicz and Bylinski, 2004].

**Mizar-2 (1981)** introduced the `environment` part of an article, at that time containing statements that are checked only syntactically, i.e., without requiring their justification. Later this part evolved into its current form used for importing theorems, definitions and other items from other articles. Type definitions were introduced: types were no longer disjoint sorts, but non empty sets or classes defined by a predicate. This marked the beginning of another large Mizar research topic: its soft type system added on top of the underlying first-order logic. Types in Mizar are neither foundational nor disjoint as in the HOL and Automath families [Wiedijk, 2007]. The best way in which to think of the Mizar types is as a hierarchy of predicates (not just monadic: n-ary predicates result in dependent types), where traversing of this hierarchy – the subtyping, intersection, disjointness relations, etc. – is to a large extent automated and user-programmable, allowing the automation of large parts of the mundane mathematical reasoning.[12] Where such automation fails, the `RECONSIDER` statement can be used, allowing one to change a type of an object explicitly after a justification.

**Mizar-3 and Mizar-4 (1982-1988)** divided the processing into multiple cheaper passes with file-based communication, such as scanning, parsing, type and natural-deduction analysis, and justification checking. The use of special *vocabulary* files for symbols together with allowing infix, prefix, postfix notation and their combinations resulted in greater closeness to mathematical texts. *Reservations* were introduced, for predeclaring variables and their default types. Other changes and extensions included unified syntax for definitions of functors, predicates, attributes and types, keywords for various correctness conditions related to definitions such as `uniqueness` and `existence`, etc. In 1986, Mizar-4 was ported to the IBM PC platform running MS-DOS and renamed to PC-Mizar in 1988.

---

[12]This soft typing system bears some similarities to the sort system implemented in ACL2, and also to the type system used by the early Ontic system. Also the more recent soft (non-foundational) typing mechanisms such as type classes in Isabelle and Coq, and canonical structures in Coq, can be to a large extent seen as driven by the same purpose as types have in Mizar: non-foundational mechanisms for automating the work with hierarchies of defined concepts that can overlap in various ways.

*PC-Mizar and the Mizar Mathematical Library (1988-)*

In 1987-1991 a relatively large quantity of national funding was given to the Mizar team to develop the system and to use it for substantial formalization. The main modification to Mizar to allow that were mechanisms for importing parts of other articles. The official start of building of the Mizar Mathematical Library (MML) dates to January 1, 1989, when three basic articles defining the language and axioms of set theory and arithmetic were submitted. Since then the development of Mizar has been largely driven by the growth of the MML. Apart from the further development of the language and proof-checking mechanisms, a number of tools for proof and library refactoring have been developed. The Library Committee has been established, and gradually more and more work of the core Mizar team shifted to refactoring the library so that duplication is avoided and theories are developed in general and useful form [Rudnicki and Trybulec, 2003]. Perhaps the largest project done in Mizar so far has been the formalization of about 60% of the Compendium of Continuous Lattices [Bancerek and Rudnicki, 2002], which followed the last QED workshop organized by the Mizar team in Warsaw.[13] This effort resulted in about 60 MML articles. One of the main lessons learned (see also the Kreisel's motto above) by the Mizar team from such large projects has been expressed in [Rudnicki and Trybulec, 1999] as follows:

> The MIZAR experience indicates that computerized support for mathematics aiming at the QED goals cannot be designed once and then simply implemented. A system of mechanized support for mathematics is likely to succeed if it has an evolutionary nature. The main components of such a system – the authoring language, the checking software, and the organization of the data base - must evolve as more experience is collected. At this moment it seems difficult to extrapolate the experience with MIZAR to the fully fledged goals of the QED Project. However, the people involved in MIZAR are optimistic.

## 5   SYSTEMS BASED ON POWERFUL AUTOMATION

The LCF approach and the systems based on type theory all tend to emphasize a highly foundational approach to proof, with a (relatively) small proof kernel and a simple axiomatic basis for the mathematics used. While Mizar's software architecture doesn't ostensibly have the same foundational style, in practice its automation is rather simple, arguably an important characteristic since it also enables batch proof script checking to be efficient. Thus, all these systems emphasize simple and secure foundations and try to build up from there. Nowadays LCF-style systems in particular offer quite powerful automated support, but this represents the culmination of decades of sometimes arduous research and development work in foundational implementations of automation. In the first decade

---

[13]http://mizar.org/qed/

of their life, many systems like Coq and Isabelle/HOL that nowadays seem quite powerful only offered rather simple and limited automation, making some of the applications of the time seem even more impressive.

A contrasting approach is to begin with state-of-the-art automation, regardless of its foundational characteristics. Systems with this philosophy were usually intended to be applied immediately to interesting examples, particularly in software verification, and in many cases were intimately connected with custom program verification frameworks. (For example, the GYPSY verification framework [Good *et al.*, 1979] tried to achieve just the kind of effective blend of interaction and automation we are considering, and had a significant impact on the development of proof assistants.) Indeed, in many cases these proof assistants became vehicles for exploring approaches to automation, and thus pioneered many techniques that were later re-implemented in a foundational context by the other systems. Although there are numerous systems worthy of mention — we note in passing EVES/Never [Craigen *et al.*, 1991], KIV [Reif, 1995] and SDVS [Marcus *et al.*, 1985] — we will focus on two major lines of research that we consider to have been the most influential. Interestingly, their overall architectures have relatively little common ground — one emphasizes automation of inductive proofs with iterated waves of simplification by conditional rewriting, the other integration of quantifier-free decision procedures via congruence closure. In their different ways, both have profoundly influenced the field. One might of course characterize them as *automated* provers rather than *interactive* ones, and some of this work has certainly been influential in the field of pure automation. Nevertheless, we consider that they belong primarily to the *interactive* world, because they are systems that are normally used to attack challenging problems via a human process of interaction and lemma generation, even though the automation in the background is unusually powerful. For example, the authors of NQTHM say the following [Boyer and Moore, 1988]:

> In a shallow sense, the theorem prover is fully automatic: the system requires no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the proof attempt. However, in a deeper sense, the theorem prover is interactive: the data base – and hence the user's past actions – influences the behavior of the theorem prover.

## 5.1   NQTHM and ACL2

The story of NQTHM and ACL2 really starts with the fertile collaboration between Robert Boyer and J Strother Moore, both Texans who nevertheless began their work together in 1971 when they were both at the University of Edinburgh. However, we can detect the germ of some of the ideas further back in the work of Boyer's PhD advisor, Woody Bledsoe. Bledsoe was at the time interested in more human-oriented approaches to proof, swimming against the tide of the then-dominant

interest in resolution-like proof search. For example, Bledsoe and Bruell [1974] implemented a theorem prover that was used to explore semi-automated proof, particularly in general topology. In a sense this belongs in our list of pioneering interactive systems because it did provide a rudimentary interactive language for the human to guide the proof, e.g. PUT to explicitly instantiate a quantified variable. The program placed particular emphasis on the systematic use of *rewriting*, using equations to simplify other formulas. Although this also appeared in other contexts under the name of *demodulation* [Wos *et al.*, 1967] or as a special case of superposition in completion [Knuth and Bendix, 1970], and has subsequently developed into a major research area in itself [Baader and Nipkow, 1998], Bledsoe's emphasis was instrumental in establishing rewriting and simplification as a key component of many interactive systems.

Although Boyer and Moore briefly worked together on the popular theme of resolution proving, they soon established their own research agenda: formalizing proofs by *induction*. In a fairly short time they developed their 'Pure LISP theorem prover', which as the name suggests was designed to reason about recursive functions in a subset of pure (functional) Lisp.[14] The prover used some relatively simple but remarkably effective techniques. Most of the interesting functions were defined by primitive recursion of one sort or another, for example over $\mathbb{N}$ by defining $f(n+1)$ in terms of $f(n)$ or over lists by defining $f(\text{CONS } h\ t)$, where CONS is the Lisp list constructor, $h$ the head element and $t$ the tail of remaining elements, in terms of $f(t)$. (In fact, only the list form was primitive in the prover, with natural numbers being represented via lists in zero-successor form.) The pattern of recursive definitions was used to guide the application of induction principles and so produce explicit induction hypotheses. Moreover, the prover was also able to *generalize* the statement to be proved in order better to apply induction — it is a well-known phenomenon that this can make inductive proofs easier because one strengthens the inductive hypothesis that is available. These two distinctive features were at the heart of the prover, but it also benefited from a number of additional techniques like the systematic use of rewriting. Indeed, it was emphasized that proofs should first be attempted using more simple and controllable techniques like rewriting, with induction and generalization only applied if that was not sufficient.

The overall organization of the prover was a prototypical form of what has become known as the 'Boyer-Moore waterfall model'. One imagines conjectures as analogous to water flowing down a waterfall down to a 'pool' below. On their path to the pool below conjectures may be modified (for example by rewriting), they may be proven (in which case they evaporate), they may be refuted (in which case the overall process fails) or they may get split into others. When all the 'easy' methods have been applied, generalization and induction take place, and the new induction hypotheses generated give rise to another waterfall. This process is graphically shown in the traditional picture in Figure 4, although not

---

[14]Note that the prover was not then *implemented* in Lisp, but rather in POP-2, a language developed at Edinburgh by Robin Popplestone and Rod Burstall.
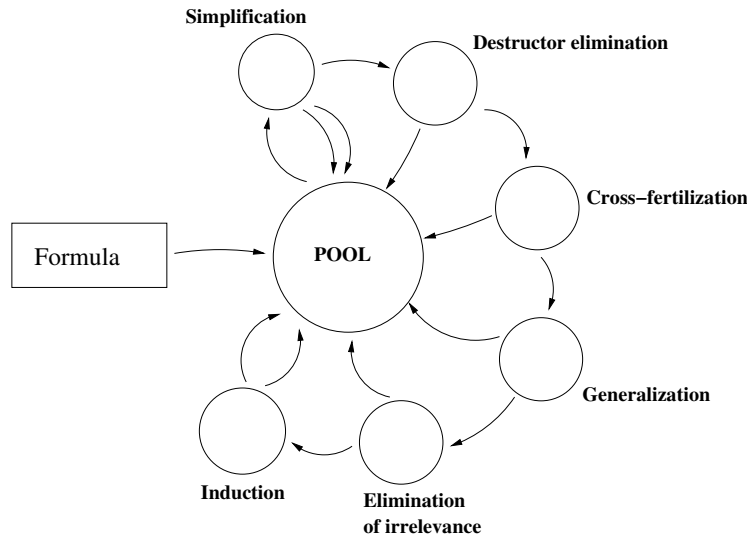
Figure 4: The Boyer-Moore 'Waterfall' model

all the initial steps were present from the beginning.

The next stage in development was a theorem prover concisely known as `THM`, which then evolved via `QTHM` ('quantified `THM`') into `NQTHM` ('new quantified `THM`'). This system was presented in book form [Boyer and Moore, 1979] and brought Boyer and Moore's ideas to a much wider audience as well as encouraging actual use of the system. Note that Boyer and Moore did not at that time use the term `NQTHM` in their own publications, and although it was widely known simply as 'the Boyer-Moore theorem prover', they were too modest to use that term themselves.

`NQTHM` had a number of developments over its predecessors. As the name implies, it supported formulas with (bounded) quantification. It made more extensive and systematic use of simplification, using previously proved lemmas as conditional, contextual rewrite rules. A so-called *shell principle* allowed users to define new data types instead of reducing everything explicitly to lists. The system was able to handle not only primitive recursive definitions and structural inductions [Burstall, 1969] over these types, but also definitions by wellfounded recursion and proofs by wellfounded induction, using an explicit representation of countable ordinals internally. A decision procedure was also added for rational linear arithmetic. All these enhancements made the system much more practical and it was subsequently used for many non-trivial applications, including Hunt's pioneering verification of a microprocessor [Hunt, 1985], Shankar's checking of Gödel's First Incompleteness Theorem [Shankar, 1994], as well as others we will discuss briefly later on. In a significant departure from the entirely automatic Pure Lisp Theorem Prover, NQTHM also supported the provision of hints for guidance and a proof checker

allowing each step of the proof to be specified interactively.

The next step in the evolution of this family, mainly the work of Moore and Matt Kaufmann, was a new system called ACL2, 'A Computational Logic for Applicative Common Lisp' [Kaufmann *et al.*, 2000b]. (Boyer himself helped to establish the project and continued as an important inspiration and source of ideas, but at some point stopped being involved with the actual coding.) Although many aspects of NQTHM including its general style were retained, this system all but eliminated the distinction between its *logic* and its *implementation language* — both are a specific pure subset of Common Lisp. One advantage of such an identification is efficiency. In many of the industrial-scale applications of NQTHM mentioned above, a key requirement is efficient *execution* of functions inside the logic. Instead of the custom symbolic execution framework in NQTHM, ACL2 simply uses direct Lisp execution, generally much faster. This identification of the implementation language and logic also makes possible a more principled way of extending the system with new verified decision procedures, a topic we discuss later. ACL2 has further accelerated and consolidated the use of the family of provers in many applications of practical interest [Kaufmann *et al.*, 2000a].

Besides such concrete applications of their tools, Boyer and Moore's ideas on induction in particular have spawned a large amount of research in automated theorem proving. A more detailed overview of the development we have described from the perspective of the automation of inductive proof is given by Moore and Wirth [2013]. Among the research topics directly inspired by Boyer and Moore's work on induction are Bundy's development of *proof planning* [Bundy *et al.*, 1991] and the associated techniques like *rippling*. Since this is somewhat outside our purview we will not say more about this topic.

## 5.2   EHDM and PVS

There was intense interest in the 1970s and 1980s in the development of frameworks that could perform computer system verification. This was most pronounced, accompanied by substantial funding in the US, for verification of security properties such as isolation in time-sharing operating systems (these were then quite new and this property was a source of some concern), which was quite a stimulus to the development of formal verification and theorem proving in general [MacKenzie, 2001]. Among the other systems developed were AFFIRM, GYPSY [Good *et al.*, 1979], Ina Jo and the Stanford Pascal Verifier. Closely associated with this was the development of combined decision procedures by Nelson and Oppen [1980] and by Shostak [1984]. One other influential framework was HDM, the 'hierarchical development methodology'. The 'hierarchical' aspect meant that it could be used to describe systems at different levels of abstraction where a 'black box' at one level could be broken down into other components at a lower level.

HDM top-level specifications were written in SPECIAL, a 'Specification and Assertion Language'. A security flow analyzer generated verification conditions that were primarily handled using the Boyer-Moore prover discussed previously.

Unfortunately, the SPECIAL language and the Boyer-Moore prover were not designed together, and turned out not to be very smoothly compatible. This meant that a layer of translation needed to be applied, which often rendered the back-end formulas difficult to understand in terms of the original specification. Together with the limited interaction model of the prover, this effectively made it clumsy for users to provide any useful interactive guidance.

Based on the experiences with HDM, a new version EHDM ('Enhanced HDM') was developed starting in 1983, with most of the system designed by Michael Melliar-Smith, John Rushby and Richard Schwarz, while Shostak's decision procedure suite STP was further developed and used as a key component [Melliar-Smith and Rushby, 1985]. Technically this was somewhat successful, introducing many influential ideas such as a system of modules giving parametrization at the theory level (though not fine-grained polymorphism in the HOL sense). It was also used in a number of interesting case studies such as the formalization [Rushby and von Henke, 1991] of an article by Lamport and Melliar-Smith [1985] containing a proof of correctness for a fault-tolerant clock synchronization algorithm, which identified several issues with the informal proof.

Working with Sam Owre and Natarajan Shankar, John Rushby led the project to develop PVS (originally at least standing for 'Prototype Verification System') [Owre *et al.*, 1992] as a new prover for EHDM. Over time it took on a life of its own while EHDM for a variety of technical, pragmatic and social reasons fell into disuse. Among other things, Shostak and Schwarz left to start the database company Paradox, and US Government restrictions made it inordinately difficult for many prospective users to get access to EHDM. Indeed, it was common to hear PVS expanded as 'People's Verification System' to emphasize the more liberal terms on which it could be used.

The goal of PVS was to retain the advantages of EHDM, such as the richly typed logic and the parametrized theories, while addressing some of its weaknesses, making automated proof more powerful (combining Shostak-style decision procedures and effective use of rewriting) and supporting top-down interactive proof via a programmable proof language. At the time there was a widespread belief that one had to make an exclusive choice between a rich logic with weak automation (Automath) or a weak logic with strong automation (NQTHM). One of the notable successes of PVS was in demonstrating convincingly that it was quite feasible to have both.

The PVS logic (or 'specification language') is a strongly typed higher-order logic. It does not have the sophisticated dependent type constructors found in some constructive type theories, but unlike HOL it allows some limited use of dependent types, where types are parametrized by terms. In particular, given any type $\alpha$ and a subset of (or predicate over) the type $\alpha$, there is always a type corresponding to that subset. In other words, PVS supports *predicate subtypes*. In HOL, the simple type system has the appealing property that one can infer the most general types of terms fully automatically. The price paid for the predicate subtypes in PVS is that in general typechecking (that is, deciding whether a term has a specific

type) may involve arbitrarily difficult theorem proving, and the processes of type checking and theorem proving are therefore intimately intertwined. On the other hand, because of the powerful automation, many of the type correctness conditions (TCCs) can still be decided without user interaction.

