https://doi.org/10.1007/s00165-019-00492-1
The Author(s) © 2019
Formal Aspects of Computing (2019) 31: 675–698

Formal Aspects
of Computing

From LCF to Isabelle/HOL

Lawrence C. Paulson[1], Tobias Nipkow[2] and Makarius Wenzel[3]

[1] Computer Laboratory, University of Cambridge, Cambridge, UK
[2] Fakultät für Informatik, Technische Universität München, Munich, Germany
[3] Augsburg, Germany

Abstract. Interactive theorem provers have developed dramatically over the past four decades, from primitive beginnings to today's powerful systems. Here, we focus on Isabelle/HOL and its distinctive strengths. They include automatic proof search, borrowing techniques from the world of first order theorem proving, but also the automatic search for counterexamples. They include a highly readable structured language of proofs and a unique interactive development environment for editing live proof documents. Everything rests on the foundation conceived by Robin Milner for Edinburgh LCF: a proof kernel, using abstract types to ensure soundness and eliminate the need to store proofs. Compared with the research prototypes of the 1970s, Isabelle is a practical and versatile tool. It is used by system designers, mathematicians and many others.

Keywords: LCF, HOL, Isabelle, Interactive theorem proving.

1. Introduction

Today's interactive theorem provers originated in research undertaken during the 1970s on the verification of functional programs. Two quite different tools were built: the Boyer/Moore theorem prover (now ACL2, described elsewhere in this volume) and Edinburgh LCF [GMW79, Pau18].

Descendants of the latter include every member of the HOL family (HOL4, HOL Light, ProofPower) [GM93] as well as Coq [BC04] and Isabelle [NPW02]. As we shall see in the sequel, the achievements of the past 40 years lie on several dimensions, including type systems, proof languages and user interfaces. Automation is the key to usability; this includes automated search for counterexamples as well as for proofs.

These developments are reflected in the size of the tools themselves. In 1977, the Edinburgh LCF distribution was 900 KB, including an implementation of the ML programming language in Lisp. In 1986, the Isabelle distribution was a mere 324 KB (not counting ML, now a separate distribution). By 1991, Isabelle exceeded LCF: 937 KB. In 2019, the Isabelle distribution had reached 133 MB! And by way of comparison, HOL Light (which is much more closely related to LCF) was 84 MB. Much of this bulk consists of proof libraries rather than executable code, though libraries make a crucial contribution to a system's capabilities. Today's mature systems also include documentation and examples.[1]

Correspondence and offprint requests to: L. C. Paulson, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, England, UK. E-mail: lp15@cam.ac.uk

[1] The figures are for uncompressed distribution directories containing no binaries, but possibly PDF files.

Isabelle is a leading interactive theorem prover. It is generic, supporting a number of different formal calculi, but by far the most important of these is its instantiation to higher-order logic: Isabelle/HOL. Already during the 1990s, Isabelle/HOL was being applied with great success to the task of verifying cryptographic protocols [Pau98]. Turning to mathematics, it played a critical role in Hales's Flyspeck project, which verified his proof of the Kepler conjecture [HAB$^+$17]. It is the basis for the seL4 project, under which an entire operating system kernel was verified, proving full functional correctness [KAE$^+$10]. It was adopted by researchers outside the verification milieu for specifying and verifying algorithms for replicated datatypes that provide "eventual consistency" [GKMB17]. Numerous other projects are underway around the world. Like other proof assistants, Isabelle is not directly concerned with program verification, i.e. with verifying code written in a programming language, but it can be used as a back end to prove verification conditions. Isabelle can also be used as a verified programming environment, where mathematical functions can be proved correct and then automatically translated to executable code in one of several different programming languages. The translation process itself is currently unverified, but even this is likely to change in the near future.

This essay focuses on Isabelle/HOL and therefore has little to say about techniques common to most systems. For example, simplification by rewriting—coupled with recursive simplification to handle conditional rewrite rules—was already realised in both the Boyer/Moore theorem prover [BM79] and Edinburgh LCF by the end of the 1970s. Recursive datatype and function definitions, as well as inductive definitions, were commonplace by around 1995. Linear arithmetic decision procedures were also widely available by then.

Our title echoes Mike Gordon's paper "From LCF to HOL: a Short History" [Gor00]. Like Mike, we begin with LCF, the source of the most fundamental ideas. But we pass over this material quickly in order to focus on Isabelle. There is no way to surpass Mike's account of the early years. He starts in 1969, with Dana Scott's Logic for Computable Functions, and covers the original LCF project at Stanford University. He describes in detail the development of the successor LCF systems at Edinburgh and then at Cambridge. He names all the people involved and finally outlines his own development of HOL.

We begin with LCF because of its seminal importance, continuing to HOL because it is so strongly linked (Sect. 2). Then we focus exclusively on Isabelle. We begin with the core ideas of a generic reasoner built around unification and backtracking (Sect. 3). Then we consider the task of supporting higher-order logic, which required the introduction of type classes (Sect. 4). There follows an account of automatic proof search and its dual, the search for counterexamples (Sect. 5–6). We also discuss the generation of code from logical functions (Sect. 7). We then turn to Isabelle's most distinctive features: its structured proof language (Sect. 8) and the powerful user interface architecture supporting it (Sect. 9). The next section describes Isabelle's Archive of Formal Proofs (Sect. 10). To conclude, we discuss the synergy among these ideas, making them more powerful in combination than individually (Sect. 11).

Coq and other systems built around constructive type theories represent a distinctive strand of development and fall outside our scope.

2. LCF and HOL

Edinburgh LCF is best known for the so-called *LCF approach*: implementing the inference rules of a logical calculus within a *proof kernel* that has the exclusive right to create theorems. Such a kernel is indeed found in most modern systems, and is responsible for their good record of soundness. But in fact, LCF introduced a broader set of norms that are now taken for granted: a focus on backward proof, the practice of working in a theory hierarchy, and above all, the central role of a functional programming language, ML [Mil85].

ML is short for *meta language*, as it was the vehicle for operating on formulas belonging to the *object language* (namely, the proof calculus). Radical at the time, ML was soon seen to be a general-purpose programming language and today exerts a strong influence on language design. Crucial to ML is its sound, polymorphic type system with first-class functions, and in particular, its support for abstract types. An *abstract type* encapsulates a data structure's internal representation, allowing access only through a fixed set of functions. A typical abstract type might be a dictionary, where the implementation (e.g. balanced trees) is only accessible through well-defined operations such as insert, update and delete; then the implementer is free to change the representation (e.g. to improve performance or to introduce additional operations) without affecting any of the code outside the abstract data type declaration itself.

Robin Milner's key insight was that abstract data types could save space by eliminating the need to store proofs. (In his early experiments, he had kept running out of memory [Gor15].) He declared an abstract type of theorems, where the operations were simply the inference rules of his logical calculus. The resulting type, `thm`, was the type of theorems and ML's type checker was our guarantee that anything of type `thm` had definitely been created exclusively by the application of inference rules. Just as a dictionary doesn't need to keep a record of the operations applied to it but only the dictionary itself, type `thm` doesn't need to store the proofs of theorems but only their statements. The resultant savings of space are as important now as they were in the 1970s, for although today we have more memory, we also have vastly bigger proofs.