The PVS proof checker presents the user with a goal-directed view of the proving process, representing goals using multiple-conclusion sequents. Many basic commands for decomposing and simplifying goals are as in many other interactive systems like Coq or HOL. But PVS also features powerful and tightly integrated decision procedures that are able to handle many routine goals automatically in response to a simple invocation of the `simplify` command. Although PVS does not make the full implementation language available for programming proof procedures, there is a special Lisp-like language that can be used to link proof commands together into custom *strategies*.

## 6    RESEARCH TOPICS IN INTERACTIVE THEOREM PROVING

Having seen some of the main systems and the ideas they introduced in foundations, software architecture, proof language etc., let us step back and reflect on some of the interesting sources of diversity and examine some of the research topics that naturally preoccupy researchers in the field.

### 6.1    Foundations

For those with only a vague interest in foundations who somehow had the idea that ZF set theory was the standard foundation for mathematics, the diversity, not to say Balkanization, of theorem provers according to foundational system may come as a surprise. We have seen at least the following as foundations even in the relatively few systems we've surveyed here:

- Quantifier-free logic with induction (NQTHM, ACL2)

- Classical higher-order logic (HOLs, PVS)

- Constructive type theory (Coq, NuPRL)

- First-order set theory (Mizar, EVES, Isabelle/ZF)

- Logics of partial terms (LCF, IMPS, Isabelle/HOLCF)

Some of this diversity arises because of specific philosophical positions among the systems' developers regarding the foundations of mathematics. For example, modern mathematicians (for the most part) use nonconstructive existence proofs without a second thought, and this style fits very naturally into the framework of classical first-order set theory. Yet ever since Brouwer's passionate advocacy [van Dalen, 1981] there has been a distinct school of *intuitionistic* or *constructive*

mathematics [Beeson, 1984; Bishop and Bridges, 1985]. While Brouwer had an al-most visceral distaste for formal logic, Heyting introduced an intuitionistic version of logic, and although there are workable intuitionistic versions of formal set the-ory, the type-theoretic frameworks exploiting the Curry-Howard correspondence between propositions and types, such as Martin-Löf's type theory [Martin-Löf, 1984], are arguably the most elegant intuitionistic formal systems, and it is these that have inspired Coq, NuPRL and many other provers.

Other motivations for particular foundational schemes are pragmatic. For ex-ample, HOL's simple type theory pushes a lot of basic domain reasoning into automatic typechecking, simplifying the task of producing a reasonable level of mechanical support, while the very close similarity with the type system of the ML programming language makes it feel natural to a lot of computer scientists. The quantifier-free logic of NQTHM may seem impoverished, but the very restric-tiveness makes it easier to provide powerful automation, especially of inductive proof, and forces definitions to be suitable for actual execution.

Indeed, a little reflection shows that the distinction between philosophical and pragmatic motivations is not clear-cut. While one will not find any philosophical claims about constructivism associated with NQTHM and ACL2, it is a fact that the logic is even more clearly constructive than intuitionistic type theories.[15] De-spite the Lisp-like syntax, it is conceptually close to primitive recursive arithmetic (PRA) [Goodstein, 1957]. And many people find intuitionistic logic appealing not so much because of philosophical positions on the foundations of mathematics but because at least in principle, the Curry-Howard correspondence has a more prag-matic side: one can consider a proof in a constructive system actually to be a program [Bates and Constable, 1985].

The language we use can often significantly influence our thoughts, whether it be natural language, mathematical notation or a programming language [Iverson, 1980]. Similarly, the chosen foundations can influence mathematical formalization either for good or ill, unifying and simplifying it or twisting it out of shape. Indeed, it can even influence the kinds of proofs we may even *try* to formalize. For example, ACL2's lack of traditional quantifiers makes it unappealing to formalize traditional epsilon-delta proofs in real analysis, yet it seems ideally suited to the reasoning in *nonstandard* analysis, an idea that has been extensively developed by Gamboa [1999]; for another development of this topic in Isabelle/HOL see [Fleuriot, 2001].

In particular, the value of *types* is somewhat controversial. Both types [White-head and Russell, 1910; Ramsey, 1926; Church, 1940] and the axiomatic approach to set theory culminating in modern systems like ZF, NBG etc., originated in attempts to resolve the paradoxes of naive set theory, and may be seen as two competing approaches. Set theory has long been regarded as the standard foun-dation, but it seems that at least when working in concrete domains, most math-ematicians do respect natural type distinctions (points versus lines, real numbers versus sets of real numbers). Even simpler type systems like that of HOL make

---

[15]At least in its typical use — we neglect here the built-in interpreter axiomatized in NQTHM, which could be used to prove nonconstructive results [Kunen, 1998].

a lot of formalizations very convenient, keeping track of domain conditions and compatibility automatically and catching blunders at an early stage.

However, for some formalizations the type system ceases to help and becomes an obstacle. This seems to occur particularly in traditional abstract algebra where constructions are sometimes presented in a very type-free way. For example, a typical "construction" of the algebraic closure of a field proceeds by showing that one can extend a given field $F$ with a root $a$ of a polynomial $p \in F[x]$, and then roughly speaking, iterating that construction transfinitely (this is more typically done via Zorn's Lemma or some such maximal principle, but one can consider it as a transfinite recursion). Yet the usual way of adding a single root takes one from the field $F$ to a equivalence class of polynomials over $F$ (its quotient by the ideal generated by $p$). When implemented straightforwardly this might lie two levels above $F$ itself: if we think of elements of $F$ as belonging to a type $\alpha$ then polynomials over $F$ might be functions $\mathbb{N} \to F$ (albeit with finite support) and then equivalence classes represented as Boolean functions over that type, so we have moved to $(\mathbb{N} \to F) \to 2$. And that whole process needs to be iterated transfinitely. Of course one *can* use cardinality arguments to choose some sufficiently large type once and for all and map everything back into that type at each stage. One may even argue that this gives a more refined theorem with information about the cardinality of the algebraic closure, but the value of being *forced* to do so by the foundation is at best questionable. Another limitation of the simple HOL type system is that there is no explicit quantifier over polymorphic type variables, which can make many standard results like completeness theorems and universal properties awkward to express, though there are extensions with varying degrees of generality that fix this issue [Melham, 1992; Voelker, 2007; Homeier, 2009]. Inflexibilities of these kinds certainly arise in simple type theories, and it is not even clear that more flexible *dependent* type theories (where types can be parametrized by terms) are immune. For example, in one of the most impressive formalization efforts to date [Gonthier *et al.*, 2013] the entire group theory framework is developed in terms of subsets of a single universe group, apparently to avoid the complications from groups with general and possibly heterogeneous types.

Even if one considers types a profoundly useful concept, it does not follow of course that they need to be hardwired into the logic. Starting from a type-free foundation, it is perfectly possible to build *soft* types as a derived concept on top, and this is effectively what Mizar does, arguably giving a good combination of flexibility, convenience and simplicity [Wiedijk, 2007]. In this sense, types can be considered just as sets or something very similar (in general they can be proper classes in Mizar). On the other hand, some recent developments in foundations known as *homotopy type theory* or *univalent foundations* give a distinctive role to types, treating equality on types according to a homotopic interpretation that may help to formalize some everyday intuitions about identifying isomorphic objects.

Another interesting difference between the various systems (or at least the way mathematics is usually formalized in them) is the treatment of undefined terms

like $0^{-1}$ that arise from the application of functions outside their domain. In informal mathematics we often filter out such questions subconsciously, but the exact interpretation of such undefinedness can be critical to the assertion being made. We can identify three main approaches taken in interactive provers:

- Totalization (usual in HOL) — functions are treated as total, either giving them an arbitrary value outside their domain or choosing one that is particularly convenient for making handy theorems work in the degenerate cases too. For example, setting $0^{-1} = 0$ [Harrison, 1998] looks bizarre at first sight, but it lets us employ natural rewrite principles like $(x^{-1})^{-1} = x$, $-x^{-1} = (-x)^{-1}$, $(xy)^{-1} = x^{-1}y^{-1}$ and $x^{-1} \geq 0 \Leftrightarrow x \geq 0$ without any special treatment of the zero case. (There is actually an algebraic theory of *meadows*, effectively fields with this totalization [Bergstra *et al.*, 2007].) While simple, it has the disadvantage that equations like $f(x) = y$ do not carry with them the information that $f$ is actually defined at point $x$, arguably a contrast with informal usage, so one must add additional conditions or use relational reformulations.

- Type restrictions (usual in PVS) — the domain restrictions in the partial functions are implemented via the type system, for example giving the inverse operation a type $: \mathbb{R}' \to \mathbb{R}$ where $\mathbb{R}'$ corresponds to $\mathbb{R} - \{0\}$. This seems quite natural in some ways, but it can mean that types become very intricate for complicated theorems. It can also mean that the precise meaning of formulas like $\forall x \in \mathbb{R}.\ \tan(x) = 0 \Rightarrow \exists n \in \mathbb{Z}.x = n\pi$, or even whether such a formula is acceptable or meaningful, can depend on quite small details of how the typechecking and basic logic interact.

- Logics of partial terms (as supported by IMPS [Farmer *et al.*, 1990]) — here there is a first-class notion of 'defined' and 'undefined' in the foundational system itself. Note that while it is possible to make the logic itself 3-valued so there is also an 'undefined' proposition [Barringer *et al.*, 1984], this is not necessary and many systems allow partial terms while maintaining bivalence. One can have different variants of the equality relation such as 'either both sides are undefined or both are defined and equal'. While apparently complicated and apt to throw up additional proof obligations, this sort of logical system and interpretation of the equality relation arguably gives the most faithful analysis of informal mathematics.

## 6.2 Proof language

As we noted at the beginning, one significant design decision in interactive theorem proving is choosing a language in which a human can communicate a proof outline to the machine. From the point of view of the user, the most natural desideratum might be that the machine should understand a proof written in much the same way as a traditional one from a paper or textbook. Even accepting that this is

indeed desirable, there are two problems in realizing it: getting the computer to understand the linguistic structure of the text, and having the computer fill in the gaps that human mathematicians consider as obvious. Recently there has been some progress in elucidating the structure of traditional mathematical texts such that a computer could unravel much of it algorithmically [Ganesalingam, 2013], but we are still some way from having computers routinely understand arbitrary mathematical documents. And even quite intelligent human readers sometimes have difficulty in filling in the gaps in mathematical proofs. Subjectively, one can sometimes argue that such gaps amount to errors of omission where the author did not properly appreciate some of the difficulties, even if the final conclusion is indeed accurate. All in all, we are some way from the ideal of accepting existing documents, if ideal it is. The more hawkish might argue that formalization presents an excellent opportunity to present proofs in a more precise, unambiguous and systematic — one might almost say machine-like — way [Dijkstra and Scholten, 1990].

In current practice, the proof languages supported by different theorem proving systems differ in a variety of ways. One interesting dichotomy is between *procedural* and *declarative* proof styles [Harrison, 1996c]. This terminology, close to its established meaning in the world of programming languages, was suggested by Mike Gordon. Roughly, a declarative proof outlines *what* is to be proved, for example a series of intermediate assertions that act as waystations between the assumptions and conclusions. By contrast, a *procedural* proof explicitly states *how* to perform the proofs ('rewrite the second term with lemma 7 ...'), and some procedural theorem provers such as those in the LCF tradition use a full programming language to choreograph the proof process. To exemplify procedural proof, here is a HOL Light proof of the core lemma in the theorem that $\sqrt{2}$ is irrational, as given in [Wiedijk, 2006]. It contains a sequence of procedural steps and even for the author, it is not easy to understand what they all do without stepping through them in the system.

```
let NSQRT_2 = prove
 (`!p q. p * p = 2 * q * q ==> q = 0`,
  MATCH_MP_TAC num_WF THEN REWRITE_TAC[RIGHT_IMP_FORALL_THM] THEN
  REPEAT STRIP_TAC THEN FIRST_ASSUM(MP_TAC o AP_TERM `EVEN`) THEN
  REWRITE_TAC[EVEN_MULT; ARITH] THEN REWRITE_TAC[EVEN_EXISTS] THEN
  DISCH_THEN(X_CHOOSE_THEN `m:num` SUBST_ALL_TAC) THEN
  FIRST_X_ASSUM(MP_TAC o SPECL [`q:num`; `m:num`]) THEN
  POP_ASSUM MP_TAC THEN CONV_TAC SOS_RULE);;
```

By contrast, consider the following *declarative* proof using the Mizar mode for HOL Light [Harrison, 1996b], which is a substantial fragment of the proof of the Knaster-Tarski fixed point theorem [Knaster, 1927; Tarski, 1955].[16] There is not a single procedural step, merely structuring commands like variable introduction

---

[16]See http://code.google.com/p/hol-light/source/browse/trunk/Examples/mizar.ml.

('`let a`') together with a sequence of intermediate assertions and the premises
from which they are supposed (somehow) to follow:

```
consider a such that
    lub: (!x. x IN Y ==> a <= x) /\
         (!a'. (!x. x IN Y ==> a' <= x) ==> a' <= a)
    by least_upper_bound;
take a;
!b. b IN Y ==> f a <= b
proof
  let b be A;
  assume b_in_Y: b IN Y;
  then L0: f b <= b by Y_thm;
  a <= b by b_in_Y, lub;
  so f a <= f b by monotonicity;
  hence f a <= b by L0, transitivity;
end;
so Part1: f(a) <= a by lub;
so f(f(a)) <= f(a) by monotonicity;
so f(a) IN Y by Y_thm;
so a <= f(a) by lub;
hence thesis by Part1, antisymmetry;
```

A more declarative style can offer significant advantages. Declarative proof
scripts are generally easier to write and understand independent of the prover,
whereas one usually needs to develop, or even understand, a procedural script by
running it step-by-step through the system to see the intermediate results. (By
analogy, consider replaying a chess game from a newspaper chess column given
just the sequence of moves with no diagrams of the board state.) Because they
lack explicit inference instructions, declarative proofs are also usually less tied to
the details of the prover's implementation and so are likely to be more readable
for non-experts and portable in a general sense. On the other hand, declarative
proofs can be clumsy and verbose when they involve large and complex terms or
are naturally expressed as a simple process of transformation, something that is
particularly common in verification applications. As for making modifications to
an existing proof, which is often an important activity [Curzon, 1995], there are
pluses and minuses on both sides. In a declarative proof, key variable introductions
and intermediate statements are made explicit instead of arising as a side-effect
of some intermediate step, and are more amenable to systematic changes. Some
existing experience [Chen, 1992; Gonthier, 1996] supports the declarative style as
yielding proofs that are easier to maintain. On the other hand, some procedural
proofs can be surprisingly insensitive to small changes and may even work with
no changes at all. These questions are surveyed in a bit more detail in [Harrison,
1996c].

At one extreme, the most declarative proof language is the one supported by a completely *automated* theorem prover: just state the theorem to be proved without any hint as to how to prove it! Of course, the whole point of interactive theorem proving is to allow user guidance, but some theorem provers that we have classified as interactive do indeed use this approach, the only difference being that the user must identify a series of intermediate lemmas that can be stepping-stones to the main result such that the gap between successive steps is within the scope of the automation. In particular, proof scripts in NQTHM and ACL2 are usually just sequences of lemmas without much procedural information. The primary additional data is that for each lemma, the user may add *hints* about how the automation should use that lemma in the future (rewrite, elimination, generalization or induction lemma).

Mainstream mathematics normally uses much richer quantifier structures than are possible (or at least conveniently usable) in NQTHM/ACL2. This can make it less appealing to use a simple series of toplevel lemmas as a proof outline, because one often wants to structure the proof according to the introduction and elimination of variables and localize reasoning to some assumed environment. Several theorem proving systems and program verifiers adopted a structured style of proof close to natural deduction, organized around variable elimination and introduction rules. Consider for example the following sample proof from NuPRL's precursor, the program verifier PL/CV [Constable and O'Donnell, 1978]:

```
LEMMA_T:
/# TRANSITIVITY OF DIVIDES #/
ALL (A, B, C) FIXED . (DIV(A,B) & DIV(B, C) => DIV(A, C))
BY INTRO,
PROOF;
     CHOOSE M1 FIXED WHERE M1*A = B;
     CHOOSE M2 FIXED WHERE M2*B = C;
     M2*(M1*A) = C;  /# BY SUBSTITUTION #/
     (M2*M1)*A = C;  /# BY ASSOCIATIVITY OF * #/
     DIV(A,C) BY SOMIN, M2*M1;
QED;
```

Exactly such a *structured* approach was used by Mizar, inspired not only by the Jaśkowski-Fitch approach to natural deduction [Jaśkowski, 1934; Fitch, 1952] but also by the block structure of the Pascal programming language [Jensen and Wirth, 1974]. As the examples given above show, the proof is structured around a 'skeleton' indicating the introduction and elimination of variables and assumptions, and then the intermediate steps are just stated as assertions, without any procedural information. In this sense Mizar's proof style successfully combined *structure* and a *declarative* style. (By contrast, the PL/CV example does have some explicit inference rules and instantiation, although they are only used in one line of the above example.)

The relative readability of Mizar proofs contrasts quite starkly with the obscurity of many procedural languages, particularly the traditional tactic scripts in LCF-style systems with their invocations of mysterious and arcane transformations, once parodied by Conor McBride as `EAR_OF_BAT_TAC`. Because of this, there has been considerable interest in supporting more declarative proof styles in other systems. This started with the 'Mizar mode for HOL' [Harrison, 1996b] and was followed by several other declarative languages for other systems [Syme, 1997; Wenzel, 1999; Zammit, 1999]. In particular the Isar language for Isabelle, already discussed above, has now largely superseded the use of the traditional ML level, although the Isar proofs are usually a mix of a structured skeleton with intermediate proof steps, making them highly structured but only partly declarative. Indeed, it is natural to desire a smooth combination of both procedural and declarative approaches. A number of experiments in HOL Light have resulted in quite usable systems [Wiedijk, 2001; Wiedijk, 2012b] that have been subsequently applied and refined by Bill Richter.[17]

The ability to program other proof styles like 'Mizar mode' in LCF systems shows that although the traditional style of such systems is highly procedural, one can layer virtually any proof style on top. This applies even if one wants a purely *procedural* language that differs from the full implementation language. Among procedural systems, the traditional LCF approach where the implementation language read-eval-print loop is the primary interface stands at one end of a spectrum, with very simple macro languages at the other. A high level of programmability can be extremely valuable, for example in allowing simple custom inference rules to be implemented, or a slew of related subgoals to be disposed of by a single script. In order to maintain programmability and flexibility while keeping the language somewhat more constrained — for example to close off obscure language loopholes, enforce a more uniform style, or allow more straightforward parsing and processing by other tools — there is a good case for adopting some intermediate position on this spectrum. The main Ltac language of Coq is a good example [Delahaye, 2000], as is the more recent SSReflect language.

Even if one has a declarative style as the intention, it can be difficult to avoid explicit invocations of proof commands unless there is a kind of default automation that is rather powerful. Mizar's built-in checker has a deliberately constrained first-order prover, though it also includes other features like congruence closure, representing a certain point of view on what constitutes an obvious inference [Rudnicki, 1987b]. This prover is simple and efficient, but quite limited compared to state-of-the-art automated systems [Wiedijk, 2000]. Thus, to support a suitably high-level declarative proof style, automation is an important component, and we turn to this question next.

---

[17]See                        `http://code.google.com/p/hol-light/source/browse/trunk/`
`RichterHilbertAxiomGeometry/HilbertAxiom_read.ml`

## 6.3   Automation and certification

The convenience and practicality of interactive proof greatly increases if the system is able to automate as much routine work as possible. Given the extensive development of automated methods for propositional logic, first-order logic, arithmetic etc. and special tools like model checkers and computer algebra systems, it is natural to want to use some of the same ideas in interactive tools, and perhaps even use automated systems themselves as subcomponents.

It is usually not too difficult to identify appropriate subsets of the logic supported by an interactive prover with those automated by special tools like SAT solvers. Once this is done, it is usually feasible to to implement similar algorithms oneself, and often much easier and faster to use highly engineered off-the-shelf tools themselves as subroutines. For example [Seger and Joyce, 1991] and [Rajan *et al.*, 1995] describe effective combinations of theorem provers and model checkers. However, even if one makes the effort to maintain full control over the implementation, it can be difficult to get such complex algorithms right, so there is a danger of compromising the high standards of rigour. Milner [MacKenzie, 2001] compared using an unverified decision procedure to 'selling your soul to the Devil [...] you get this enormous power [...but] you've lost proof in some sense'. Since many share these qualms, though they might not express them so forthrightly, there has been considerable interest in the implementation of automation in a highly reliable way. We can identify three main approaches:

- Fully-expansive algorithm design: rewrite the algorithm to perform inference at each step and carry through formally proven theorems.

- Reflection: use the theorem proving tool to prove the correctness of the new proof procedure's implementation and only then include it in the trusted code base.

- Certification: organize the algorithm so that it produces not only a result but some kind of 'proof' or 'certificate' that can be independently checked.

The first approach, fully expansive recoding of the algorithm, is the traditional LCF answer to most such problems. A long-established methodology — see [Melham, 1989] for early non-trivial examples — means that it is often a fairly routine matter to translate many symbolic algorithms into inference-producing versions, for example replacing each ad-hoc term transformation with a *conversion* as outlined above. A good example of such methods at work is the implementation by Slind [1991] of Knuth-Bendix completion in HOL. Other methods are even easier to integrate into an LCF-style system because they mainly involve heuristics and other techniques for putting together existing rules or tactics — this applies for example to Boyer-Moore automation of induction and related techniques like proof planning, which have indeed been implemented in LCF-style provers [Boulton, 1992; Dixon, 2005; Papapanagiotou, 2007].
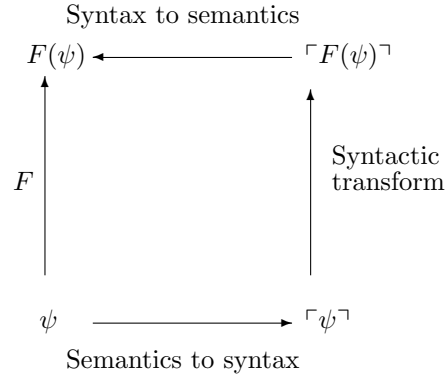
Syntax to semantics

$$F(\psi) \longleftarrow \ulcorner F(\psi) \urcorner$$

$F$

Syntactic
transform

$$\psi \longrightarrow \ulcorner \psi \urcorner$$

Semantics to syntax

Figure 5: Using 'reflection' inside the logic

One reason why this fully-expansive inference-producing style is more practical than might appear at first sight is that to implement many derived rules, all the complicated reasoning can be embedded in a single 'proforma theorem'. To take a trivial example, the fact that from $p \wedge q$ we can deduce $p$ can be embedded in the theorem $\vdash p \wedge q \Rightarrow p$. Now in any particular instance $\vdash a \wedge b$, we need only instantiate this theorem, to get $\vdash a \wedge b \Rightarrow a$, and perform Modus Ponens and get $\vdash a$. Of course, this is a trivial example but one can embed much more interesting reasoning in a single theorem that later merely needs to be instantiated.

In more elaborate cases, it may even be worth defining some special syntactic forms inside the logic so that the workings of the algorithm can be expressed more directly via general proforma theorems, at the cost of some folding and unfolding of equivalent forms to apply it to concrete cases. (This is setting up a *deep embedding* of a sublogic, in the terminology we discuss near the end of this paper.) For example, in the implementation of Cooper's algorithm [Cooper, 1972] for integer quantifier elimination in HOL Light, a kind of 'shadow syntax' is defined for a class of quantifier-free first-order formulas inside the logic, with their semantics defined using an interpretation function `interp`. The key syntactic transforms involved can then be expressed as a simple equivalence in interpretations of the shadow syntax. Now, in order to eliminate a quantifier in HOL from an expression $\exists x.P[x]$, one first rewrites backwards with the definition of `interp` to map it into a formula in the canonical form $\exists x.\mathtt{interp}\ x\ p$, appeals to the general theorem to transform it into a quantifier-free equivalent, then rewrites forward with the definition of `interp` to eliminate the internal syntax. In general, we can justify a transformation of some formula $\psi$ to another one $F(\psi)$ via a formalized transformation on the syntactic form $\ulcorner \psi \urcorner$ (see Figure 5).

Taking the 'proforma theorem' approach to its logical extreme, one can formally define the entire workings of an algorithm as operations on the embedded 'shadow syntax' inside the logic. Most algorithms that can be written in a functional language can also be developed entirely inside the logic in this way. Of course

the process of 'executing' inside the logic involves inference steps, meaning that is likely to be substantially slower, perhaps only by a constant factor but most likely a substantial one [McLaughlin and Harrison, 2005]. However, Coq features a highly efficient reduction mechanism that comes close in performance to a native functional language, so provided the shadow syntax lives in the right subset, it can be executed quite fast. On the other side of the coin, conventional inference in Coq is relatively slow and memory-hungry because it actually creates explicit proof objects. All this means that the benefits of the 'shadow syntax' approach, commonly called 'reflection' in the Coq world, are much more compelling even for relatively simple algorithms.

More generally, we use 'reflection' to refer to any scheme where one is basically 'verifying code and executing it'. In most established examples like the pioneering use of *metafunctions* in NQTHM [Boyer and Moore, 1981], the code is actually extracted to a conventional programming language rather than, as in Coq, executed inside the logical kernel. (On the other hand for ACL2 this distinction essentially does not exist.) A nexus of distinct but related ideas often going under the name of 'reflection' have been tried in theorem proving systems like FOL/GETFOL [Weyhrauch, 1980; Weyhrauch and Talcott, 1994; Weyhrauch, 1982] and NuPRL [Knoblock and Constable, 1986; Allen *et al.*, 1990; Howe, 1992]. These ideas are surveyed in more detail in [Giunchiglia and Smaill, 1989; Harrison, 1995b]. Significant recent examples of some classical logical decision procedures implemented using reflection include [Chaieb and Nipkow, 2008] and [Cohen and Mahboubi, 2010].

In many important cases, there is a simpler approach to implementing correct proof procedures: have the proof procedure produce some kind of *certificate* that can be checked relatively simply and efficiently by proof. The general merit of this kind of approach — not limited to foundational theorem proving — was emphasized by Blum [1993]. He suggests that in many situations, checking results may be more practical and effective than verifying code. A very simple case is verifying that a number is *not* prime, which can be done easily, even by logical inference, given a factorization as the certificate, even if *finding* that certificate is difficult. An early example of the certification approach in our context is the linkup between the HOL theorem prover and Maple computer algebra system reported by Harrison and Théry [1998]. Here Maple is used to perform polynomial factorization and transcendental function integration. In each case the checking process (respectively multiplying polynomials and taking derivatives) is substantially easier than the process of finding the certificate. And note that by taking this path we are often able to inherit the extensive implementation and optimization work done by others on the external tools without any additional work on our own part.

In the cases considered so far (integer and polynomial factorization and antiderivatives) the certificate is just what one might intuitively call 'the answer'. Sometimes, though, it is useful to have a more information in the certificate in order to verify it without an expensive and complicated search process. This applies in particular to various decision procedures for quantifier-free first-order

arithmetic theories. These can provide relatively compact certificates that can be checked reasonably easily. For example, an invalid conjunction of linear arithmetic constraints can be checked by providing a linear combination that sums to give a trivial inequality like $1 < 0$; the existence of such linear combinations is essentially the content of Farkas's lemma [Webster, 1995]. This method was first used in a formal context by Boulton [1993] in a linear arithmetic procedure for HOL, using his own implementation of the certificate-finding, and by Necula (in connection with proof-carrying code), using an off-the-shelf linear programming package to find the certificate [Necula and Lee, 2000]. It has reached its apotheosis in the work of Alexey Solovyev [Solovyev and Hales, 2011], who has checked the very large linear programs in the Flyspeck project (discussed in more detail later) inside HOL Light using such certification, remarkably efficiently. Analogous techniques based on the Hilbert Nullstellensatz can handle the universal theory of integral domains or fields (see [Harrison, 2009b] for a detailed discussion), and this first done in our context in [Harrison, 2001]. A similar but more complicated technique works for real-closed fields like the real numbers using certificates involving sums of squares that can be found using semidefinite programming tools [Parrilo, 2003], and this has also been exploited for formal proof [Harrison, 2007]. This is perhaps best illustrated by a specific example. Suppose we want to show that if a quadratic equation has a (real) solution, its discriminant is nonnegative:

$$\forall a\ b\ c\ x.\ \ ax^2 + bx + c = 0 \Rightarrow b^2 - 4ac \geq 0$$

A suitable certificate is $b^2 - 4ac = (2ax + b)^2 - 4a(ax^2 + bx + c)$. Since the first term on the right is a square, and the second is zero by hypothesis, it is clear that the LHS is nonnegative. Almost all the conceptual/implementation difficulty and computational cost is in coming up with the right algebraic rearrangement; checking this and the consequent reasoning is then easy.

In other cases like first-order theorem proving, successful runs of the prover may perform a lot of search, but usually find a relatively short proof. Provided the automated system is indeed able to produce the final proof, it is in principle a fairly straightforward matter to check it by inference in the ITP system. This approach has been used for a long time to incorporate first-order proof methods into LCF-style provers [Kumar et al., 1991], first of all with custom code to find the proof and later with off-the-shelf external provers [Hurd, 1999]. This brings with it a host of apparently minor but sometimes quite knotty problems. First of all, one is usually interested in using provers for pure first-order logic to tackle problems in ITPs with richer logics like polymorphically typed higher-order logic, which raises interesting choices about how to relate the two worlds [Harrison, 1996b; Dahn and Wernhard, 1997; Hurd, 2003; Meng and Paulson, 2006; Blanchette et al., 2013]. Many off-the-shelf first-order provers are not particularly good at returning proofs, sometimes necessitating quite elaborate programming to reconstruct them. And typically, off-the-shelf systems have a bias towards relatively small synthetic problems and can be swamped if they are presented with a large database of lemmas to use, making premise selection an issue.

The first convincing arrangement that successfully addresses all these problems is the 'Sledgehammer' framework for Isabelle developed by Paulson [Paulson and Blanchette, 2010]. For many Isabelle users this has led to a distinct change in their approach to theorem proving, tending to let the automation plug the gaps eagerly, even working in the background while the user thinks. Traditionally, LCF systems have tended to encourage tight and careful control over the proof process, but as a result of the success of Sledgehammer, there is now a trend among Isabelle users at least back to the earlier automated approaches — for example, Ontic [McAllester, 1989] already anticipated the effective automated use of a large background database. Related 'hammer frameworks' for HOL Light and Mizar offering a variety of premise selection schemes [Kühlwein *et al.*, 2012] based on machine learning and allowing use of a central server over the network, are described by Kaliszyk and Urban [2014a] and Urban *et al.* [2013].

Quite generally, the current trend in many Artificial Intelligence domains (for example, translation between languages) is to use general machine learning on huge datasets [Shawe-Taylor and Cristianini, 2004] in preference to intricately hand-crafted algorithms. In the light of this the relative lack of such techniques in the world of theorem proving seems surprising. Apart from the premise selection task, Urban, Schulz, Bridge, Kaliszyk, Kühlwein and others have recently used machine learning for selecting suitable theorem-proving strategies [Kühlwein *et al.*, 2013; Schulz, 2002; Bridge *et al.*, 2014] and automated construction of such strategies [Urban, 2014] over large sets of related problems, mining the large inference graphs of ITP systems for suitable lemmas and conjectures [Kaliszyk and Urban, 2014b; Denzinger and Schulz, 1996], and fine-grained guidance of the automated theorem provers [Urban *et al.*, 2011; Schulz, 2000]. Combinations of such machine learning systems with automated theorem provers on the large ITP libraries may result in AI-style feedback loops, where the learning and the proving components gradually improve from the data supplied by the other component [Urban *et al.*, 2008]. Note that since these methods usually contribute high-level guidance such as a set of lemmas, it's generally straightforward to integrate them into fully-expansive systems without any issues. And they greatly benefit from training on the large libraries of formal mathematics that are associated with interactive systems. Thus, somewhat paradoxically, machine learning impinges at least as effectively on *interactive* theorem proving as traditional automated proving.

As with first-order provers, many SAT solvers are capable of emitting proofs (usually recast as resolution proofs even if that doesn't reflect how they were found), which have applications not only in proof-checking but for invariant generation via interpolation [McMillan, 2003]. Unlike FOL proofs, SAT proofs tend to be relatively large, but nevertheless it has turned out to be possible to check them in a fully expansive way in a time comparable to the time used to generate them [Weber and Amjad, 2009]. Combined decision procedure suites can also be checked in a fully-expansive way; this was first done by Boulton for his own implementation [Boulton, 1993] and then for an off-the-shelf SMT system by Mclaughlin *et al.* [2005]. However, the case of quantified Boolean formulas (QBF) appears

to be a less favorable one where the proof reconstruction in a formal prover can be considerably more time-consuming than the process of finding it [Weber, 2010; Kunčar, 2011; Kumar and Weber, 2011].

Finally, we noted above the relative triviality of proving non-primality. The dual problem of proving primality doesn't admit quite such a straightforward certification, but there are known certificates of primality that are usable for this purpose, the first to be presented being due to Pratt [Pratt, 1975]. At the time, the key interest of Pratt's observation was to establish that primality testing is in the complexity classes NP *and* co-NP. Though primality testing was much later established to be in P [Agrawal *et al.*, 2004], Pratt's result retains its interest as an effective way of certifying primality. Caprotti and Oostdijk [2001] first implemented primality proving in Coq using an optimized Pocklington variant of Pratt certificates. Proving primality of $p$ in this way requires at least a partial prime factorization of $p - 1$ and hence recursive proofs of primality of those factors; these certificates can be generated without proof using sophisticated off-the-shelf factorization software and checked by applying Pocklington's theorem. Much subsequent work has optimized the implementation in Coq and extended this to more sophisticated primality-proving methods based on elliptic curves [Théry and Hanrot, 2007]. While this has mainly been pursued for its pure intellectual interest and as a motivation for optimizing basic operations inside Coq [Grégoire *et al.*, 2006], proving the primality of specific numbers can actually have genuine applications in verification [Harrison, 2003].