Edinburgh LCF also introduced proof *tactics* and *tacticals* to express backward, goal-directed reasoning. An LCF tactic typically applies a specific inference rule, while tacticals denote control structures, e.g., `THEN` (one tactic followed by another), `ORELSE` (which allows one of several tactics to be attempted) and `REPEAT` (repeated execution of a tactic until it fails). Remarkably, even tactics lie outside the proof kernel. An LCF tactic is a function that takes a goal (a formula with its assumptions) and returns a list of subgoals that it claims are logically sufficient. To justify this claim, it returns a function operating on values of type `thm`, but we have no guarantee that this function will deliver the promised theorem in the end. We see that the LCF architecture makes proof procedures harder to implement, while reducing the amount of code that has to be trusted.

We see incidentally two meanings of the word *proof*:

1. formal deductions of theorems from axioms using the inference rules of a logical calculus;
2. executable code written using tactics or other primitives, expressing the search for such deductions.

To resolve this ambiguity, the former are sometimes called *proof objects* or *proof terms* and the latter, *proof scripts* or *proof texts*. Thus we see that the LCF approach eliminates the need to store proof terms and allows proof scripts to be coded in ML using tactics and tacticals. Nevertheless, proof assistants based on constructive type theories retain proof objects, as they are intrinsic to such formalisms.

The most fundamental question in the design of a theorem prover is what calculus to support. Boyer and Moore made the inspired choice of Pure Lisp, which was sufficient for verifying simple functional programs and formalising elementary number theory in the 1970s, and which has grown to support advanced applications today. For Edinburgh LCF, Milner chose the Logic for Computable Functions, which was perfect for the domain-theoretic investigations topical at the time but proved to be too quirky for general adoption.

Mike Gordon, who was one of the designers of Edinburgh LCF, ultimately adopted higher-order logic as the basis for his research into hardware verification [Gor86, Gor00]. Higher-order logic is most easily understood as a typed predicate calculus including function and Boolean types, and therefore also set types. Gordon launched his HOL system in 1986 [Gor86, GM93], presenting both the logic and its application to hardware specification and verification. HOL turned out to be extremely versatile and it soon attracted a global user community. New versions emerged, in particular HOL Light [Har96]. Members of the HOL family have been used in verification projects of every description, including the formalisation of great bodies of mathematics.

However, it wasn't obvious in the 1980s that one or two formalisms could be sufficient for the needs of verification. Type theories, dynamic logics and many other formalisms were being proposed. Gordon himself had reworked LCF twice in his hardware verification research. This was the origin of the idea that a theorem prover could be *generic*: supporting a spectrum of formalisms through a common framework. Today this includes syntactic tools (parsing, pretty printing), inference tools (rewriting, unification), a common proof language (Isar) and user interface foundation (PIDE).

3. Isabelle in the early days: a logical framework

Isabelle originated in a project to build an LCF-style proof assistant for Martin-Löf's constructive type theory [ML84]. Two ideas influenced the design from the outset [Pau90]:

- Reasoning should be based on *unification* rather than pattern matching, so that goals could contain variables that could be instantiated.
- A tactic should be able to return multiple results in a lazy list, representing alternative proof attempts for *backtracking*.

Both ideas were aimed at supporting proof search. The LCF work had demonstrated conclusively that verification was repetitious and tedious, requiring the best possible automation. The combination of unification and backtracking would allow the use of logic programming techniques, as in Prolog [CM87]. For example, in Martin-Löf's type theory we have the following (derived) rule:

$$\frac{c \in A \times B}{\mathrm{fst}(c) \in A.}$$

Backward chaining, using this rule to prove something of the form $\mathrm{fst}(c) \in A$, leaves us with a subgoal of the form $c \in A \times ?B$, where the question mark indicates that $?B$ is a hole to be filled: in Prolog parlance, a logical variable. Unification fills in the holes later, while backtracking can orchestrate alternative ways of filling the holes. In this example, the form of c will determine the choice of $?B$. The ability to defer choices until they become clear is as valuable in theorem proving as it is in Prolog programming.

It may be worth mentioning that Prolog was a hot topic in the 1980s. However, standard first-order unification was simply not applicable to Martin-Löf type theory, which is based on a form of typed λ-calculus. *Higher-order unification* would be necessary. Undecidable in its full generality, a reasonably practical unification procedure had recently been published by Gerard Huet [Hue75].

Higher-order unification is strikingly different from first-order unification. The latter is easily implemented and delivers a unique answer modulo variable renaming. Higher-order unification allows even functions to be variables and can instantiate such variables with λ-abstractions generated on the fly, yielding multiple results. For example, it is possible to unify $F\,M$ with 3 in two different ways: $F = \lambda\, x.\, 3$ (with no constraint on M) and $F = \lambda\, x.\, x$, $M = 3$. Unifying $F\,3$ with $3 + 3$ could yield four different results: $F = \lambda\, x.\, x + x$, $F = \lambda\, x.\, x + 3$, $F = \lambda\, x.\, 3 + x$ and $F = \lambda\, x.\, 3 + 3$. The search for such unifiers works by descending through the terms one level at a time, attempting two kinds of steps: *projections* (use of a bound variable) and *imitations* (copying the opposite term). The examples above show how it can find different ways of abstracting one term from another.

With such a sophisticated syntactic mechanism built in at the lowest level, it became clear that the LCF approach to inference rules could be radically changed [Pau86]. Consider the proof rule of universal elimination:

$$\frac{\forall\, x.\, \phi(x)}{\phi(a)} \tag{1}$$

Its LCF representation is an ML function taking two arguments: a theorem (which must have the form $\forall\, x.\, \phi(x)$) and a term a (which must have the same type as x). It then generates the desired conclusion, namely $\phi(a)$, raising an exception if any of the preconditions is violated. This approach is general; the drawback is the tedium and attendant risk of error when there are dozens of rules. With our λ-calculus framework, inference rules such as the one above can simply be written out in the form of a template and instantiated using unification. Better still, unification could be applied either to the premise or to the conclusion, yielding forward or backward proof through a single mechanism. This is Isabelle's central idea [Pau86].

The precise nature of these templates remained to be determined, and the best approach turned out to be a sort of *logical framework*. These are specialised formalisms whose purpose is to encode other formalisms. The Edinburgh Logical Framework [HHP93] is the best known of these. It differs from Isabelle's by incorporating proof objects into the calculus itself, neutralising one of the key advantages of the LCF architecture: that proofs do not have to be stored.

So Isabelle can be seen as an instance of the traditional LCF approach, but where type `thm` formalises a logical framework or *meta-logic*, where other formalisms (the *object-logics*) can later be encoded. Such encodings can also be proved correct [Pau89]. Our logical framework approach is less general than the original LCF representation, where an inference rule can undertake an arbitrary computation. Nevertheless, it captures a variety of possibilities, and moreover, it is open-ended: for example, Isabelle defines intuitionistic first-order logic, which in a succession of formal theories is extended with classical logic, then with Zermelo–Fraenkel set theory, then with the axiom of choice. With the original LCF approach, once you define the type `thm`, it can never be extended.

The first object-logic was constructive type theory (Isabelle/CTT). It was followed by a classical first-order sequent calculus (Isabelle/LK) and by natural deduction calculi for intuitionistic and classical first-order logic (Isabelle/IFOL and Isabelle/FOL) [Pau93]. Later, a substantial development of Zermelo–Fraenkel set theory was developed on top of Isabelle/FOL [PG96]. This is one of the leading tools for formal reasoning in axiomatic set theory.

Although Isabelle took a declarative approach to defining logics, the system architecture was still based on LCF's philosophy that everything was done in ML. All interactions with Isabelle took place at the ML toplevel. For example, the declaration of new types, constants with their definitions, and axioms all required calling appropriate ML functions with any necessary data as arguments. This began to change in the early 1990s [Pau94] when logics could be defined in a *theory file* with types, constants and proof rules declared using a natural syntax.[2] Proofs, however, were still expressed in ML. The structured proof language Isar (Sect. 8.1) came later. Of today's LCF-based systems, the HOL family has remained the most faithful to the original conception, with proofs coded in ML.

Paulson's original implementation of higher-order unification [Pau86] was practical, but still slow. Yet in Isabelle practice, many unification problems are first-order, or almost so. Dale Miller [Mil91] discovered a subclass of λ-terms, later called (higher-order) patterns [Nip91a], which behave like first-order terms: unification is decidable and if two terms are unifiable, they have a *most-general unifier*. A term (in β-normal form) is called a *higher-order pattern* if every free occurrence of a variable has as arguments a list of distinct bound variables. Most unification problems in Isabelle are of this form. Nipkow gave a succinct implementation of pattern unification and added it to Isabelle [Nip93a]. Full higher-order unification is invoked only if pattern unification encounters a non-pattern.

A special case of unification is matching where the variables of only one of the two terms are instantiated. Isabelle's rewrite engine (aka the *simplifier*) is based on higher-order pattern matching. Thus the simplifier can deal with many standard transformations of quantified terms, for example the following ones:

$$\begin{aligned}(\forall x.\, P(x) \wedge Q(x)) &= (\forall x.\, P(x)) \wedge (\forall x.\, Q(x))\\ (\forall x.\, P \vee Q(x)) &= P \vee (\forall x.\, Q(x))\\ (\forall x.\, x = t \wedge P(x)) &= P(t)\end{aligned}$$

It appears that Isabelle was the first theorem prover to support higher-order rewrite rules [NP98].

4. Type classes and Isabelle/HOL

Gordon's HOL system became a runaway success, dominating the verification arena. By 1991 it was being used in over 80 separate projects around the world [Kal91]. Isabelle claimed to be a generic theorem prover, but it couldn't handle higher-order logic. The problem was that the templates mentioned above could not refer to types. This was not an issue in the original application of Martin-Löf type theory, where types and formulas were effectively identified and the inference rules referred to types explicitly. In the following example, A and $A + B$ are types and the rule expresses how type checking should be done for a term of the form inl(a).

$$\frac{a \in A}{\mathrm{inl}(a) \in A + B}$$

Contrast with the previous inference rule (1), where no types are visible. There are many formalisms, including many-sorted first-order logic[3] as well as higher-order logic, where types are kept in the background and type constraints are enforced implicitly. Users would not like to be forced to prove statements like $i + 1 : \mathrm{int}$.

Adequate support for logics having implicitly-typed variables required another idea, order-sorted polymorphism, leading to an even more powerful idea, axiomatic type classes.

4.1. Order-sorted polymorphism

Isabelle had always supported polymorphism internally; the difficulty lay in making it available to users. In its simplest form, a polymorphic type variable may take on any type whatsoever. This could not be allowed in a logical framework, where some types are intrinsic to the framework itself and other types might be unsuitable to a particular object-logic. In first-order logic, the type of x in $\forall x.\, \phi(x)$ must not involve functions or Booleans.[4]

[2] This represented a return to the original Edinburgh LCF, which also supported theory files. The Isabelle version was inspired by Goguen's OBJ system [Gog79, FGJM85], in particular concerning the declaration of new mixfix syntax.

[3] Beware of terminological confusion regarding the word *sort*, which for first-order logic is synonymous with *type*. Our use of *sort* below will be entirely different.

[4] Quantification over Booleans (and therefore, relations) requires at least second-order logic [Bar77, p. 7].

Order-sorted polymorphism solves such difficulties by introducing a hierarchy of sorts on types [Nip91b]. A *sort* is a finite intersection of *type classes*, which are essentially collections of types. For first-order logic, we may introduce the type class *FO* for all types for which quantification is permitted. For higher order logic, we would have a different type class (say *HO*), containing Boolean and function types. We write $\tau :: C$ if type τ has class C.

Now we need to express how (possibly nullary) *type constructors* such as *nat* (the type of natural numbers), Cartesian product, function space and *list* (the type of lists) act on type classes. For example, for first-order logic we want that *nat* :: *FO* and that if τ :: *FO* then τ *list* :: *FO*. These assertions can be codified in Isabelle as instance declarations:

instance *nat* :: *FO*
instance *list* :: (*FO*) *FO*

Thus *nat list* :: *FO* holds, but in the absence of corresponding instances, *bool* :: *FO* and *bool list* :: *FO* do not: truth values, or lists of them, do not belong to *FO*. Technically, these instance declarations form the *signature* of the (term) algebra of types. Under certain natural conditions on the interaction of the class hierachy with the instance declarations, order-sorted unification of types, like unsorted unification, is still *unitary*: solvable unification problems have *most-general unifiers*. Therefore these conditions guarantee that we still have *principal types*. This theory, combined with extensions to higher-order unification [Nip91b], allows full support for higher-order logic. And so the "templates" of our framework contain explicit type variables, in addition to the ordinary variables that we had before. This version was first released in 1991 [NP92]. At the same time, the correspondence to Haskell's type classes was worked out [NS91, NP93].

4.2. Axiomatic type classes

The full power of type classes is realised when they are combined with axioms. *Axiomatic type classes* support the flexible and principled overloading of symbols [Nip93b, Wen97]. A type class can be introduced on the basis of a specific vocabulary of symbols (or *signature*) possibly coupled with axioms (a *specification*) to constrain those symbols—and possibly extending other type classes. Orderings are a natural example: Isabelle/HOL introduces a succession of type classes for increasingly stronger notions of ordering:

- *ord*: the ordering symbols $<$, $\leq$, but with no attached properties
- *preorder*: adding reflexivity and transitivity, and defining $x < y$ iff $x \leq y \land \neg(y \leq x)$
- *order*: adding antisymmetry
- *linorder*: adding linearity

For the case of lists, the lexicographic ordering yields a partial ordering if the list elements are partially ordered and yields a linear ordering if the list elements are linearly ordered. Such details are easily expressible:

instance *list* :: (*order*) *order*
instance *list* :: (*linorder*) *linorder*

Instance declarations for axiomatic type classes require the user to supply definitions of any associated symbols (here $<$ and $\leq$) for the supplied type along with proofs of the associated properties using those definitions. This overloading is principled in that although $<$ and $\leq$ will have separate definitions for each type for which they are defined, they will always satisfy the associated axioms.