## 6.4 Sharing

This chapter has repeatedly hinted at the variety of different proof assistants, often with radically different foundations. On the positive side this means that there is diversity of experience in using various systems in applications, which helps us to better understand the strengths and weaknesses of approaches. However, given the overhead in learning to use even one of these systems effectively, there is a tendency for researchers to get trammelled into using just one system, as a result of which essentially similar work gets duplicated many times. It would obviously be appealing to have some practical way of sharing work among different systems.

One approach, arguably the simplest, is to import *theorems* from one system to another without in any sense attempting to check their correctness, though perhaps tagging them in some way to indicate their provenance. If we assume that we are importing from a similarly reliable tool, then such results should have been checked at a similar level of rigour. The primary difficulty is ensuring a meaningful semantic match between the two systems, i.e. making sure that precise definitions, types, treatment of partial functions etc. in the source system are compatible with the target. This tends to work better when the target system's logic is at least as rich in some sense, so that there are indeed faithful models of the source results on the target side, whatever constructs get used there. The first such example was the pioneering work of Felty and Howe [Felty and Howe, 1997] on importing

mathematics from HOL to NuPRL. Other more recent examples have included the import from ACL2 into hol90 [Staples, 1999] and HOL4 [Gordon *et al.*, 2006].

Still more challenging is importing not only *statements* but actually reading in and checking *proofs* in the target system. Given the strong tendencies to foundational precision and rigour in this research community, this has attracted more attention recently. Examples include

- hol90 → Coq [Denney, 2000]

- hol90 → NuPRL [Naumov *et al.*, 2001]

- HOL4 → Isabelle/HOL [Obua and Skalberg, 2006]

- HOL Light → Isabelle/HOL [Obua and Skalberg, 2006; Kaliszyk and Krauss, 2013]

- Isabelle/HOL → HOL Light [McLaughlin, 2006]

- HOL Light → Coq [Keller and Werner, 2010]

The work of McLaughlin [2006] on importing Isabelle/HOL to HOL Light shows that the general requirement that the target system should be at least as rich as the source is not an absolute prohibition. Isabelle/HOL extends the simple type theory of HOL with a system of axiomatic type classes, so in some sense is richer. McLaughlin handles this by mapping an Isabelle theorem to an ML *functor* that can be instantiated to produce specific instances.

Translation between systems like HOL4, HOL Light and ProofPower is particularly appealing since despite their differences as systems they implement effectively the same logic. (They do not have exactly the same primitive rules, but they have the same provable theorems and it is easy to implement the primitive rules of one in terms of those of another.) OpenTheory [Hurd, 2010] is a general framework designed to support the transfer of theorems and proofs between HOL family provers, while HOL Zero [Adams, 2010] is a specially simple and transparent version of HOL designed as a vehicle for proof import and checking.

Yet another alternative is still to transfer results without proofs but to have a *machine-formalized* argument about why the two systems correspond. In this case it would be natural to have a simple theorem proving system to act as a metatheory to analyze other systems, perhaps proving relative consistency results etc. and so justifying the importing of results from one to another. Just such a general scheme was proposed as a solution to the current Tower of Babel in the ambitious 'QED manifesto' [Anonymous, 1994]. Perhaps the closest concrete work is in the Logosphere project,[18] using Twelf as the metalogic.

---

[18] http://www.logosphere.org/

## 6.5   User interfaces, search and presentation tools

Interacting with theorem provers and their large formal libraries combines aspects of programming, writing mathematical papers and system specifications, and also aspects of managing and searching large encyclopedias. A comprehensive overview of the early ITP interfaces and of the human-computer interaction (HCI) research [Hewett, 1992] in the ITP world is given by Aitken *et al.* [1998], focusing on HOL and LCF-style systems, and differentiating three views of formal proof: (i) *proof as programming* as for example the tactical programming in the LCF world, (ii) *proof by pointing* as proposed by Bertot *et al.* [1994] for selecting subexpressions of the current goal (typically, using a mouse), and *proof as structure editing* used in the ALF system for editing the proof objects [Magnusson and Nordström, 1993].

A number of these ideas have been incorporated in the Proof General Emacs interface by Aspinall [2000], which has dominated proof development in Coq and Isabelle for over a decade. More recent widely used non-Emacs interfaces to Coq and Isabelle include the GTK-based CoqIDE [Bertot and Théry, 1998], the web-based ProofWeb[19] by Kaliszyk [2007] and the Isabelle/jEdit IDE by Wenzel [2012]. The traditional mode of interaction in such interfaces for LCF-style systems has been coupled with the read-eval-print loop of the underlying interpreter-like ITPs, using region-locking corresponding to the processed part of the formal article, together with forward and backward (undo/reload) commands. For the compiler-style Mizar system, the Emacs user interface [Urban, 2006a] instead processes the whole article at one go (similarly to some IDEs for TeX and for compiled languages like C and Pascal), directly putting the error messages afterwards into the edited buffer. This relies on the general speed of the Mizar verifier, and also on mechanisms that speed up the verification by omitting the parts that had already been verified and by parallelizing the verification process [Urban, 2012]. An early example of a rapprochement between these two user-interface approaches is tmEgg [Mamane and Geuvers, 2007] – a document oriented Coq plugin for TEXMacs that allows more liberal editing mode (inspired by LaTeX) than Proof General and CoqIDE. Similar document-centric effort is done in the Isabelle/JEdit plugin.

As in programming and scientific writing, the authoring can be done with different degrees of collaboration: there are essentially one-person projects such as HOL Light and its core libraries, but also widely distributed loosely managed projects such as the construction of the Mizar Mathematical Library. A number of projects are in between these two extremes. In the Flyspeck project, the formal proofs of a large part of the formalization outlined by Tom Hales have been carried out by a group of mathematicians in Hanoi. Another example of such collaborative project with a strong leader is the formal proof of the Feit-Thompson theorem, with Georges Gonthier proposing and controlling the overall formalization plan, the proof style, the integrity of concept and theorem naming, the automation methods used, etc. While in the one-person projects, most of the library developed is typically remembered by its author, this is no longer the case with large

---

[19]http://proofweb.cs.ru.nl/

collaborative projects where library re-use turns into a major issue. This has led to the development of a number of search facilities. The most obvious and still widely used search method is just `grep` (regular expression search), which can be further modified in various ways: for example searching the whole multiline statements and searching the formalizations in their library-processing order has turned quite useful when working with the Mizar library [Urban, 2006a]. Regular expressions however have only limited knowledge about the term and type structure, symbol overloading, and other ITP specifics.

The next level of ITP search tools is typically aware of such issues, providing more semantic ways for specifying the search patterns. In HOL, Isabelle and Coq this includes tools such as `find_theorems` and `SearchAbout`, provided directly by the interactive shell of these systems and well-integrated with the underlying datastructures used for representing terms, formulas and theories. A potential disadvantage of such deeply integrated tools is that they usually work only with the theories loaded by the user into the current interactive session, possibly omitting a number of other developments and libraries because the user does not know about them, or because they require additional effort to load due to various incompatibilities. This has led to the development of tools that are both globally usable across all developments (in the same way as grep is) and 'semantic', i.e., allowing the ITP-specific search patterns. Such tools include Bancerek's MML Query [Bancerek and Rudnicki, 2003; Bancerek, 2006], which processes and searches the whole Mizar library and all its versions, and the Whelp system [Asperti *et al.*, 2004] developed at University of Bologna for searching Coq libraries.

Such semantic search tools are already close to the present automated theorem proving linkups for Isabelle, Mizar and HOL Light mentioned above. Where such full-scale proof finding turns out to be too hard, the use of ATP-indexing methods such as perfect discrimination trees is still possible for performing more restricted search such as type-aware subsumption over the millions of lemmas in the whole ITP libraries [Urban, 2006b]. Also, the machine learning premise-selection methods, used today mostly in the context of large-theory ATP, can be used as separate search tools for the libraries and integrated into the authoring interfaces. An early example is the Mizar Proof Advisor [Urban, 2006a]. All such 'global' tools are typically external to the ITP systems and often work in a server mode over an internet connection, usually providing further presentation capabilities.

This introduces another large topic, which is presentation of the ITP developments, targeting not just their authors (writers), but a much wider audience interested in their possible re-use, or just in the study of fully formal proofs. Particularly with the arrival of HTML and the World Wide Web in the 1990s, formal mathematical libraries, where the meaning of each symbol and proof step is completely disambiguated, seemed to be a strong candidate for general presentation of mathematics in a cross-linked HTML-ized form where learning the exact definition of a particular concept is just one click away. The first large project that took advantage of HTML-based cross-linking was the Journal of

Formalized Mathematics[20] started in 1995, presenting the Mizar abstracts (i.e., with proofs omitted) in a completely cross-linked HTML format. Similar cross-linking is today available for Coq[21] (using the coqdoc tool) and for MetaMath [Megill, 1996] developments.[22] Practically all large ITP libraries, such as the Isabelle Archive of Formal Proofs[23], the Coq Users' Contributions[24], and the HOL Light, Flyspeck and HOL4 libraries, are easily accessible on the web today. The HTML presentations have gradually acquired more features than just symbol-linking: particularly useful is the optional display of the proof state (thesis) after each proof step in tools like Proviola [Tankink *et al.*, 2010] and in the more recent Mizar HTML[25] [Urban, 2005], linking with online interfaces to the ITPs and ATPs such as ProofWeb and MizAR[26] [Urban and Sutcliffe, 2010], experimental wiki platforms for formal mathematics [Urban *et al.*, 2010; Alama *et al.*, 2011], and even linking with the world of the Semantic Web [Tankink *et al.*, 2012] which has taken off in the meantime, and particularly benefited from informal resources such as Wikipedia.

Describing mathematics in Wikipedia has in some sense inherited some of the library and encyclopedia-building spirit of the formal libraries projects, but allowed much more massive collaboration in the informal setting, resulting in much faster coverage and cross-linking of the mathematical landscape. In 2010 — only a decade since the inception of Wikipedia — the number of Wikipedia mathematical articles grew over 25,000,[27] with the number of active participants in the Mathematics WikiProject counting over 400. These numbers dwarf the long-developed formal ITP libraries by an order of magnitude. Allowing a similarly explosive level of collaboration and providing further alignment of such shallow semantic corpora with the fully formal ITP libraries are again very interesting research topics. An early effort in this direction is Hales's cross-linking of his informal LATEX book on Flyspeck [Hales, 2012] with the formal Flyspeck development in HOL Light and a wiki platform for such joint informal/formal alignment [Tankink *et al.*, 2013].

## 6.6  *Ultimate reliability*

For purely automated theorem proving, the main emphasis is usually on power and convenience. While of course reliability is always an important goal, it is not usually at the forefront of the research agenda. In interactive theorem proving, by contrast, we are not usually so interested in having the computer impress us with its creativity (though we are always willing to be pleasantly surprised). The

---

[20]http://mizar.org/JFM
[21]http://coq.inria.fr/library/
[22]http://us.metamath.org/mpegif/mmset.html
[23]http://afp.sourceforge.net/
[24]http://coq.inria.fr/contribs
[25]http://mizar.org/version/current/html/
[26]http://mizar.cs.ualberta.ca/~mptp/MizAR.html
[27]https://web.archive.org/web/20101222014223/http://en.wikipedia.org/wiki/Portal:
Mathematics

primary goal is to rather to assist the human to construct a proof while checking all the low-level details precisely. Because of this, reliability comes higher up the list of desiderata, since it's hard to justify the extra labor usually involved in formalization if the end result is likely to be just as fallible as a human proof.

However, the degree of importance attached to reliability varies among the ITP system communities and among individual members. In the PVS world there has traditionally been a rather relaxed and pragmatic view of correctness, while the HOL world has usually attached a very high level of importance to — one might also say fetishized — secure foundations and reliability. A simplified caricature of the case against caring might look something like this:

> Even if a theorem prover does have obscure bugs that in principle affect soundness, for proofs in verification it is still orders of magnitude more reliable than a human proof. In any case such issues pale into insignificance compared with the very real problems of getting specifications right and correctly modeling the system, where a small error can make any 'correctness proof' meaningless. Verification is almost always more valuable for discovering bugs rather than producing some nebulous 'guarantee', and a prover that lets you find interesting bugs quickly is the most useful, even if it has issues of its own.

As the reader may guess, we do not entirely accept this critique. In any case, there is surely considerable intellectual interest in seeing how far a rigorously foundational approach can be pushed. So, without necessarily taking any particular position on this issue, we will just consider how systems may be made highly reliable *if* one does indeed care enough for it to be an issue. First of all, we can identify two major sources of unreliability:

- The logic implemented by the theorem prover or the mathematical axioms assumed may be unsound or even inconsistent.[28]

- The actual implementation of the logical system as computer code may be incorrect, or the implementation may include additional infrastructure like arithmetic decision procedures not part of the abstract description of the logic.

We should not ignore the first possibility. Many notable logicians including Frege, Curry and Martin-Löf have proposed logical systems that turned out to be inconsistent for fundamental conceptual reasons. And even starting from a valid semi-formal idea, specifying the details of a logical system can be a painstaking and error-prone activity where even famously careful and precise workers like Church have erred — problems with variable capture are a perennial source of errors. Such

---

[28]Since Gödel we have known that a system can be consistent yet unsound — for example add to a consistent system an axiom asserting the inconsistency of that system. We will just refer vaguely to this nexus of concepts when we use words like 'sound', 'correct' or 'reliable'.

errors seem fairly unlikely using relatively simple and time-tested foundations like first order set theory or HOL-style simple type theory, and more likely when the foundations are newer and more complex. Yet even in a system as simple as HOL, early versions had a conceptual error in the definitional mechanism (allowing polymorphic type variables occurring in the definiens but not the constant) that led to their being inconsistent, as was later discovered independently by Roger Jones and Mark Saaltink.

Nevertheless, we will concentrate mainly on errors of the second kind. Since serious proof checkers are large and complex systems of software, skepticism about their correctness is certainly reasonable. In general, it is much easier to feel confident about a theorem proving program that has a relatively small trusted kernel so that correctness of this kernel is all that needs to be established. Type theory provers are often organized according to what has been called the *de Bruijn criterion* [Barendregt, 1997]: they can output a proof that is checkable by a much simpler checker program that can be considered to be the logical kernel. LCF systems of course already do perform all inference using a small trusted kernel, and they satisfy the de Bruijn criterion into the bargain because it is very easy to instrument the kernel so that it actually records proofs that can be separately checked [Wong, 1993]. Most of the experiments on sharing proofs among different systems outlined above depend on exactly this kind of proof export, and such checking using another system provides an additional level of confirmation even beyond the rigor of the native LCF kernel. Thus, systems that are architected around logical kernels, and those in the LCF style in particular, can reasonably be considered more reliable than those with freer design principles, other things being equal.

But of course, the relative size and complexity of the kernels is a significant factor in reliability. For example, HOL Light [Harrison, 1996a] has a logical kernel consisting of about 600 lines of mostly functional OCaml, and the most complex inference rule (type instantiation, `INST_TYPE`) can be described as

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]}$$

On the other hand, Coq's logical kernel consists of about 20,000 lines of code, sometimes quite stateful and with some 2,500 of them being in C, one of the more complex rules ($\mathcal{K}$-match) being the following[29]

---

[29]Strictly speaking this is drawn from the documentation for Matita [Asperti *et al.*, 2006], which is supposed to be an implementation of essentially the same foundations, though it is no longer exactly the same. Indeed, there does not seem to be any precise written specification of Coq's current foundations, other than the actual code.

$$\frac{\begin{array}{l} (\Sigma', \Phi', \mathcal{I}) \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \rhd_{\mathrm{whd}} I_l^p \overrightarrow{u_l} \overrightarrow{u_r} \\ A_p[\overrightarrow{x_l/u_l}] = \Pi \overrightarrow{y_r : Y_r}.s \quad K_p^j[\overrightarrow{x_l/u_l}] = \overrightarrow{\Pi x_{n_j}^j : Q_{n_j}^j.I_l^p \overrightarrow{x_l} \overrightarrow{v_r}} \quad j = 1 \dots m_p \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash U : V \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash V \rhd_{\mathrm{whd}} \Pi z_r : \overrightarrow{Y_r}.\Pi z_{r+1} : I_l^p \overrightarrow{u_l} \overrightarrow{z_r}.s' \\ (s, s') \in \mathsf{elim}(\mathsf{PTS}) \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \overrightarrow{\lambda x_{n_j}^j : P_{n_j}^j}.t_j : T_j \qquad\qquad\qquad\qquad\qquad j = 1, \dots, m_p \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T_j \downarrow \overrightarrow{\Pi x_{n_j}^j : Q_{n_j}^j.U \overrightarrow{v_r}} (k_j^p \overrightarrow{u_l} \overrightarrow{x_{n_j}^j}) \qquad j = 1, \dots, m_p \end{array}}{\begin{array}{l} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \quad \mathbf{match}\ t\ \mathbf{in}\ I_l^p\ \mathbf{return}\ U \\ \qquad [k_1^p\ (\overrightarrow{x_{n_1}^1 : P_{n_1}^1}) \Rightarrow t_1 \mid \dots \mid k_{m_p}^p\ (\overrightarrow{x_{n_{m_p}}^{m_p} : P_{n_{m_p}}^{m_p}}) \Rightarrow t_{m_p}\ ] : U \overrightarrow{u_r} t \end{array}}$$

$(\mathcal{K}-\mathrm{match})$

Even if the system demonstrably only produces valid theorems, there is the danger that humans can misunderstand the *appearance* of those theorems. First of all, one can be confused over the precise definitions involved, particularly if they have some less intuitive features like aggressive totalization, though of course all systems make it easy to inspect those definitions. But for many purposes the designers of theorem provers tend to think at the level of abstract syntax and neglect the concrete representation that the user sees. From a practical point of view, one might consider all this as part of the logical kernel, since not many users in practice would be willing to read the abstract syntax trees. As such, even nominally reliable systems can produce quite confusing and counterintuitive results, as Adams [2010] and Wiedijk [2012a] observe.