The value of axiomatic type classes can be seen in the formalisation of mathematical analysis. Type classes for groups, rings and other algebraic structures provide overloading for the common arithmetic symbols, but in addition, we have type classes for metric and topological spaces, vector and Euclidean spaces. When we construct the type of complex numbers for example, by showing that they form a field and a Euclidean space, we instantly inherit substantial libraries of facts covering limits, convergence, derivatives and topology, which would otherwise have to be largely duplicated from the analogous facts for the real numbers [HIH13]. The same thing happens again when we construct more advanced number systems, such as the quaternions [Woo18], a number system for three-dimensional space. We have a surprising variety of numeric types, such as the extended reals ($\mathbb{R} \cup \{+\infty, -\infty\}$) and the nonnegative extended reals; both of these belong to a number of ordering and topological type classes.

An exposition of axiomatic type classes with fully worked out examples is available in a paper by Paulson [Pau04]. It's slightly outdated—we have a much more elaborate type class hierarchy now and a somewhat different syntax—but the principles are the same.

4.3. Logical foundations

Wenzel [Wen97] reduced type classes to (1) overloaded constant definitions (OCDs) for the signature and (2) predicate definitions over a single type variable for the specification. Getting the notion of definition with type-polymorphism right is notoriously tricky. For example, in an early version of Gordon's HOL system, constant definitions could introduce inconsistencies [Art16]. The problem is simply to detect ill-formed definitions, in particular circularities, automatically. Circularities can lead to inconsistencies. Type classes link the level of types with the level of terms, opening up new opportunities to create highly obscure circularities.

Wenzel analysed the impact of OCDs on the consistency of an arbitrary theory. He sketched conditions under which they were *meta-safe*, which roughly means that they can be removed without affecting provability. Meta-safety implies conservativity, which in turn implies consistency preservation. However, his proof sketch considered an idealized version of Isabelle's actual definition facilities, which were only partially implemented and in conflict with existing application theories. In particular—in order to stay within Gordon's HOL and the Isabelle logical framework—he excluded the interleaving of OCDs with type definitions.

Obua [Obu06] found that Isabelle accepted OCDs that introduced inconsistencies. He gave a more rigorous and general formulation of Wenzel's conditions [Wen97] and implemented them using an external termination checker, outside of the inference kernel. He also sketched a proof that these conditions ensured conservativity. As an alternative, Wenzel and Haftmann proposed a much simpler (and stronger) check on OCDs [HW06] inside the logical kernel, but this excluded a few application theories with ambitious overloading.

Several years later Kunčar and Popescu [KP19, KP18] rediscovered that combining overloaded constant definitions with type definitions could introduce circularities that were missed by the previous analyses. They gave extended checks to avoid these circularities and proved that these checks ensure consistency of theories that extend the initial HOL theory (comprising the standard HOL axioms) with overloaded constant definitions and type definitions. In the end [KP18], they strengthened the result to show that such definitional extensions are in fact meta-safe over the initial HOL theory.

These particular concerns were specific to detecting circular definitions, but all proof assistants potentially contain errors, like any other software. A notable case is PVS [Owr06], which lacking an LCF-style proof kernel was particularly vulnerable to soundness errors (at least in the 1990s):

> PVS still seems to contain a lot of bugs and frequently new bugs show up. . . . It would be desirable that the bugs in PVS would only influence completeness and not soundness. Unfortunately, this is not the case, as some recent proofs of `true==false` have shown [15, bug numbers 71, 82, 113 and 160]. . . . It is reasonable to assume that PVS will continue to contain soundness bugs. The obvious question thus arises, why use a proof tool that probably contains soundness bugs? Our answer is threefold: PVS is still a very critical reader of proofs. PVS lets fewer mistakes slip through than many of our human colleagues. . . . Furthermore, history tells us that the fixed soundness bugs are hardly ever unintentionally explored, we know of only a single case. Thirdly, most mistakes in a system that is to be verified are detected in the process of making a formal specification. [GH98, pp. 134–135]

Notwithstanding such forbearance, users deserve a much higher standard of correctness than this. At the same time, it's vital to stress that users need to take responsibility for their own definitions: we know how to detect circularity, but no system can check whether a definition conforms to the user's true intentions.

4.4. Isabelle/HOL versus HOL

As mentioned at the start of this section, the purpose of the foregoing work was to make possible an Isabelle instantiation of higher-order logic. Isabelle/HOL emerged in 1991 [NP92] and was soon a fully capable alternative to HOL [NPW02]. In some ways it emulated the latter, particularly in its axiom system. Nevertheless, it is thoroughly Isabelle in its treatment of theorems, tactics, etc; we cannot regard Isabelle/HOL as a member of the HOL family the way that HOL Light is.

Both Isabelle and HOL implement Milner's idea of a proof kernel implementing a formal calculus as an abstract type called `thm`. The essential difference is that Isabelle's formal calculus is a logical framework in which other formalisms can be defined, while HOL's is simply higher-order logic. Either way, a function of type `thm->thm` implements an inference rule. In the case of Isabelle, this would be an operation at the level of the logical framework, such as the resolution rule through which proofs are constructed. In the case of HOL, this would be an inference in higher-order logic itself, like the quantifier rule (1). In HOL, a tactic must be coded separately corresponding to each rule of inference, while in Isabelle, inference rules such as (1) are all expressed declaratively and applied using the generic resolution rule.

Isabelle's generic nature can be seen in how many of its capabilities are shared among its various instances. The common libraries include a general representation of syntax, with parsing, pretty printing, type checking, simplification and other proof procedures, as well as the proof language and user interface. They are available in other instances of Isabelle, such as Isabelle/ZF, which has been the basis for substantial developments in Zermelo–Fraenkel set theory [Pau03].

5. Automation

It was clear from the outset that machine proof was extremely laborious and could only be feasible if the machine itself provided as much automation as possible. But it was also clear that fully automatic theorem proving was not practically achievable. Significant advances in automation had already been made by the end of the 1970s:

1. decision procedures for arithmetic, arrays, lists, etc., as well as methods for combining decision procedures [NO80];
2. resolution for first-order logic [Rob65, Ove75], complete in principle but frequently disappointing in practice;
3. the signature automation of the Boyer/Moore theorem prover [BM79]: conditional rewriting plus powerful heuristics for induction.

Of these, simplifiers based on rewriting (by previously proved theorems of the form $t = u$) quickly found their way into LCF, HOL, etc. All such simplifiers eventually supported *conditional rewriting*, for rewrites of the form $\phi \implies t = u$; in such a case, the necessary instance of ϕ would be proved recursively by the simplifier itself. Arithmetic decision procedures were eventually adopted in many systems. On the other hand, resolution had acquired a bad reputation. As late as 2002, Shankar could write

> The popularity of uniform proof methods like resolution stems from the simple dogma that since first-order logic is a generic language for expressing statements, generic first-order proof search methods must also be adequate for finding proofs. This central dogma seems absurd on the face of it. . . . A more sophisticated version of the dogma is that a uniform proof method can serve as the basic structure for introducing domain-specific automation. There is little empirical evidence that even this dogma has any validity. . . . Automated reasoning has for too long been identified with uniform proof search procedures in first-order logic. This approach shows very little promise. [Sha02, pp. 3–4]

Of the various proof assistants, only Isabelle saw a sustained effort to incorporate ideas from resolution.

5.1. The classical reasoner

As mentioned in Sect. 3 above, Isabelle supported both unification and backtracking from the start, with the aim of incorporating ideas from first-order automatic proof procedures. In the context of interactive proof, unification provided the ability to prove a subgoal of the form $\exists\, x.\, \phi(x)$ by removing the quantifier and proving $\phi(?t)$, where $?t$ stood as a placeholder for a concrete term to be supplied later. Through unification, this term could even be built up incrementally. Dually, unification provided a means of using a universally quantified fact $\forall\, x.\, \phi(x)$, when the required instances were not immediately obvious.

Simple automation is achievable through a combination of obvious applications of the propositional connectives ($\wedge$, $\vee$, $\neg$, etc.) along with heuristics for performing quantifier reasoning. Stronger automation is obtainable by borrowing well-known techniques for classical first-order logic theorem proving. But the most important idea is to embrace the concepts of natural deduction in application theories as well as in pure logic. Natural deduction prefers the use of simple inference rules focusing on a single symbol. For example, conjunction is effectively defined by the following three rules:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \qquad \frac{\phi \wedge \psi}{\phi} \qquad \frac{\phi \wedge \psi}{\psi}$$

The intersection of two sets has a technical definition that would greatly complicate reasoning, but it is easy to derive inference rules for intersection in the style of natural deduction (and analogous to those above):

$$\frac{a \in A \quad a \in B}{a \in A \cap B} \qquad \frac{a \in A \cap B}{a \in A} \qquad \frac{a \in A \cap B}{a \in B}$$

Many other reasoning steps can be expressed similarly:

$$\frac{A \subseteq B \quad a \in A}{a \in B} \qquad \frac{A \subseteq B \quad B \subseteq A}{A = B} \qquad \frac{k \ \mathbf{dvd}\ m \quad k \ \mathbf{dvd}\ n}{k \ \mathbf{dvd}\ \mathrm{gcd}(m, n)}$$

So the crucial idea is to build proof tactics that support reasoning with inference rules of this sort. Though they borrow techniques from first-logic theorem provers, they are far more effective than expanding the various operators into their low-level definitions and attempting to prove the resulting formulas of pure logic. And note: they would be formulas of higher-order logic, where automation is considerably more difficult than for first-order logic.

Such tactics have collectively become known as Isabelle's *classical reasoner* [Pau99]. A principle of natural deduction is that the syntactic form of each rule clearly identifies which symbol it is concerned with. And therefore hundreds of such rules can coexist in the classical reasoner without causing a combinatorial explosion. The user who invokes *auto*—which combines classical reasoning and simplification—gains the benefit of built-in knowledge about everything in Isabelle's standard libraries. As users build their own libraries, they can continue to augment this knowledge. This combination of classical reasoning with rewriting is still unique to Isabelle.

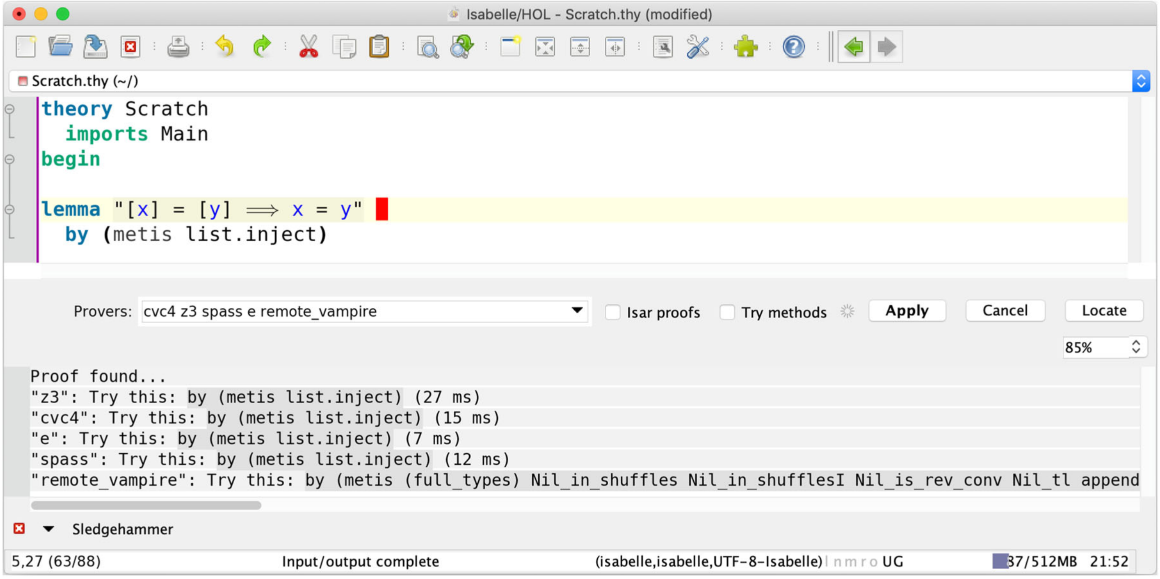
5.2. Sledgehammer

By the early 2000s, resolution theorem provers [RV02, Sch04, Wei01] were demonstrating power far beyond anything the classical reasoner could ever achieve. The idea of some sort of interface between Isabelle and these systems was beguiling. This idea wasn't new: such linkups have been attempted on a number of past occasions, always unsuccessfully. The key was to make such a linkup useful:

> The guiding idea is that user interaction should be minimal. The system should invoke automatic provers spontaneously or in response to a trivial gesture such as a mouse click. These proof attempts should run in the background, not disturbing the user unless a proof is found. Proofs should refer to a large library of known lemmas: users should not have to select the relevant ones. The automatic prover should not be trusted; instead, proofs should be translated back into the formalism of the interactive prover. Proofs should be delivered in source form to the user, who can simply paste them into her proof script. [MQP06, p. 1576]

A major difficulty with building an interface between Isabelle/HOL and first-order automatic theorem provers is that they operate on quite different formalisms. Isabelle/HOL has λ-abstractions, types and type classes, while first-order logic has none of these. One click invocation could only be achieved if the system itself took care of everything: the translation of higher-order syntax, some representation of type constraints, etc. An additional requirement was *relevance filtering*: to identify the most suitable of the thousands of facts available in an Isabelle session, since providing too many would overwhelm the first-order provers.