One of the primary intended applications of interactive theorem provers is in computer system correctness, that is, 'proving programs correct' (we discuss this in more detail in the next section). As such, just as compilers compile themselves, it seems natural to use proof assistants to verify themselves. There are two distinct but related ideas that we might try to pursue here: prove metaproperties such as consistency of the logical system itself, and verify that the prover's code correctly implements that logical system. Combining these, we could actually conclude with pretty high confidence that the output of the actual implementation is correct.

However, the analogy with compilers breaks down because of limitative results of logic. Tarski's theorem on the undefinability of truth tells us that no formal system of the type we consider here can formalize its own semantics, and Gödel's Second Incompleteness Theorem tells us that it cannot prove its own consistency in any way at all — unless of course it *isn't* consistent, in which case it can prove anything [Smullyan, 1992]. Thus, even ignoring implementation aspects, successfully proving $\vdash_P Con(P)$, the consistency of the logic implemented by $P$ within $P$ itself, would actually imply either that $P$'s logic is *not* consistent or that the implementation is *wrong*!

One can still use a proof checker to formalize its own inference system (far from being ruled out by Gödel-type theorems, this is a key idea in their usual proof) and so discuss the correctness of a proof checking program relative to the assumed

correctness of the logic [Wright, 1994]. If one wants a truly semantic correctness theorem, perhaps the most satisfying approach would be to prove the correctness of prover $P$ inside a different prover $Q$ implementing a sufficiently strong logic for the limitative results not to present an obstacle — for example proving HOL correct using Mizar.

However, probably because not many people are sufficiently conversant with two different systems, most existing experiments have been closer to self-verifications where the same or similar systems are used, but one either proves the correctness of a weakened version or uses additional axioms in the proof. Barras and Werner [1996] describe the formalization inside Coq itself of a proof-checker for the core Calculus of Constructions. Harrison [2006b] presents two HOL-in-HOL consistency proofs that include not only the abstract logic but a reasonably faithful model of the actual system code with the exception of its definitional mechanisms.

- $\vdash_{\mathtt{HOL}} Con(\mathtt{HOL} - \{\infty\})$ proves in plain HOL the consistency of HOL with the axiom of infinity removed. This removal allows the whole type hierarchy to be modeled inside one infinite set; although this is a mathematically trivial universe there is considerable interest in the basic correctness of the implementations of the various syntax operations and inference rules (no variable capture etc.)

- $I \vdash_{\mathtt{HOL}} Con(\mathtt{HOL})$ proves the consistency of plain HOL inside HOL strengthened with a new axiom $I$ about sets, that there is an uncountable cardinal $\kappa$ so that whenever $\lambda < \kappa$, we also have $2^\lambda < \kappa$. From the point of view of ZF set theory this trivially holds (e.g. $\kappa = |V_{\omega+\omega}|$), but with respect to the simple type construction principles of HOL this plays a role analogous to the existence of inaccessible cardinals.

Even Harrison's work does not model all aspects of the actual implementation, and the correspondence between code and its formalization is naive. These shortcomings, however, are being systematically addressed. Kumar *et al.* [2014] have ported the proof to use HOL4 as the metaframework (while still being a verification *of* HOL Light), extended the consistency proof to cover the definitional principles and proven the correctness of an implementation in a special ML dialect CakeML with a machine-checked formal semantics. In fact, we could not even expect to prove a similarly strong result for the main OCaml implementation of HOL Light, since OCaml has additional 'real world programming language' mechanisms beyond traditional functional constructs that can be used to break the LCF abstraction boundaries, such as a general type-casting operation `Obj.magic`. CakeML even has a formal semantic link via *decompilation* to machine code, meaning that one can anticipate a correctness proof for an implementation of HOL Light from a toplevel semantic soundness right down to the level of the machine code that runs it. An analogous result has already been achieved [Myreen and Davis, 2014] for Milawa, a simplified bootstrapping version of ACL2 developed by Jared Davis [2009]. These are remarkable achievements very much in the spirit of the pioneering CLInc stack [Young, 1993].

Even if the implementation is proved correct at this level, we can never ultimately banish skepticism completely. We are talking about things running on a computer, a real physical object, and we cannot make any final statement connecting any mathematical model to reality, nor can we exclude soft errors in the computer hardware (these are intermittent faults usually resulting from external particle bombardment, which are becoming increasingly significant as miniaturization advances). However, using a small logical kernel, performing rigorous verification from its semantics down to the machine code that runs it, and being able to independently check proofs in other systems — perhaps repeatedly so to make the chance of soft errors even more astronomically small — seems to give about the best guarantee one could possibly hope for.

## 6.7   Applications

Interactive theorem provers have found applications in two main areas, the formalization of mathematics and the formal verification of computer systems. In a general sense, both of these involve the formalization of mathematical proof, but the applications tend to place different demands on a system, and usually those we have described here were developed for some more or less specific application. For example, Mizar was clearly intended for formalizing mathematics, while HOL was intended for verifying hardware. Yet most of the systems have found unexpected applications in other areas too. And sometimes the two areas are mutually reinforcing — for example in order to verify some very concrete and practical floating-point algorithms, one may need first to verify a significant amount of real analysis and number theory [Harrison, 2000].

The *formalizability in principle* of mathematical proof is widely accepted among professional mathematicians as the final arbiter of correctness. Bourbaki [1968] clearly says that 'the correctness of a mathematical text is verified by comparing it, more or less explicitly, with the rules of a formalized language', while Mac Lane [1986] is also quite explicit (p. 377):

> As to precision, we have now stated an absolute standard of rigor: A
> Mathematical proof is rigorous when it is (or could be) written out in
> the first-order predicate language $L(\in)$ as a sequence of inferences from
> the axioms ZFC, each inference made according to one of the stated
> rules. [. . . ] When a proof is in doubt, its repair is usually just a partial
> approximation to the fully formal version.

However, before the advent of computerization, the idea of actually formalizing proofs had seemed quite out of the question. The painstaking volumes of proofs in *Principia Mathematica* [Whitehead and Russell, 1910] are for extremely elementary results compared with even classical real analysis, let alone mathematics at the research level. It is only because of the availability of modern interactive proof assistants that we can contemplate the *actual* formalization of non-trivial amounts of contemporary mathematics.

[1] The result of Problem 11 contradicts the results announced by Levy [1963b]. Unfortunately, the construction presented there cannot be completed.

[2] The transfer to ZF was also claimed by Marek [1966] but the outlined method appears to be unsatisfactory and has not been published.

[3] A contradicting result was announced and later withdrawn by Truss [1970].

[4] The example in Problem 22 is a counterexample to another condition of Mostowski, who conjectured its sufficiency and singled out this example as a test case.

[5] The independence result contradicts the claim of Felgner [1969] that the Cofinality Principle implies the Axiom of Choice. An error has been found by Morris (see Felgner's corrections to [1969]).

Figure 6: Footnotes from Jech's 'Set Theory', p. 118

Such formalization may answer a real need. Well-established branches of mathematics such as elementary real analysis are by now precisely formulated (one might almost say ossified) and presented in rigorous way. But at the research level, mathematics is often quite vaguely formulated, because mathematicians can usually rely on the deep understanding and intuition of themselves and their fellows to keep them out of trouble. Indeed, as Lakatos [1976] describes, mathematicians often begin to prove theorems before the fundamental concepts they involve are clearly articulated, and the concepts often change in response to criticism of such theorems. Even if mathematical assertions are formulated precisely, there is still plenty of scope for errors in proofs. Mathematical proofs are subjected to peer review before publication, but there are plenty of well-documented cases where published results turned out to be faulty. A notable example is the first purported proof of the 4-color theorem [Kempe, 1879]; the error in this proof was eventually pointed out in print a decade later [Heawood, 1890]. A book by Lecat [1935] gave 130 pages of errors made by major mathematicians up to 1900. With the abundance of theorems being published today, often emanating from writers who are not trained mathematicians, one fears that a project like Lecat's would be practically impossible, or at least would demand a journal to itself! Consider for example the five footnotes from a single page of [Jech, 1973] shown in Figure 6.

Such examples can be adduced to argue that the usual social process works, at least for results that are considered sufficiently important. Yet we live in an age of increasingly large proofs relying on highly specialized knowledge, where the number of people who reasonably could check them is small indeed. In cases where the proof relies on extensive computer calculation, it is difficult to really say convincingly that any human being has checked it [Lam, 1990]. Even Ruffini's, arguably quite correct, 1799 proof of what is now a very classical result, the insolvability of general quintics by radicals, was in its day considered too unwieldy to reward study and was largely ignored. Nowadays we have proofs of key results that are much larger, such as the classification of finite simple groups, which is spread over numerous journal articles with a total page count of the order of 10,000. Is the

social process really a reliable method of checking such huge proofs?

Over the last few decades there has been a significant amount of mathematics formalized in interactive proof assistants. Wiedijk [2009] surveys the libraries of formal mathematics provided by several interactive theorem provers, with the Mizar MML discussed above being the largest.[30] Despite this, the frontier of what has been formalized is on average perhaps 100 years behind the actual development of that mathematics — for example it was only quite recently that some of the jewels of 19th century mathematics such as the Prime Number Theorem were formalized [Avigad *et al.*, 2007; Harrison, 2009a]. The first notable exception to this state of affairs was arguably Gonthier's machine-checked proof [Gonthier, 2005; Gonthier, 2008] of the 4-color theorem, originally proved in 1976 [Appel and Haken, 1976]. This also showed convincingly that proofs involving a large amount of computation could be brought within the purview of formalization. More recently Gonthier has also led a team that formalized a proof of of the Feit-Thompson theorem, a milestone in group theory from 1963 that forms an important part of the classification of finite simple groups [Gonthier *et al.*, 2013].

In one case formalization has reached the frontier of mathematics research itself and led to the advocacy of formalization by a notable contemporary mathematician, Thomas Hales. The venerable *Kepler conjecture* states that no arrangement of identical balls in ordinary 3-dimensional space has a higher packing density than the obvious cannonball arrangement. Hales, working with Ferguson, finally proved this conjecture in 1998, but the size of the proof was daunting: about 300 pages of traditional mathematics: geometry, measure, graph theory and related combinatorics, as well as about 40,000 lines of supporting computer code performing graph enumeration, nonlinear optimization and linear programming. Hales submitted the proof to *Annals of Mathematics*, and after four years of deliberation, the referees were still unable to provide a satisfactory review:

> The news from the referees is bad, from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for.

Hales's proof was indeed eventually published [Hales, 2005], and no significant error has been found in it [Hales *et al.*, 2010]. Nevertheless, the verdict is disappointingly lacking in clarity and finality. As a result, Hales initiated a project called *Flyspeck* to completely formalize the proof [Hales, 2006]. This project involved a sustained effort by a large and geographically distributed team, many of whom were originally complete novices in the field of formalization. In a major milestone for formalization of mathematics, the project has just been completed at time of writing.[31] That is, all the ordinary mathematics has been formalized, the linear and nonlinear optimizations have been reimplemented in a proof-producing

---

[30]The list `http://www.cs.ru.nl/~freek/100/` gives a more selective ideas about formalizations of specific theorems.

[31]`https://code.google.com/p/flyspeck/wiki/AnnouncingCompletion`

fashion and the graph enumeration code has been rewritten in ML and formally proved correct. That a contemporary proof so large, intricate and heterogeneous can be completely formalized down to the most basic logical principles indicates just how much can already be achieved with sufficient labour. The process of formalization, and some other parallel developments, have resulted in a significant simplification of the proof and its reorganization into a formalization-friendly form [Hales, 2012].

Much of the early development of interactive theorem provers, as we have noted above, was motivated not by pure mathematics but by problems in computer system verification, with the verification of security properties a particular focus [MacKenzie, 2001]. Using formalization to verify the correct behavior of computer systems (e.g. hardware, software, protocols and their combinations) is an easy application to justify on utilitarian grounds. We might wish to prove that a sorting algorithm really does always sort its input list, that a numerical algorithm does return a result accurate to within a specified error bound, that a server will under certain assumptions always respond to a request, or will ensure certain security properties, etc. etc.

Although recently many of the most notable successes in verification have been in *hardware* verification, and there was great interest in higher-level *system* verification, the traditional focus has been *software* verification or 'proving programs correct'. (Perhaps in the 1970s, hardware was considered too simple to need verifying?) Traditionally, program verification has usually been based on special axiomatic systems for simple imperative programming languages, which were developed by Hoare [1969] and Dijkstra [1976] among others from earlier work by Naur [1966] and Floyd [1967]. The axiomatic approach to program semantics asserts certain relationships between the *precondition* and *postcondition*, which are predicates over the state before and after execution — in Dijkstra's formulation a program is identified with a *predicate transformer* mapping any postcondition to the weakest precondition that ensures that postcondition.

Actual correctness proofs can be done in several ways. One can use a proof system embodying the Hoare logic rules, or use *refinement* of the specification into a program [Back, 1980; Morgan, 1990]. Alternatively, one can annotate a program with special assertions indicating that certain properties of the state should hold whenever that point is reached. Such an annotated program can be distilled into a set of purely mathematical assertions called *verification conditions*: if these can be proved then the correctness of the whole program in terms of Hoare logic follows automatically. The first mechanical verifier was built by King [1969] in a PhD supervised by Floyd, with another more interactive one soon developed by Good [1970] under the supervision of Ralph London, who had himself pioneered manual proofs of programs [London, 1970]. Subsequently Good *et al.* [1979] developed the GYPSY interactive program verifier which was applied quite successfully to some non-trivial problems [Good, 1983], and had a significant influence on the field of automated reasoning.

The use of special verification frameworks continues to this day with systems

such as KIV [Reif, 1995] and various tools supporting the B method [Abrial, 1996].
But the use of general interactive theorem provers for verification proofs has become a popular alternative to the use of such systems. Of course, the boundary
between the two is not always sharp, but interactive provers generally make available a richer mathematical infrastructure, make it easier to call on a wide variety
of automated methods, and offer good standards of reliability. In particular, it
is not necessary to hardwire into the framework a connection with a specific programming language, and one can reason explicitly about the semantics of different
languages and their relationship, for example to talk about compiler correctness.
A significant theme here is the *embedding* of other formalisms, whether they be
programming languages, hardware description languages, specification languages,
and even other logics, inside general theorem provers like HOL and PVS. One
approach, perhaps the most obvious, is to create formal models of the syntax and
semantics of the formalism inside the prover's logic and use the system to reason
about them. Following Boulton *et al.* [1993], this has become known as *deep
embedding*, contrasting with the alternative of *shallow embedding*. In the latter,
language constructs are associated directly with logical entities, and the notation
is merely a convenience. This fits naturally with the view, expressed for example by Dijkstra [1976], that a programming language should be thought of first
and foremost as an algorithm-oriented system of mathematical notation, and only
secondarily as something to be run on a machine. A seminal paper by Gordon
[1989] showed how a simple imperative programming language could be semantically embedded in higher order logic in such a way that the classic Floyd-Hoare
rules simply become derivable theorems.[32]

Such shallow embedding seems especially natural in the case of embedding less
expressive logics — for instance the basic operators of temporal logic [Gabbay *et
al.*, 1994] can easily be considered just as shorthands for quantified statements
about sequences in higher-order logic. When one writes '$\Box P$' in LTL, Linear
Temporal Logic [Pnueli, 1977], for example, it means that '$P$ is true now and at all
future times'. Considering temporal propositions systematically as mappings from
natural numbers ('times') to Booleans, we can actually *define* $\Box$ by the equation
$(\Box P)(t) =_{def} \forall t'.t' \geq t \Rightarrow P(t')$. One then doesn't need any layer of translation
between LTL and the richer mathematical framework. Actually, the fact that this
is a flexible and satisfying way to reason about other logics may explain why any
explosion of interest in provers for different logics (as might have been anticipated
by the designers of logical frameworks) has so far been relatively muted.

A kind of reversal of the idea of semantic embedding is to consider a subset of the
logic as a programming language and then translate (often 'extract', though some
use that in the sense of extracting programs from constructive proofs) into a real
programming language, which is most naturally a *functional* one like Lisp or ML,
languages that can be said to 'wear their semantics on their sleeves' [Henson, 1987].
For a system like ACL2, there is by design a near-equivalence between the logic and

---

[32]Gordon (private communication) recalls that the idea of embedding Hoare logic directly in
HOL in this way may have come from Roger Jones.

the implementation language, so this works in a particularly straightforward and elegant way. For richer logics like HOL or Coq, one needs to carefully demarcate a subset that has a natural correspondence with a real functional language. Even without the actual extraction to code, Coq supports a highly efficient reduction mechanism inside the logic so that one can effectively 'run programs' without stepping out of the formal inference system, at least in principle. This can be done in other systems like HOL too, but because of the much more parsimonious logical kernel, the performance penalty is considerable.