The final difficulty was to translate the proofs discovered by the external reasoning tools into Isabelle's proof kernel. This translation moreover had to be expressed as a source-level proof so that the expensive proof search would not have to be repeated. Resolution theorem provers typically print a formal justification of their reasoning, but it is difficult to interpret and frequently ambiguous.



```
theory Scratch
  imports Main
begin

lemma "[x] = [y] ⟹ x = y"
  by (metis list.inject)
```

Provers: cvc4 z3 spass e remote_vampire

☐ Isar proofs ☐ Try methods

```
Proof found...
"z3": Try this: by (metis list.inject) (27 ms)
"cvc4": Try this: by (metis list.inject) (15 ms)
"e": Try this: by (metis list.inject) (7 ms)
"spass": Try this: by (metis list.inject) (12 ms)
"remote_vampire": Try this: by (metis (full_types) Nil_in_shuffles Nil_in_shufflesI Nil_is_rev_conv Nil_tl append
```

Fig. 1. Sledgehammer GUI in Isabelle/jEdit: clicking on highlighted output inserts the proposed proof snippet into the text

The approach eventually adopted was to extract from their output nothing but the list of axioms—each a known Isabelle theorem—used in the proof. This list would typically contain no more than eight axioms, yielding a problem simple enough to be proved by simple tactics integrated with Isabelle's kernel [PS07]. This tool became known as *Sledgehammer*. The original version was developed at Cambridge, but Sledgehammer was comprehensively rewritten at Munich, particularly by Blanchette [BBP13], who greatly increased its scope and power.

Sledgehammer helps beginners by identifying and using theorems that they didn't know existed. It puts the latest theorem provers, such as Vampire and Z3 [dMB08], at their disposal with a single click (Fig. 1). But even advanced users are often surprised by the proofs Sledgehammer comes up with. It is now seen as indispensable, and similar subsystems are being developed for other interactive theorem provers.

6. Counterexample search

Isabelle's proof methods and Sledgehammer are effective for proving *theorems*, but given an invalid conjecture they normally fail to detect the invalidity, let alone produce an informative counterexample. Novices and experts alike state invalid theorems and find themselves wasting hours on impossible proofs. To make proving more enjoyable and productive, Isabelle includes counterexample generators that complement the proof tools. The main ones are Quickcheck and Nitpick. As a simple example, suppose that the user types in this lemma statement (where *rev* reverses a list and @ appends two lists):

lemma $rev\ (xs\ @\ ys) = rev\ xs\ @\ rev\ ys$

Quickcheck is invoked automatically and displays the counterexample $xs = [a_1],\ ys = [a_2]$.

6.1. Quickcheck

As the name suggests, Quickcheck is Isabelle's counterpart to the QuickCheck testing tool [CH00] for Haskell. The original Quickcheck [BN04] combined Isabelle's code generation infrastructure (Sect. 7) with random testing, covering both recursive functions and inductive predicates. It aimed at providing fully automatic counterexample search, in contrast to Haskell's QuickCheck, which is an infrastructure for building specialized random generators for testing. Therefore it worked automatically whenever all functions and inductive predicates were executable, but, depending on the property, it might take forever to find a counterexample. Consider especially conditional properties $\phi \Longrightarrow \psi$ where most of the random values simply falsify ϕ; this situation requires a special random generator that yields only values that satisfy ϕ.

Therefore Bulwahn [Bul12a, Bul12b, Bul12c, BBN11] refined Quickcheck in three directions:

- Exhaustive enumeration of small values, an idea due to Runciman et al. [RNL08].
- Generating values not randomly but synthesizing generators from the premises of conditional properties.
- Symbolic testing with *narrowing*, a technique from *functional-logic programming* [AH10].

Isabelle's Quickcheck was inspired by the work of Dybjer et al. [DHT03] in the theorem prover Agda: they had followed the original QuickCheck design, which expected users to set up specialized random generators in the host language/logic. Over the next decade, other major theorem provers adopted analogous checkers: PVS [Owr06], ACL2 [CDKM11] and Coq [PHD$^+$15].

Quickcheck is one of Isabelle's best-loved tools (after Sledgehammer), partly because it is invoked automatically and silently every time the user types in a lemma. You may not even be aware of its existence and it suddenly announces that it has found a counterexample to your purported lemma. The result is a surprised and grateful user. Quickcheck is most effective in the context of functional programming combined with inductive predicates.

6.2. Nitpick

A radically different approach to Quickcheck is based on systematic model enumeration using a SAT solver. This approach was pioneered by the tool Refute [Web05, Web08] and is now embodied by Nitpick [BN10, Bla12]. Nitpick looks for finite fragments (substructures) of infinite countermodels, soundly approximating problematic constructs. Common Isabelle idioms, such as inductive and coinductive predicates and datatypes as well as recursive and corecursive functions, are treated specially to ensure efficient SAT solving [Bla13]. The actual reduction to SAT is performed by the Kodkod library [TJ07]. Given a conjecture, Nitpick (via Kodkod and the SAT solver) searches for a standard set-theoretic model that falsifies it while satisfying any relevant axioms and definitions. Nitpick is innately better suited to problems from set theory and logic than Quickcheck. Nitpick revels in particular in finite combinatorial problems.

The first tool that exploits SAT-solving for finding counterexamples in a theorem prover (ACL2) seems to be due to Sumners [Sum02]. A second prototype tool, again for ACL2, was developed by Spiridonov and Khurshid [SK07] and was based on Kodkod. Blanchette was unaware of this work when developing Nitpick.

7. Code generation

Code generation is the process of generating efficiently executable code from definitions in the logic of a theorem prover. It serves three main purposes: to obtain actual runnable software, to validate definitions by executing them on concrete values, and to search for counterexamples to properties by testing.

Most major theorem provers are based on logics that have an executable sublanguage. In the Boyer and Moore theorem prover, from its earliest incarnation [BM79] to present-day ACL2 [HKMS17], the logic is itself a purely functional fragment of Lisp: all expressions can be executed according to Lisp semantics. In Martin-Löf type theory [ML84], the term language—including proof terms—has a well-defined operational semantics. Terms are also executable, subject to certain conditions, in other type-theory based systems such as Coq [BC04]. Note that the types (which correspond to the formulas of predicate logic) are not executable.

In the case of HOL, a sublanguage can be identified that corresponds to a functional programming language. Then we superimpose an operational semantics on this fragment: programs are sets of equations that are to be used as rewrite rules, from left to right. This idea goes back to term rewriting and was expressed succinctly by titles like *Computing in Systems Described by Equations* [O'D77] and *Programming with Equations* [HO82]. The step-by-step execution of an equational program by rewriting corresponds to performing a proof in equational logic. If the rewrite rules come from recursive function definitions in the first place, it is natural to translate them to programs in an ML-like functional language with pattern-matching. The first such tool was created by Rajan [Raj93] for Gordon's HOL system [GM93].

7.1. History

Berghofer and Nipkow [BN02] realised the approach above, creating a compiler in Isabelle from a subset of HOL into ML. Datatypes and recursive functions defined by pattern matching are translated directly into their ML counterparts. They also extended the approach by making a subset of inductive predicates executable. The idea is to view them as Prolog programs and to perform a *mode analysis* [DW88]. Each possible mode partitions the arguments of an inductive predicate into inputs and outputs, and for each mode the inductive predicate is compiled into an ML function that maps an input tuple to a stream of output tuples. A subset of HOL terms and certain queries involving inductive predicates can now be compiled and executed in ML. This formed the basis of Isabelle's initial Quickcheck (Sect. 6.1).