Many in the 1970s dreamed of the pervasive use of correctness proofs from top to bottom: applications, operating systems, compilers, programming languages and hardware. A pioneering example of such a verified tower of basic systems was the 'CLInc stack' [Young, 1993]. There have been more recent projects in the same style such as Verisoft [Paul, 2008], but many recent efforts have been more piecemeal and focused on particular areas, with applications to floating-point arithmetic being particularly successful [Moore *et al.*, 1998; Russinoff, 1998; O'Leary *et al.*, 1999; Harrison, 2000; Kaivola and Aagaard, 2000; Kaivola and Kohatsu, 2001; Sawada and Gamboa, 2002; Slobodová, 2007; Hunt and Swords, 2009; O'Leary *et al.*, 2013]. A key feature of such examples is that they are being carried out by those in the hardware industry, not only by academics in universities, indicating at least some successful penetration of interactive theorem proving into the 'real world'.

Two substantial formal verifications carried out with interactive theorem proving in recent years are the correctness proofs of a 'CompCert' optimizing compiler from a significant subset of C using Coq [Leroy, 2009], and of a designed-for-verification version of the commercial L4 microkernel using Isabelle/HOL [Klein *et al.*, 2010]. While these are currently isolated and independent efforts, they demonstrate the applicability of formal methods to key items of system software, and have to some extent inspired further interconnecting verifications — for example the CompCert compiler has been extended with a formalized treatment of floating-point arithmetic [Boldo *et al.*, 2013] while the actual ARM binary for the L4 microkernel has also been verified, using HOL4, by Sewell *et al.* [2013]. Much of the impetus for the development of theorem provers and verification frameworks arose from the interest in proving isolation properties of time-sharing operating systems, as we have seen, so it is especially pleasing to see that this was not just a dream, but can be realized — albeit with considerable human effort — using today's technology.

## ACKNOWLEDGEMENTS

## BIBLIOGRAPHY

[Aagaard and Harrison, 2000] M. Aagaard and J. Harrison, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Abrahams, 1963] Paul Abrahams. *Machine Verification of Mathematical Proof.* PhD thesis, Massachusetts Institute of Technology, 1963.

[Abrial, 1996] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Adams, 2010] Mark Adams. Introducing HOL Zero. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Proceedings of ICMS 2010, the Third International Congress on Mathematical Software*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143, Kobe, Japan, 2010. Springer-Verlag.

[Agrawal *et al.*, 2004] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[Aitken *et al.*, 1998] J.S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive theorem proving: An empirical study of user activity. *Journal of Symbolic Computation*, 25(2):263 – 284, 1998.

[Alama *et al.*, 2011] Jesse Alama, Kasper Brink, Lionel Mamane, and Josef Urban. Large formal wikis: Issues and solutions. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 133–148. Springer, 2011.

[Allen *et al.*, 1990] S. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 95–107, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Andersen and Petersen, 1991] Flemming Andersen and Kim Dam Petersen. Recursive boolean functions in HOL. In Archer et al. [1991], pages 367–377.

[Anonymous, 1994] Anonymous. The QED Manifesto. In Bundy [1994], pages 238–251.

[Appel and Haken, 1976] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82:711–712, 1976.

[Archer *et al.*, 1991] Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors. *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, University of California at Davis, Davis CA, USA, 1991. IEEE Computer Society Press.

[Asperti *et al.*, 2003] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical Knowledge Management in HELM. *Ann. Math. Artif. Intell.*, 38(1-3):27–46, 2003.

[Asperti *et al.*, 2004] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2004.

[Asperti *et al.*, 2006] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *LNCS*, pages 18–32. Springer, 2006.

[Aspinall, 2000] David Aspinall. Proof General: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[Autexier *et al.*, 2010] Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors. *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *LNCS*. Springer, 2010.

[Avigad *et al.*, 2007] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1:2):1–23, 2007.

[Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Back, 1980] Ralph Back. *Correctness Preserving Program Transformations: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[Balaa and Bertot, 2000] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Aagaard and Harrison [2000], pages 1–16.

[Bancerek and Rudnicki, 2002] Grzegorz Bancerek and Piotr Rudnicki. A compendium of continuous lattices in Mizar. *J. Autom. Reasoning*, 29(3-4):189–224, 2002.

[Bancerek and Rudnicki, 2003] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in MML. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2003.

[Bancerek, 2006] Grzegorz Bancerek. Information retrieval and rendering with MML Query. In Jonathan M. Borwein and William M. Farmer, editors, *MKM*, volume 4108 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2006.

[Barendregt, 1992] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, 1992.

[Barendregt, 1997] Henk Barendregt. The impact of the lambda calculus on logic and computer science. *Bulletin of Symbolic Logic*, 3:181–215, 1997.

[Barras and Werner, 1996] Bruno Barras and Benjamin Werner. Coq in Coq. Unpublished report, available at `www.lix.polytechnique.fr/~barras/publi/coqincoq.pdf`, 1996.

[Barringer *et al.*, 1984] H. Barringer, J. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[Bates and Constable, 1985] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7:113–136, 1985.

[Beckert and Posegga, 1995] Bernhard Beckert and Joachim Posegga. lean$\mathcal{T}^A\mathcal{P}$: Lean, tableau-based deduction. *Journal of Automated Reasoning*, 15:339–358, 1995.

[Beeson, 1984] M. J. Beeson. *Foundations of constructive mathematics: metamathematical studies*, volume 3 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1984.

[Berghofer and Wenzel, 1999] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL — lessons learned in formal-logic engineering. In Bertot et al. [1999], pages 19–36.

[Bergstra *et al.*, 2007] J. A. Bergstra, Y. Hirshfield, and J. V. Tucker. Meadows and the equational specification of division. *Theoretical Computer Science*, 410:1261–1271, 2007.

[Bertot and Théry, 1998] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.*, 25(2):161–194, 1998.

[Bertot *et al.*, 1994] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994.

[Bertot *et al.*, 1999] Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, Nice, France, 1999. Springer-Verlag.

[Biggs *et al.*, 1976] Norman L. Biggs, E. Keith Lloyd, and Robin J. Wilson. *Graph Theory 1736–1936*. Clarendon Press, 1976.

[Birtwistle and Subrahmanyam, 1989] Graham Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Bishop and Bridges, 1985] Errett Bishop and Douglas Bridges. *Constructive analysis*, volume 279 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1985.

[Blanchette *et al.*, 2013] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2013.

[Blanchette *et al.*, 2014] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Klein and Gamboa [2014], pages 93–110.

[Bledsoe and Bruell, 1974] W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5:51–72, 1974.

[Bledsoe and Gilbert, 1967] W. W. Bledsoe and E. J. Gilbert. Automatic theorem proof-checking in set theory: A preliminary report. Technical Report SC-RR-67-525, Sandia National Lab, 1967.

[Bledsoe, 1984] W. W. Bledsoe. Some automatic proofs in analysis. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 89–118. American Mathematical Society, 1984.

[Blum, 1993] Manuel Blum. Program result checking: A new approach to making programs more reliable. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 1993. Springer-Verlag.

[Boldo et al., 2013] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Siedel, and Ping Tak Peter Tang, editors, *21st IEEE symposium on computer arithmetic, ARITH 21, proceedings*, pages 107–115. IEEE, 2013.

[Boulton et al., 1993] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland.

[Boulton, 1992] Richard Boulton. Boyer-Moore automation for the HOL system. In Claesen and Gordon [1992], pages 133–142.

[Boulton, 1993] Richard John Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993. Author's PhD thesis.

[Bourbaki, 1968] Nicolas Bourbaki. *Theory of sets*. Elements of mathematics. Addison-Wesley, 1968. Translated from French 'Théorie des ensembles' in the series 'Eléments de mathématique', originally published by Hermann in 1968.

[Bove et al., 2009] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[Boyer and Moore, 1979] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[Boyer and Moore, 1981] Robert S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[Boyer and Moore, 1988] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, 1988.

[Bridge et al., 2014] James P. Bridge, Sean B. Holden, and Lawrence C. Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, pages 1–32, 2014.

[Bryant, 1986] Randall E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[Buchberger, 1965] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematisches Institut der Universität Innsbruck, 1965. English translation in Journal of Symbolic Computation vol. 41 (2006), pp. 475–511.

[Bumcrot, 1965] R. Bumcrot. On lattice complements. *Proceedings of the Glasgow Mathematical Association*, 7:22–23, 1965.

[Bundy et al., 1991] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–323, 1991.

[Bundy, 1994] Alan Bundy, editor. *12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, Nancy, France, 1994. Springer-Verlag.

[Burch et al., 1992] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98:142–170, 1992.

[Burstall, 1969] R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12:41–48, 1969.

[Camilleri and Melham, 1992] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1992.

[Caprotti and Oostdijk, 2001] Olga Caprotti and Martin Oostdijk. Formal and efficient primality proofs by the use of computer algebra oracles. *Journal of Symbolic Computation*, 32:55–70, 2001.

[Carnap, 1937] Rudolf Carnap. *The Logical Syntax of Language*. International library of psychology, philosophy and scientific method. Routledge & Kegan Paul, 1937. Translated from 'Logische Syntax der Sprache' by Amethe Smeaton (Countess von Zeppelin), with some new sections not in the German original.

[Cederquist *et al.*, 1998] Jan Cederquist, Thierry Coquand, and Sara Negri. The Hahn-Banach Theorem in Type Theory. In G. Sambin and J. Smith, editors, *Twenty-five years of Constructive Type Theory*, pages 57–72, 1998.

[Chaieb and Nipkow, 2008] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41, 2008.

[Chen, 1992] Wilfred Chen. Tactic-based theorem proving and knowledge-based forward chaining. In Kapur [1992], pages 552–566.

[Chou, 1988] S.-C. Chou. *Mechanical Geometry Theorem Proving*. Reidel, 1988.

[Church, 1940] Alonzo Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Claesen and Gordon, 1992] Luc J. M. Claesen and Michael J. C. Gordon, editors. *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions A: Computer Science and Technology*, IMEC, Leuven, Belgium, 1992. North-Holland.

[Clarke and Emerson, 1981] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dextrer Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.

[Cohen and Mahboubi, 2010] Cyril Cohen and Assia Mahboubi. A formal quantifier elimination for algebraically closed fields. In *Symposium on the Integration of Symbolic Computation and Mechanised Reasoning, Calculemus*, volume 6167 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 2010.

[Constable and Bates, 1983] Robert L. Constable and Joseph L. Bates. The Nearly Ultimate PEARL. Technical Report TR 83-551, Department of Computer Science, Cornell University, January 1983.

[Constable and O'Donnell, 1978] Robert L. Constable and Michael J. O'Donnell. *A programming logic, with an introduction to the PL/CV verifier*. Winthrop, 1978.

[Constable, 1986] Robert Constable. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1986.

[Cooper, 1972] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Melzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Elsevier, 1972.

[Coq development team, 2012] Coq development team. *The Coq proof assistant reference manual*, 2012. Version 8.4pl4.

[Coquand and Huet, 1988] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[Coquand and Paulin, 1990] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[Coquand *et al.*, 2005] Catarina Coquand, Dan Synek, and Makoto Takeyama. An Emacs-Interface for Type-Directed Support for Constructing Proofs and Programs. In *European Joint Conferences on Theory and Practice of Software*, 2005.

[Cousineau and Mauny, 1998] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[Craigen *et al.*, 1991] D. Craigen, S. Kromodimeoljo, I. Meisels, B. Pase, and M. Saaltink. EVES: An overview. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1991.

[Curzon, 1995] Paul Curzon. Tracking design changes with formal machine-checked proof. *The Computer Journal*, 38:91–100, 1995.

[Dahn and Wernhard, 1997] Ingo Dahn and Christoph Wernhard. First order proof problems extracted from an article in the MIZAR mathematical library. In Maria Paola Bonacina and Ulrich Furbach, editors, *International Workshop on First-Order Theorem Proving*, volume 97-50 of *RISC-Linz Report Series*, pages 58–62. RISC-Linz, 1997.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Davis, 1957] M. Davis. A computer program for Presburger's algorithm. In *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University*, pages 215–233. Institute for Defense Analyses, Princeton, NJ, 1957. Reprinted in Siekmann and Wrightson [1983], pp. 41–48.

[Davis, 1981] Martin Davis. Obvious logical inferences. In Patrick J. Hayes, editor, *IJCAI*, pages 530–531. William Kaufmann, 1981.

[Davis, 2009] Jared Davis. *A Self-Verifying Theorem Prover*. PhD thesis, University of Texas at Austin, 2009.

[de Bruijn, 1968a] N.G. de Bruijn. AUTOMATH, a language for mathematics. Technical Report 68-WSK-05, Department of Mathematics, Eindhoven University of Technology, November 1968.

[de Bruijn, 1968b] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions, 1968.

[de Bruijn, 1968c] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration, IRIA, Versailles, France*, number 125 in LNCS, pages 29–61. Springer, December 1968. Also published in Selected Papers on Automath, Studies in Logic and the Foundations of Mathematics, Vol. 133, Ed. R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer, chapter A2, pp. 73-100.

[de Bruijn, 1969] N.G. de Bruijn. SEMIPAL 2, an extension of the mathematical notational language SEMIPAL. Technical Report Notitie 1969/43, Department of Mathematics, Eindhoven University of Technology, March 1969.

[de Bruijn, 1970] N.G. de Bruijn. A Processor for PAL. Technical Report Notitie 1970/30, Department of Mathematics, Eindhoven University of Technology, March 1970.

[de Bruijn, 1978] N.G. de Bruijn. Verslag over het project Wiskundige Taal AUTOMATH, September 1978.

[de Bruijn, 1979] N.G. de Bruijn. Wees contextbewust in WOT. *Euclides*, (55):7–12, 1979.

[de Bruijn, 1990] N.G. de Bruijn. Gedachten rondom AUTOMATH, March 1990.

[de Groote, 1993] Philippe de Groote. Defining Lambda-Typed Lambda-Calculi by Axiomatizing the Typing Relation. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *STACS 93, 10th Annual Symposium on Theoretical Aspects of Computer Science, Würzburg, Germany, February 25-27, 1993, Proceedings*, volume 665 of *LNCS*, pages 712–723. Springer, 1993.

[de Moura and Passmore, 2013] Leonardo de Moura and Grant Passmore. The strategy challenge in SMT solving. In Maria Paola Bonacina and Mark E Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer-Verlag, 2013.

[Delahaye, 2000] David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic Programming and Automated Reasoning, LPAR 2000*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95, La reunion, France, 2000. Springer-Verlag.

[Denney, 2000] Ewen Denney. A prototype proof translator from HOL to Coq. In Aagaard and Harrison [2000], pages 108–125.

[Denzinger and Schulz, 1996] J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21(4/5):523–541, 1996.

[Dick, 2011] Stephanie Dick. The work of proof in the age of human-machine collaboration. *Isis*, 102:494–505, 2011.

[Dijkstra and Scholten, 1990] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dixon, 2005] Lucas Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.

[Eekelen *et al.*, 2011] Marko Van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors. *Interactive Theorem Proving, Second International Conference ITP*, volume 6898 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2003.

[Eisinger and Ohlbach, 1986] Norbert Eisinger and Hans Jürgen Ohlbach. The Markgraf Karl refutation procedure (MKRP). In Jorg H. Siekmann, editor, *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 681–682, Oxford, England, 1986. Springer-Verlag.

[Farmer *et al.*, 1990] William Farmer, Joshua Guttman, and Javier Thayer. IMPS: an interactive mathematical proof system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 653–654, Kaiserslautern, Federal Republic of Germany, 1990. Springer-Verlag.

[Feigenbaum and Feldman, 1995] Edward A. Feigenbaum and Julian Feldman, editors. *Computers & Thought*. AAAI Press / MIT Press, 1995.

[Felty and Howe, 1997] A.P. Felty and D. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In William McCune, editor, *Automated Deduction — CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 351–365, Townsville, Australia, 1997. Springer-Verlag.

[Fitch, 1952] Frederic Brenton Fitch. *Symbolic Logic: an introduction*. The Ronald Press Company, New York, 1952.

[Fleuriot, 2001] Jacques Fleuriot. *A Combination of Geometry Theorem Proving and Nonstandard Analysis with Application to Newton's Principia*. Distinguished dissertations. Springer-Verlag, 2001. Revised version of author's PhD thesis.

[Floyd, 1967] R. W. Floyd. Assigning meanings to programs. In *Proceedings of AMS Symposia in Applied Mathematics, 19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[Furbach and Shankar, 2006] Ulrich Furbach and Natarajan Shankar, editors. *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, Seattle, WA, 2006. Springer-Verlag.

[Gabbay *et al.*, 1994] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic*. Oxford University Press, 1994.

[Gamboa, 1999] Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. PhD thesis, University of Texas at Austin, 1999.

[Ganesalingam, 2013] Mohan Ganesalingam. *The Language of Mathematics: A Linguistic and Philosophical Investigation*, volume 7805 of *Lecture Notes in Computer Science*. Springer-Verlag, 2013.

[Gelerntner, 1959] H. Gelerntner. Realization of a geometry-theorem proving machine. In *Proceedings of the International Conference on Information Processing, UNESCO House*, pages 273–282, 1959. Also appears in Siekmann and Wrightson [1983], pp. 99–117 and in Feigenbaum and Feldman [1995], pp. 134–152.

[Geuvers and Barendsen, 1999] Herman Geuvers and Erik Barendsen. Some logical and syntactical observations concerning the first-order dependent type system lambda-P. *Mathematical Structures in Computer Science*, 9(4):335–359, 1999.

[Gilmore, 1960] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4:28–35, 1960.

[Giunchiglia and Smaill, 1989] Fausto Giunchiglia and Alan Smaill. Reflection in constructive and non-constructive automated reasoning. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, 1989.

[Goldberg and Novikov, 2002] Evgueni Goldberg and Yakov Novikov. BerkMin: a fast and robust Sat-solver. In Carlos Delgado Kloos and Jose Da Franca, editors, *Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, pages 142–149, Paris, France, 2002. IEEE Computer Society Press.

[Gonthier and Mahboubi, 2010] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[Gonthier *et al.*, 2008] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008.

[Gonthier *et al.*, 2013] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Fourth International Conference on Interactive Theorem Proving, ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179, Rennes, France, 2013. Springer-Verlag.

[Gonthier, 1996] Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th international conference on computer aided verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 462–465, New Brunswick, NJ, 1996. Springer-Verlag.

[Gonthier, 2005] Georges Gonthier. A computer-checked proof of the four colour theorem. Available at http://research.microsoft.com/~gonthier/4colproof.pdf, 2005.

[Gonthier, 2008] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 35:1382–1393, 2008.

[Good *et al.*, 1979] D. I. Good, R. L. London, and W. W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1:59–67, 1979.

[Good, 1970] D. I. Good. *Toward a Man-Machine System for Proving Program Correctness*. PhD thesis, University of Wisconsin, 1970.

[Good, 1983] D. I. Good. Proof of a distributed system in Gypsy. In M. J. Elphick, editor, *Formal Specification: Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar*, pages 44–89. University of Newcastle upon Tyne, Computing Laboratory, 1983.

[Goodstein, 1957] R. L. Goodstein. *Recursive Number Theory*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1957.

[Gordon *et al.*, 1979] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gordon *et al.*, 2006] Michael J. C. Gordon, James Reynolds, Warren Hunt, and Matt Kaufmann. An integration of HOL and ACL2. In *Proceedings of the 6th international conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 153–160. IEEE Computer Society Press, 2006.

[Gordon, 1982] Michael J. C. Gordon. Representing a logic in the LCF metalanguage. In D. Néel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.

[Gordon, 1985] Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1985.

[Gordon, 1989] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In Birtwistle and Subrahmanyam [1989], pages 387–439.

[Gordon, 2000] M. J. C. Gordon. From LCF to HOL: A short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language, and interaction: essays in honour of Robin Milner*. MIT Press, 2000.

[Grégoire *et al.*, 2006] Benjamin Grégoire, Laurent Théry, and Benjamin Wener. A computational approach to Pocklington certificates in type theory. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 97–113. Springer-Verlag, 2006.

[Griffin, 1988] Timothy Griffin. Efs - an interactive environment for formal systems. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *LNCS*, pages 740–741. Springer, 1988.

[Guard *et al.*, 1969]  J. R. Guard, F. C. Oglesby, J. H. Bennett, and L. G. Settle. Semi-automated mathematics. *Journal of the ACM*, 16:49–62, 1969.

[Gunter, 1993]  Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In Joyce and Seger [1993], pages 141–154.

[Hales *et al.*, 2010]  Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete and Computational Geometry*, 44:1–34, 2010.

[Hales, 2005]  Thomas C. Hales.  A proof of the Kepler conjecture.  *Annals of Mathematics*, 162:1065–1185, 2005.

[Hales, 2006]  Thomas C. Hales. Introduction to the Flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Hales, 2012]  Thomas Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs*, volume 400 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 2012.

[Hanna and Daeche, 1986]  F. K. Hanna and N. Daeche.  Specification and verification using higher-order logic: A case study.  In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 179–213, 1986.

[Harper *et al.*, 1987]  Robert Harper, Furio Honsell, and Gordon D. Plotkin. A Framework for Defining Logics. In *Proceedings, Symposium on Logic in Computer Science, 22-25 June 1987, Ithaca, New York, USA*, pages 194–204. IEEE Computer Society, 1987.

[Harrison and Théry, 1998]  John Harrison and Laurent Théry. A sceptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[Harrison, 1995a]  John Harrison. Inductive definitions: automation and application. In Phillip J. Windley, Thomas Schubert, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

[Harrison, 1995b]  John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.ps.gz`.

[Harrison, 1996a]  John Harrison.  HOL Light: A tutorial introduction.  In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[Harrison, 1996b]  John Harrison. A Mizar mode for HOL. In Wright et al. [1996], pages 203–220.

[Harrison, 1996c]  John Harrison.  Proof style.  In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[Harrison, 1998]  John Harrison.  *Theorem Proving with the Real Numbers*.  Springer-Verlag, 1998. Revised version of author's PhD thesis.

[Harrison, 2000]  John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[Harrison, 2001]  John Harrison. Complex quantifier elimination in HOL. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, pages 159–174. Division of Informatics, University of Edinburgh, 2001. Published as Informatics Report Series EDI-INF-RR-0046. Available on the Web at `http://www.informatics.ed.ac.uk/publications/report/0046.html`.

[Harrison, 2003]  John Harrison. Isolating critical cases for reciprocals using integer factorization. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings, 16th IEEE Symposium on Computer Arithmetic*, pages 148–157, Santiago de Compostela, Spain, 2003. IEEE Computer Society. Currently available from symposium Web site at `http://www.dec.usc.es/arith16/papers/paper-150.pdf`.

[Harrison, 2006a]  John Harrison.  The HOL Light tutorial.  Unpublished manual available at `http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf`, 2006.

[Harrison, 2006b] John Harrison. Towards self-verification of HOL Light. In Furbach and Shankar [2006], pages 177–191.

[Harrison, 2007] John Harrison. Verifying nonlinear real formulas via sums of squares. In Schneider and Brandt [2007], pages 102–118.

[Harrison, 2009a] John Harrison. Formalizing an analytic proof of the Prime Number Theorem (dedicated to Mike Gordon on the occasion of his 60th birthday). *Journal of Automated Reasoning*, 43:243–261, 2009.

[Harrison, 2009b] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[Heawood, 1890] Percy John Heawood. Map-colour theorem. *Quarterly Journal of Pure and Applied Mathematics*, 24:332–338, 1890. Reprinted in Biggs *et al.* [1976].

[Henson, 1987] Martin C. Henson. *Elements of functional languages*. Blackwell Scientific, 1987.

[Hewett, 1992] Thomas T. Hewett. ACM SIGCHI curricula for human-computer interaction. Technical report, New York, NY, USA, 1992.

[Hickey *et al.*, 2003] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL - A Modular Logical Environment. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *LNCS*, pages 287–303. Springer, 2003.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, 1969.

[Homeier, 2005] Peter V. Homeier. A design structure for higher order quotients. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 130–146, Oxford, UK, 2005. Springer-Verlag.

[Homeier, 2009] Peter V. Homeier. The HOL-Omega logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 244–259, Munich, Germany, 2009. Springer-Verlag.

[Howe, 1992] Douglas J. Howe. Reflecting the semantics of reflected proof. In Peter Aczel, Harold Simmons, and Stanley Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1992.

[Huang *et al.*, 1994] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Omega-MKRP: A proof development environment. In Bundy [1994], pages 788–792.

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

[Hunt and Swords, 2009] Warren A. Hunt and Sol Swords. Centaur Technology media unit verification. In Ahmed Bouajjani and Oded Maler, editors, *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 353–367, Grenoble, France, 2009. Springer-Verlag.

[Hunt, 1985] W. A. Hunt. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas, 1985. Published by Springer-Verlag as volume 795 of the *Lecture Notes in Computer Science* series, 1994.

[Hurd, 1999] Joe Hurd. Integrating Gandalf and HOL. In Bertot et al. [1999], pages 311–321.

[Hurd, 2003] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Workshop on Design and Application of Strategies/Tactics in Higher Order Logics*. NASA, 2003. NASA technical report NASA/CP-2003-212448.

[Hurd, 2010] Joe Hurd. The OpenTheory standard theory library. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, number NASA/CP-2010-216215 in Technical Report, pages 177–191, Langley Research Center, Hampton VA 23681-2199, USA, 2010.

[Iverson, 1980] Kenneth E. Iverson. Notation as a tool of thought. *Communications of the ACM*, 23:444–465, 1980.

[Jaśkowski, 1934] S. Jaśkowski. On the rules of supposition in formal logic. *Studia Logica*, 1:5–32, 1934.

[Jech, 1973] T. J. Jech. *The Axiom of Choice*, volume 75 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1973.

[Jensen and Wirth, 1974]  Kathleen Jensen and Niklaus Wirth. *Pascal user manual and report*. Springer-Verlag, 1974.

[Jeuring *et al.*, 2012]  J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors. *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *LNCS*. Springer, 2012.

[Joyce and Seger, 1993]  Jeffrey J. Joyce and Carl Seger, editors. *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[Kaivola and Aagaard, 2000]  Roope Kaivola and Mark D. Aagaard. Divider circuit verification with model checking and theorem proving. In Aagaard and Harrison [2000], pages 338–355.

[Kaivola and Kohatsu, 2001]  Roope Kaivola and Katherine Kohatsu. Proof engineering in the large: Formal verification of the Pentium (R) 4 floating-point divider. In T. Margaria and Tom Melham, editors, *11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001*, volume 2144 of *Lecture Notes in Computer Science*, pages 196–211, Edinburgh, Scotland, 2001. Springer-Verlag.

[Kaliszyk and Krauss, 2013]  Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Proc. of the 4th International Conference on Interactive Theorem Proving (ITP'13)*, volume 7998 of *LNCS*, pages 51–66. Springer Verlag, 2013.

[Kaliszyk and Urban, 2014a]  Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 2014. `http://dx.doi.org/10.1007/s11786-014-0182-0`.

[Kaliszyk and Urban, 2014b]  Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of Symbolic Computation*, 2014. In press, `http://arxiv.org/abs/1402.3578` .

[Kaliszyk, 2007]  Cezary Kaliszyk. Web interfaces for proof assistants. *Electr. Notes Theor. Comput. Sci.*, 174(2):49–61, 2007.

[Kaluzhnin, 1962]  L. A. Kaluzhnin. On an information language for mathematics. *Applied Linguistic and Machine Translation*, pages 21–29, 1962.

[Kapur, 1992]  Deepak Kapur, editor. *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, Saratoga, NY, 1992. Springer-Verlag.

[Kaufmann and Paulson, 2010]  Matt Kaufmann and Lawrence C. Paulson, editors. *First International Conference on Interactive Theorem Proving, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, Edinburgh, UK, 2010. Springer-Verlag.

[Kaufmann *et al.*, 2000a]  Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[Kaufmann *et al.*, 2000b]  Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[Keller and Werner, 2010]  Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Kaufmann and Paulson [2010], pages 307–322.

[Kempe, 1879]  Alfred Braye Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2:193–200, 1879. Reprinted in Biggs *et al.* [1976].

[King, 1969]  J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[Klein and Gamboa, 2014]  Gerwin Klein and Ruben Gamboa, editors. *Fifth International Conference on Interactive Theorem Proving, ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, Vienna, Austria, 2014. Springer-Verlag.

[Klein *et al.*, 2010]  Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an os kernel. *Communications of the ACM*, 53:107–115, 2010.

[Knaster, 1927]  B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1927. Volume published in 1928.

[Knoblock and Constable, 1986]  T. Knoblock and R. Constable. Formalized metareasoning in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 237–248, Cambridge, MA, USA, 1986. IEEE Computer Society Press.

[Knuth and Bendix, 1970] Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon Press, 1970.

[Krafft, 1981] Dean Blackmar Krafft. *AVID: a system for the interactive development of verifiably correct programs*. PhD thesis, Department of Computer Science, Cornell University, 1981.

[Krauss, 2010] Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44:303–336, 2010.

[Kühlwein *et al.*, 2012] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 378–392. Springer, 2012.

[Kühlwein *et al.*, 2013] Daniel Kühlwein, Stephan Schulz, and Josef Urban. E-MaLeS 1.1. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 407–413. Springer, 2013.

[Kumar and Weber, 2011] Ramana Kumar and Tjark Weber. Validating QBF validity in HOL4. In Eekelen et al. [2011], pages 168–183.

[Kumar *et al.*, 1991] Ramaya Kumar, Thomas Kropf, and Klaus Schneider. Integrating a first-order automatic prover in the HOL environment. In Archer et al. [1991], pages 170–176.

[Kumar *et al.*, 2014] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness and a verified implementation. In Klein and Gamboa [2014], pages 308–324.

[Kunčar, 2011] Ondřej Kunčar. Proving valid quantified boolean formulas in HOL Light. In Eekelen et al. [2011], pages 184–199.

[Kunen, 1998] Kenneth Kunen. Nonconstructive computational mathematics. *Journal of Automated Reasoning*, 21:69–97, 1998.

[Lakatos, 1976] Imre Lakatos. *Proofs and Refutations: the Logic of Mathematical Discovery*. Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar. Derived from Lakatos's Cambridge PhD thesis; an earlier version was published in the *British Journal for the Philosophy of Science* vol. 14.

[Lam, 1990] C. W. H. Lam. How reliable is a computer-based proof? *The Mathematical Intelligencer*, 12:8–12, 1990.

[Lamport and Melliar-Smith, 1985] Leslie Lamport and P. Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32:52–78, 1985.

[Landin, 1966] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[Lecat, 1935] Maurice Lecat. *Erreurs de Mathématiciens des origines à nos jours*. Ancne Libraire Castaigne et Libraire Ém Desbarax, Brussels, 1935.

[Leroy, 2009] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Letichevsky *et al.*, 2013] A.A. Letichevsky, A.V. Lyaletski, and M.K. Morokhovets. Glushkovs evidence algorithm. *Cybernetics and Systems Analysis*, 49(4):489–500, 2013.

[London, 1970] R. L. London. Computer programs can be proved correct. In *Proceedings of the IVth Systems Symposium at Cape Western Reserve University*. Springer-Verlag, 1970.

[Loveland, 1968] Donald W. Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM*, 15:236–251, 1968.

[Loveland, 1978] Donald W. Loveland. *Automated theorem proving: a logical basis*. North-Holland, 1978.

[Luo, 1989] Zhaohui Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 386–395. IEEE Computer Society, 1989.

[Luo, 1992] Zhaohui Luo. A unifying theory of dependent types: The schematic approach. In Anil Nerode and Michael A. Taitslin, editors, *Logical Foundations of Computer Science - Tver '92, Second International Symposium, Tver, Russia, July 20-24, 1992, Proceedings*, volume 620 of *LNCS*, pages 293–304. Springer, 1992.

[Luo, 2003] Zhaohui Luo. $PAL^+$: a lambda-free logical framework. *J. Funct. Program.*, 13(2):317–338, 2003.

[Lyaletski and Verchinine, 2010] Alexander V. Lyaletski and Konstantin Verchinine. Evidence algorithm and system for automated deduction: A retrospective view. In Autexier et al. [2010], pages 411–426.

[Mac Lane, 1986] Saunders Mac Lane. *Mathematics: Form and Function.* Springer-Verlag, 1986.

[MacKenzie, 2001] Donald MacKenzie. *Mechanizing Proof: Computing, Risk and Trust.* MIT Press, 2001.

[Magnusson and Nordström, 1993] 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, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *LNCS*, pages 213–237. Springer, 1993.

[Mamane and Geuvers, 2007] Lionel Mamane and Herman Geuvers. A document-oriented Coq plugin for TeXmacs. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *MKM 2007 - Work in Progress*, volume 07-06 of *RISC Report*, pages 47–60. University of Linz, Austria, 2007.

[Marcus *et al.*, 1985] Leo Marcus, Stephen D. Crocker, and Jalsook R. Landauer. SDVS: A system for verifying microcode correctness. *ACM Sigsoft Software Engineering Notes*, 10(4):7–14, 1985.

[Martin-Löf, 1984] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[Maslov, 1964] S. Ju. Maslov. An inverse method of establishing deducibility in classical predicate calculus. *Doklady Akademii Nauk*, 159:17–20, 1964.

[Matuszewski and Rudnicki, 2005] Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4:3–24, 2005.

[McAllester, 1989] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press, 1989.

[McCarthy, 1961] John McCarthy. Computer programs for checking mathematical proofs. In *Proceedings of the Fifth Symposium in Pure Mathematics of the American Mathematical Society*, pages 219–227. American Mathematical Society, 1961.

[McCune and Padmanabhan, 1996] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*, volume 1095 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[McCune, 1997] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19:263–276, 1997.

[McLaughlin and Harrison, 2005] Sean McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *CADE-20: 20th International Conference on Automated Deduction, proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 295–314, Tallinn, Estonia, 2005. Springer-Verlag.

[Mclaughlin *et al.*, 2005] Sean Mclaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *Proceedings of the 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, volume 144 of *Electronic Notes in Theoretical Computer Science*, 2005.

[McLaughlin, 2006] Sean McLaughlin. An interpretation of Isabelle/HOL in HOL Light. In Furbach and Shankar [2006], pages 192–204.