Isabelle's code generator is part of the trusted kernel. Work by Haftmann [Haf09] and others improved its reliability and functionality significantly: code generation for inductive predicates was moved out of the kernel by translating inductive predicates into recursive functions inside HOL, where the equivalence is proved [BBH09]. Reliability of the compilation of the purely functional sublanguage was improved by a pen-and-paper correctness proof [HN10]. Code generation was extended to type classes by eliminating them in a first step. Code generation was extended to support data refinement [HKKN13], i.e. the automatic implementation of abstract types like sets by concrete types like search trees. Due to a new modular design, further target languages are easier to add: as of this writing, the code generator supports Standard ML, OCaml, Haskell and Scala.

Most recently, Hupel [HN18] has provided an alternative verified code generator that translates HOL into CakeML, an ML-like functional language with a verified compiler [KMNO14]. CakeML is the only backend for code generation that doesn't have to be trusted, since the CakeML compiler has itself been verified formally. This yields a verified tool chain from HOL to machine code, except that the CakeML compiler was verified in HOL4 [SN08] rather than Isabelle/HOL. Eliminating this gap is the subject of current research.

7.2. Applications

The code generator has been an enabling technology for a large number of applications. The following are some representative examples:

Imperative HOL Bulwahn et al. [BKH+08] developed a monadic embedding of imperative programs in HOL. That is, on the HOL level everything is still purely functional, but in a monadic style. They extended the code generator such that it translates these monadic definitions into actual imperative code in the target language (SML etc.). This leads to substantially improved performance and is used in a number of applications below.

Refinement Framework Lammich [Lam13, Lam16, Lam19] has developed an HOL framework for the stepwise refinement of (possibly nondeterministic) algorithms down to (possibly imperative) executable code. This framework is used in a number of applications below. Lammich and others [LT12, Lam14, LS19] have used the framework extensively for the verification of efficient graph algorithms.

Model Checking and SAT Solvers Esparza et al. [ELN+13] developed an executable verified model LTL model checker (10–50 times slower than SPIN [Hol97] on standard benchmarks) that was later extended with partial order reduction [BL18]. Brunner and Lammich built on work by Peled [Pel96] but found that one of his lemmas was incorrect; thus they were unable to use his actual reduction algorithm. Siegel [Sie19] found a counterexample to the correctness of Peled's algorithm with the help of the Alloy analyzer [Jac06].
Wimmer and Lammich [WL18] developed an executable model checker for timed automata whose throughput is about one order of magnitude lower than Uppaal's [LPY97] on standard benchmarks (but degenerates for large state spaces).
Lammich [Lam17] verified an executable checker for unsatisfiability certificates emitted by SAT solvers which is twice as fast as the standard unverified checker.

Term Rewriting Thiemann has been developing a huge formalisation of the theory of term rewriting called *IsaFoR*/*CeTA*[5] over more than a decade now [TS09]. Initially, IsaFoR/CeTA was aimed primarily at checking termination proofs found by automatic tools like AProVE [GAB+17]. The code generator produces these proof checkers from their verified HOL formalizations. Today, IsaFoR/CeTA can also check proofs of a term rewriting system's complexity [DJK+18] and confluence [NM16].

Computer Algebra As representative examples we mention decision procedures for univariate real polynomials (based on Sturm [Ebe15] and on cylindrical algebraic decomposition [LPP19]) and the Berlekamp-Zassenhaus factorization algorithm [DJTY19].

[5] http://cl-informatik.uibk.ac.at/software/ceta/.

Programming Languages One key application of theorem provers has been the formalization of programming languages and compilers [Ler09, KMNO14, NK14]. The code generator was used by Lochbihler and Bulwahn [LB11] to generate an interpreter directly from the semantics of a Java-like language with threads.

8. Structured proofs, structured specifications and formal contexts

In the beginning, formal proofs were tiny and the ML proof scripts were readable enough. A typical proof goal involved just a few assumptions involving two or three bound variables. But as the field progressed, researchers tackled increasingly ambitious problems. The longer proofs got, the more incomprehensible they became. Moreover, the traditional LCF approach of tactical proof had a tendency to retain too much, so that the user might be faced with a list of several dozen assumptions. These were not merely overwhelming to the eye but caused automatic proof procedures to bog down: their execution time could rise exponentially.

The Isar proof language, introduced in the late 1990s, addressed these concerns by allowing proofs to be structured into nested scopes. Local goals were proved from local assumptions, which were written out explicitly. Block structure is well understood in computer science, but here the ability to make declarations locally had to be retrofitted into Isabelle. This led to the idea of local contexts to encapsulate the assumptions (and associated bound variables) specific to a particular goal, along with other information. During an ambitious proof, the user may still have several dozen assumptions available, but these are now structured through the nesting of the contexts and accessible by name rather than in a single giant list.

Locales are a further structuring mechanism for expressing an extended series of proofs that rest on shared assumptions. They are useful for developing abstract mathematics, such as group theory, which can then be applied to particular groups.

8.1. Structured proofs: the Isar language

A distinctive feature of Isabelle is its Isar language of structured proofs [Wen07]; the acronym stands for *Intelligible semi-automated reasoning*. In the original LCF paradigm, a proof could be arbitrary ML code, which in some later systems was replaced by a command language for proofs (e.g. the Ltac scripting language in Coq). The problem with these traditional approaches is that somebody looking at a machine proof can have no idea what is being proved at a given point: it is like playing blindfold chess. In contrast, an Isar proof is a hierarchical structure containing explicit statements of assumptions and conclusions, with an indication of the use of local facts. Isar also provides some mechanisms to avoid redundancy: it allows the proof author to achieve a good balance of readability versus maintainability, such that small changes to definitions and theorem statements should lead to reasonably small changes to proofs.

Here is a tiny Isar proof that implicitly uses some derived rules for logical connectives taken from the library (similar to the classical reasoner from Sect. 5.1):

```
have A ∧ B ⟶ B ∧ A
proof
  assume ∗: A ∧ B
  show B ∧ A
  proof
    from ∗ show B ..
    from ∗ show A ..
  qed
qed
```

Here we prove a formula, $A \wedge B \longrightarrow B \wedge A$. We first assume $A \wedge B$, giving it the label $*$. Then we show $B \wedge A$, treating B and A separately. Those subproofs refer to the assumption via its label.

The Isar approach scales from a few primitive inferences, as above, to large proof developments involving heavy automated reasoning tools, allowing the user to control the extent of proof automation. The proof engine is able to check well-structured Isar proofs more efficiently than traditional tactic scripts: the hierarchical structure helps to keep internal goals concise, without the intrusion of redundant assumptions or unused lemmas. Moreover, the compositionality of Isar proofs, together with the proof irrelevance of the Isabelle framework, allows independent checking of sub-proofs—even with parallel checking on multiple cores enabled by default, due to Matthews and Wenzel [MW10, Wen13b].

The Isar proof language reuses some ideas from Mizar, a legendary software tool for doing mathematics by machine [GKN15]. But the many impressive facilities of the Mizar language are intertwined with its unusual set theoretic formalism, making the key ideas difficult to extract, especially given Mizar's notorious lack of documentation and closed sources. In contrast, Isar's generic principles of proof—fixing local variables and making local assumptions—are identified with the corresponding elements of Isabelle's logical framework. Thus it works with any object-logic that uses the builtin Natural Deduction paradigm of Isabelle, e.g. HOL, FOL, ZF. Various derived Isar language elements provide explicit proof structure for typical reasoning seen in object-logics: existential elimination and case-splitting, calculational chains of equalities and inequalities (including substitution), structured induction etc. All of this is parameterised by declarations in the library.