[McMillan, 2003] K. L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, Boulder, CO, 2003. Springer-Verlag.

[Megill, 1996] Norman D. Megill. Metamath: A computer language for pure mathematics. Unpublished; available on the Web from `ftp://ftp.shore.net/members/ndm/metamath.ps.gz`., 1996.

[Melham, 1989] Thomas F. Melham. Automating recursive type definitions in higher order logic. In Birtwistle and Subrahmanyam [1989], pages 341–386.

[Melham, 1992] Thomas F. Melham. The HOL logic extended with quantification over type variables. In Claesen and Gordon [1992], pages 3–18.

[Melliar-Smith and Rushby, 1985] P. Michael Melliar-Smith and John Rushby. The Enhanced HDM system for specification and verification. *ACM Software Engineering Notes*, 10(4), 1985.

[Meng and Paulson, 2006] Jia Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Succesful Computerized Reasoning*, 2006.

[Milner, 1972] Robin Milner. Implementation and applications of Scott's logic for computable functions. *ACM SIGPLAN Notices*, 7(1):1–6, January 1972.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Moore and Wirth, 2013] J Strother Moore and Claus-Peter Wirth. Automation of mathematical induction as part of the history of logic. Available from `http://arxiv.org/abs/1309.6226`, 2013.

[Moore *et al.*, 1998] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.

[Morgan, 1990] Carroll Morgan. *Programing from Specifications*. Prentice-Hall, 1990.

[Morse, 1965] A. P. Morse. *A theory of sets*. Academic Press, 1965.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.

[Myreen and Davis, 2014] Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound, down to the machine code that runs it. To appear in ITP 2014, 2014.

[Naumov *et al.*, 2001] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator (a practical approach to formal interoperability). In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2001.

[Naumowicz and Bylinski, 2002] Adam Naumowicz and Czeslaw Bylinski. Basic elements of computer algebra in MIZAR. *Mechanized Mathematics and Its Applications*, 2, 2002.

[Naumowicz and Bylinski, 2004] Adam Naumowicz and Czeslaw Bylinski. Improving Mizar texts with properties and requirements. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2004.

[Naur, 1966] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–216, 1966.

[Necula and Lee, 2000] George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In David McAllester, editor, *Automated Deduction — CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 25–44, Pittsburgh, PA, USA, 2000. Springer-Verlag.

[Nederpelt *et al.*, 1994] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer. *Selected papers on Automath*. Studies in logic and the foundations of mathematics. Elsevier, Amsterdam, 1994.

[Nelson and Oppen, 1979] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[Nelson and Oppen, 1980] Greg Nelson and Derek Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980.

[Newell and Simon, 1956] A. Newell and H. A. Simon. The logic theory machine. *IRE Transactions on Information Theory*, 2:61–79, 1956.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Nordström *et al.*, 1990] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[Obua and Skalberg, 2006] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Furbach and Shankar [2006], pages 298–302.

[O'Leary *et al.*, 1999] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://download.intel.com/technology/itj/q11999/pdf/floating_point.pdf`.

[O'Leary *et al.*, 2013] John O'Leary, Roope Kaivola, and Tom Melham. Relational STE and theorem proving for formal verification of industrial circuit designs. In Barbara Jobstmann and Sandip Ray, editors, *FMCAD 2013: Formal Methods in Computer-Aided Design*, pages 97–104. IEEE, 2013.

[Owre *et al.*, 1992] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Kapur [1992], pages 748–752.

[Papapanagiotou, 2007] Petros Papapanagiotou. On the automation of inductive proofs in HOL Light. Master's thesis, University of Edinburgh, 2007.

[Parrilo, 2003] Pablo A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96:293–320, 2003.

[Patkowska, 1969] Hanna Patkowska. A homotopy extension theorem for fundamental sequences. *Fundamenta Mathematicae*, 64(1):87–89, 1969.

[Paul, 2008] Wolfgang Paul. Towards a worldwide verification techology. In B. Meyer and J. Woodcock, editors, *Verified Software*, volume 4171 of *Lecture Notes in Computer Science*, pages 19–25. Springer-Verlag, 2008.

[Paulson and Blanchette, 2010] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Eugenia Ternovska, and Stephan Schulz, editors, *Proceedings of the 8th International Workshop on the Implementation of Logics*, pages 1–11, 2010.

[Paulson and Grąbczewski, 1996] Lawrence C. Paulson and Krzysztof Grąbczewski. Mechanizing set theory: Cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning*, 17:291–323, 1996.

[Paulson, 1983] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Paulson, 1987] Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Paulson, 1990] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. G. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*, pages 361–386. Academic Press, 1990.

[Paulson, 1994a] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Bundy [1994], pages 148–161.

[Paulson, 1994b] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.

[Paulson, 2003] Lawrence C. Paulson. The relative consistency of the axiom of choice mechanized using Isabelle/ZF. *LMS Journal of Logic and Computation*, 6:198–248, 2003.

[Petkovšek *et al.*, 1996] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger. *A = B*. A K Peters, 1996.

[Pfenning and Schürmann, 1999] Frank Pfenning and Carsten Schürmann. System Description: Twelf – A Meta-Logical Framework for Deductive Systems. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999.

[Pfenning, 1994] Frank Pfenning. Elf: A Meta-Language for Deductive Systems (System Descrition). In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *LNCS*, pages 811–815. Springer, 1994.

[Ploegaerts *et al.*, 1991] Wim Ploegaerts, Luc Claesen, and Hugo De Man. Defining recursive functions in HOL. In Archer et al. [1991], pages 358–366.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977.

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

[Pratt, 1975] Vaughan Pratt. Every prime has a succinct certificate. *SIAM Journal of Computing*, 4:214–220, 1975.

[Prawitz *et al.*, 1960] Dag Prawitz, Håken Prawitz, and Neri Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, 7:102–128, 1960.

[Queille and Sifakis, 1982] J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.

[Rajan et al., 1995] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof-checking. In Pierre Wolper, editor, *Computer-Aided Verification: CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, 1995. Springer-Verlag.

[Ramsey, 1926] F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society (2)*, 25:338–384, 1926.

[Reif, 1995] W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software — Final Report*, volume 1009 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Roxas, 1993] Rachel E. O. Roxas. A HOL package for reasoning about relations defined by mutual induction. In Joyce and Seger [1993], pages 129–140.

[Rudnicki and Drabent, 1985] Piotr Rudnicki and Wlodzimierz Drabent. Proving properties of Pascal programs in MIZAR 2. *Acta Inf.*, 22(3):311–331, 1985.

[Rudnicki and Trybulec, 1999] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999.

[Rudnicki and Trybulec, 2003] Piotr Rudnicki and Andrzej Trybulec. On the integrity of a repository of formalized mathematics. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2003.

[Rudnicki, 1987a] Piotr Rudnicki. Obvious Inferences. *Journal of Automated Reasoning*, 3(4):383–393, 1987.

[Rudnicki, 1987b] Piotr Rudnicki. Obvious inferences. *Journal of Automated Reasoning*, 3:383–393, 1987.

[Rushby and von Henke, 1991] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15. Association for Computing Machinery, 1991.

[Russell, 1919] Bertrand Russell. *Introduction to mathematical philosophy*. Allen & Unwin, 1919.

[Russinoff, 1998] David Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web at `http://www.russinoff.com/papers/k7-div-sqrt.html`.

[Sawada and Gamboa, 2002] Jun Sawada and Ruben Gamboa. Mechanical verification of a square root algorithms using Taylor's theorem. In M. Aagaard and John O'Leary, editors, *Formal Methods in Computer-Aided Design: Fourth International Conference FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 274–291. Springer-Verlag, 2002.

[Scheffer, 2003] Mark Scheffer. The Automath Archive, 2003.

[Schneider and Brandt, 2007] Klaus Schneider and Jens Brandt, editors. *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, Kaiserslautern, Germany, 2007. Springer-Verlag.

[Schulz, 2000] Stephan Schulz. *Learning search control knowledge for equational deduction*, volume 230 of *DISKI*. Infix Akademische Verlagsgesellschaft, 2000.

[Schulz, 2002] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[Scott, 1993] Dana Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Annotated version of a 1969 manuscript.

[Seger and Bryant, 1995] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.

[Seger and Joyce, 1991] Carl Seger and Jeffrey J. Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Department of Computer Science, University of British Columbia, 2366 Main Mall, University of British Columbia, Vancouver, B.C, Canada V6T 1Z4, 1991.

[Sewell et al., 2013] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, PLDI '13*, pages 471–482. Association for Computing Machinery, 2013.

[Shankar, 1994] N. Shankar. *Metamathematics, Machines and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[Shawe-Taylor and Cristianini, 2004] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[Shostak, 1984] Robert Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984.

[Siekmann and Wrightson, 1983] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning — Classical Papers on Computational Logic, Vol. I (1957-1966)*. Springer-Verlag, 1983.

[Slind, 1991] Konrad Slind. An implementation of higher order logic. Technical Report 91-419-03, University of Calgary Computer Science Department, 2500 University Drive N. W., Calgary, Alberta, Canada, TN2 1N4, 1991. Author's Masters thesis.

[Slind, 1996] Konrad Slind. Function definition in higher order logic. In Wright et al. [1996], pages 381–398.

[Slobodová, 2007] Anna Slobodová. Challenges for formal verification in industrial setting. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Proceedings of 11th FMICS and 5th PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2007.

[Smullyan, 1992] Raymond M. Smullyan. *Gödel's Incompleteness Theorems*, volume 19 of *Oxford Logic Guides*. Oxford University Press, 1992.

[Sokołowski, 1983] Stefan Sokołowski. A note on tactics in LCF. Technical Report CSR-140-83, University of Edinburgh, Department of Computer Science, 1983.

[Solovyev and Hales, 2011] A. Solovyev and T. Hales. Verification of bounds of linear programs. In James H. Davenport, William M. Farmer, Florian Rabe, and Josef Urban, editors, *Proceedings of CICM conference on intelligent computer mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 123–132. Springer-Verlag, 2011.

[Stålmarck and Säflund, 1990] Gunnar Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.

[Staples, 1999] Mark Staples. Linking ACL2 and HOL. Technical Report 476, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1999.

[Stickel, 1988] Mark E. Stickel. A Prolog Technology Theorem Prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[Syme, 1997] Donald Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.

[Tankink et al., 2010] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof re-animation. In Autexier et al. [2010], pages 440–454.

[Tankink et al., 2012] Carst Tankink, Christoph Lange, and Josef Urban. Point-and-write. In Jeuring et al. [2012], pages 169–185.

[Tankink et al., 2013] Carst Tankink, Cezary Kaliszyk, Josef Urban, and Herman Geuvers. Formal mathematics on display: A wiki for Flyspeck. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *MKM/Calculemus/DML*, volume 7961 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013.

[Tarski, 1936] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936. English translation, 'The Concept of Truth in Formalized Languages', in Tarski [1956], pp. 152–278.

[Tarski, 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tarski, 1956] Alfred Tarski, editor. *Logic, Semantics and Metamathematics*. Clarendon Press, 1956.

[Théry and Hanrot, 2007] Laurent Théry and Guillaume Hanrot. Primality proving with elliptic curves. In Schneider and Brandt [2007], pages 319–333.

[Trybulec, 1977] Andrzej Trybulec. Informationslogische sprache Mizar. *Dokumentation-Information: IX. Kolloquium über Information und Dokumentation vom 12. bis 14. November 1975*, (33):46–53, 1977.

[Trybulec, 1978] Andrzej Trybulec. The Mizar-QC/6000 logic information language. *ALLC Bulletin*, 6(2), 1978.

[Univalent Foundations Program, 2013] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[Urban and Sutcliffe, 2010] Josef Urban and Geoff Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in Mizar. In Autexier et al. [2010], pages 132–146.

[Urban et al., 2008] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008.

[Urban et al., 2010] Josef Urban, Jesse Alama, Piotr Rudnicki, and Herman Geuvers. A wiki for Mizar: Motivation, considerations, and initial prototype. In Autexier et al. [2010], pages 455–469.

[Urban et al., 2011] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP: Machine learning connection prover. In Kai Brünnler and George Metcalfe, editors, *TABLEAUX*, volume 6793 of *LNCS*, pages 263–277. Springer, 2011.

[Urban et al., 2013] Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 50:229–241, 2013.

[Urban, 2005] Josef Urban. XML-izing Mizar: Making semantic processing and presentation of MML easy. In Michael Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2005.

[Urban, 2006a] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 4(4):414 – 427, 2006.

[Urban, 2006b] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *Int. J. on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[Urban, 2008] Christian Urban. Nominal techniques in Isabelle/HOL. *JAR*, 40:327–356, 2008.

[Urban, 2012] Josef Urban. Parallelizing Mizar. *CoRR*, abs/1206.0141, 2012.

[Urban, 2014] Josef Urban. BliStr: The Blind Strategymaker. *CoRR*, abs/1301.2683, 2014. Accepted to PAAR'14.

[van Benthem Jutting, 1979] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system*. Number 83 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1979.

[van Dalen, 1981] Dirk van Dalen, editor. *Brouwer's Cambridge lectures on intuitionism*. Cambridge University Press, 1981.

[van der Voort, 1992] Mark van der Voort. Introducing well-founded function definitions in HOL. In Claesen and Gordon [1992], pages 117–132.

[Verchinine et al., 2008] Konstantin Verchinine, Alexander Lyaletski, Andrei Paskevich, and Anatoly Anisimov. On correctness of mathematical texts from a logical and practical point of view. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, AISC/Calculemus/MKM 2008*, volume 5144 of *Lecture Notes in Computer Science*, pages 583–598, Birmingham, United Kingdom, July 2008. Springer.

[Voelker, 2007] Norbert Voelker. HOL2P — a system of classical higher order logic with second order polymorphism. In Schneider and Brandt [2007], pages 333–350.

[Wang, 1960] Hao Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4:2–22, 1960.

[Weber and Amjad, 2009] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7:26–40, 2009.

[Weber, 2010] Tjark Weber. Validating QBF invalidity in HOL4. In Kaufmann and Paulson [2010], pages 466–480.

[Webster, 1995] Roger Webster. *Convexity*. Oxford University Press, 1995.

[Weis and Leroy, 1993] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: http://pauillac.inria.fr/caml/.

[Wenzel, 1999] Markus Wenzel. Isar - a generic interpretive approach to readable formal proof documents. In Bertot et al. [1999], pages 167–183.

[Wenzel, 2012] Makarius Wenzel. Isabelle/jEdit - a prover IDE within the PIDE framework. In Jeuring et al. [2012], pages 468–471.

[Weyhrauch and Talcott, 1994] Richard W. Weyhrauch and Carolyn Talcott. The logic of FOL systems: Formulated in set theory. In Neil D. Jones, Masami Hagiya, and Masahiko Sato, editors, *Logic, Language and Computation: Festschrift in Honor of Satoru Takasu*, volume 792 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 1994.

[Weyhrauch, 1980] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[Weyhrauch, 1982] Richard W. Weyhrauch. An example of FOL using metatheory. In Donald W. Loveland, editor, *Proceedings of the 6th Conference on Automated Deduction*, number 138 in Lecture Notes in Computer Science, pages 151–158, New York, 1982. Springer Verlag.

[Whitehead and Russell, 1910] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica (3 vols)*. Cambridge University Press, 1910.

[Wiedijk, 2000] Freek Wiedijk. CHECKER - notes on the basic inference step in Mizar. available at http://www.cs.kun.nl/∼freek/mizar/by.dvi, 2000.

[Wiedijk, 2001] Freek Wiedijk. Mizar light for HOL Light. In Richard J. Boulton and Paul B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 378–394. Springer-Verlag, 2001.

[Wiedijk, 2002] Freek Wiedijk. A New Implementation of Automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002.

[Wiedijk, 2006] Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[Wiedijk, 2007] Freek Wiedijk. Mizar's soft type system. In Schneider and Brandt [2007], pages 383–399.

[Wiedijk, 2009] Freek Wiedijk. Statistics on digital libraries of mathematics. In Adam Grabowski and Adam Naumowicz, editors, *Computer Reconstruction of the Body of Mathematics*, volume 18(31) of *Studies in Logic, Grammar and Rhetoric*. University of Białystok, 2009.

[Wiedijk, 2012a] Freek Wiedijk. Pollack inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012.

[Wiedijk, 2012b] Freek Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1), 2012.

[Wong, 1993] Wai Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993.

[Wos *et al.*, 1967] L. Wos, G. Robinson, D. Carson, and L. Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14:698–709, 1967.

[Wright *et al.*, 1996] Joakim von Wright, Jim Grundy, and John Harrison, editors. *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, 1996. Springer-Verlag.

[Wright, 1994] Joakim von Wright. Representing higher-order logic proofs in HOL. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 456–470, Valletta, Malta, 1994. Springer-Verlag.

[Wu, 1978] Wen-tsün Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:157–179, 1978.

[Young, 1993] W. D. Young. System verification and the CLI stack. In Jonathan Bowen, editor, *Towards Verified Systems*. Elsevier Science Publications, 1993.

[Zalewska, 2010] Gabriela Zalewska. Zamenhof, Ludwik (1859 – 1917). In *The YIVO Encyclopedia of Jews in Eastern Europe*. YIVO Institute for Jewish Research, 2010.

[Zammit, 1999] Vincent Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [1999], pages 185–202.

# INDEX