The Isar proof language is parametrised by user-defined Isar *proof methods*, which are the old idea of ML tactics fitted into the richer structure of the Isar engine. Thus new reasoning patterns may be added to a theory library without revisiting the design of the proof language itself. For example, the Isabelle standard library defines method *rule* for declarative forward/backward chaining explained in Sect. 3, and method *auto* for a combination of the simplifier and the classical reasoner explained in Sect. 5.1. Proof methods are either defined in ML (as in LCF or HOL), or in the Eisbach language [MMW16] (similar to Ltac definitions in Coq): Eisbach uses the source notation of existing proof methods to define new ones via simple recursion and pattern matching.

Despite its importance for the Isabelle ecosystem, the Isar proof language has not been adopted by other interactive theorem provers. There are no fundamental obstacles to doing so, but it takes some effort to do properly. Some isolated aspects of Isar have made it into the SSReflect language for Coq [GM10], notably the `have` keyword for local claims within a proof. A bit more Isar syntax made it into the Lean prover [dMKA+15] with slightly different meaning, though. Lean's `fix`, `assume`, `have`, `show`, `obtain` construct certain λ-terms in a more elementary manner than the Isabelle/Isar proof context export and goal refinement operations; e.g. see the treatment of **fix-assume-show** in [Wen07, §2.2]. The experiments by Wiedijk towards supporting structured proofs in HOL Light [Wie01] are not directly related to Isar. They are simply a family of HOL Light tactics that allow a proof script written in ML to have some similarity to a Mizar text. This style of working has not caught on in the HOL world.

Although some detailed aspects of Isar are specific to Isabelle—forward/backwards refinement via higher-order unification (Sect. 3) and local proof contexts (Sect. 8.3)—the language design as a whole is generic. It requires nothing from the underlying calculus but primitive notions of terms and types and logical notions equivalent to "implies" and "for all". It's not too hard to envisage a common language to be shared among proof assistants, offering portability of proofs exactly as today's programming languages offer portability between different machine architectures. The crucial missing ingredient is the ability to write assertions and prove subgoals while minimising explicit references to a particular calculus.

8.2. Global theory context

The original LCF approach (Sect. 2) declares an abstract ML type `thm` of theorems, but the background theory is implicit in the system state: during a session there is only one big theory under development, which grows monotonically. There are primitives to introduce new types, terms, definitions and axioms which augment the theory, but without any way to undo such a change. HOL provides some tricks to hide unwanted constants via name-space manipulation, to help interactive development.

In contrast, as early as 1986, Isabelle declared the abstract ML type `theory` alongside the type of theorems. Type `theory` makes available the initial theory of the logical framework and allows for its extension as an acyclic graph of application theories: there are operations to extend and merge theories. Multiple theories can coexist in a single Isabelle/ML session, but end-users only work with a single theory document at a time. This may import other theory documents—in foundational order from bottom to top. The concrete syntax looks like this:

theory *Test*
  imports *Main HOL-Library.Finite_Map HOL-Library.Finite_Lattice*
begin
  definition *constant* = *term*
  theorem *name*: *statement* $\langle proof \rangle$
end

Here we declare a new theory, called *Test*. It is built upon three other theories: *Main*, *Finite_Map* and *Finite_Lattice*. The last two names are qualified with the Isabelle *session* to which they belong: *HOL-Library*. Qualified names eliminate the danger of name clashes between theories belonging to different sessions. Between the **begin** and **end** brackets we can make definitions and prove theorems.

Proved results are formally *certified* against their original theory context, e.g. a theorem $\Theta \vdash \phi$ for theory Θ, and are implicitly propagated to another theory $\Theta' \vdash \phi$, provided that Θ' extends Θ by construction. For efficiency, this theory relation is implemented via symbolic *stamps* that represent definitions, proofs, etc., extending a theory: actual theory content is not compared. Stamp inclusion needs to be checked in every inference step: its complexity is logarithmic in the number of theory extensions, which can be many thousands in typical applications. These mechanisms lie within the logical kernel.

In early versions of Isabelle as a logical framework, theories coincided with object-logics plus some examples on top of them: Isabelle/FOL, Isabelle/ZF, Isabelle/HOL etc. Derivations in different branches of the theory graph could coexist in a single file without interference. Today, theories are usually derived from the *Main* entry-point of Isabelle/HOL and built as a natural hierarchy according to the structure of the application. It helps users to organise formal definitions and proofs like consecutive chapters in a book; it also helps the system for parallel checking of independent paths in the theory graph. Figure 2 shows the theory hierarchy of the Isabelle/HOL number theory development. At the top are sessions such as *Pure* and *HOL-Library*; in the middle are theories such as *Totient*, all of which are imported into *Number_Theory*.

Isabelle theory operations are purely functional updates, making undo trivial: the earlier versions continue to exist and can be returned to. This also includes add-on content like the ML environment or hints for proof tools: thus the theory context provides a default set of parameters, according to the imports from the library.

In summary, Isabelle theories provide large-scale structure to formalisation projects, with a built-in notion of monotonic reasoning over an acyclic graph of theory nodes.

8.3. Local proof context

LCF and the HOL family lack an explicit notion of proof context. There are usually some auxiliary structures to manage the current proof state (maybe just a list of subgoals) for interactive theorem proving. However, proof tools cannot refer to those; they merely see an isolated goal. Such a goal is essentially a sequent: a list of formulas (the assumptions) paired with another formula (the goal itself). Proof assistants based on dependent type theories (like Coq) do have a formal context Γ that declares variables, but this again is essentially a sequent. When applying HOL tactics it is frequently necessary to re-state the types of variables present in the statement of the theorem being proved, and one can even introduce two variables called x with different types. So at its most basic, we need a context to associate a type with each variable involved in the current proof. In fact, they do much more.

A genuine *Local Proof Context* implemented by the ML type `Proof.context` first appeared in Isabelle99, as infrastructure for the then emerging Isar proof language (Sect. 8.1). The idea is to support a block-structured *notepad* with recursive nesting of local declarations (e.g. type parameters, term parameters, assumptions) and local conclusions that are generalized when leaving a nested block: there is a generic *export* operation to move results from a nested context into the enclosing context: this usually turns context elements into the rule structure of the Isabelle logical framework (the quantifier $\bigwedge$ and connective $\Longrightarrow$). In concrete syntax, this looks as follows:

```
notepad
begin
  {
    fix x y z
    assume A x and B y
    have C x y z  ⟨proof⟩
  }
  note ‹⋀x y z. A x ⟹ B y ⟹ C x y z›
end
```

Here the final **note** recalls the result from the preceding proof block (enclosed in the curly brackets). This block introduces three bound variables, assumed to be fixed and to satisfy $A\ x$ and $B\ y$; from these we prove $C\ x\ y\ z$. The effect of this block is to prove the formula shown in the note.

The **notepad** leaves nothing behind in the theory context: it is merely an experiment. In contrast, a **theorem** statement produces an initial proof context with a pending claim: the theorem to be proved.

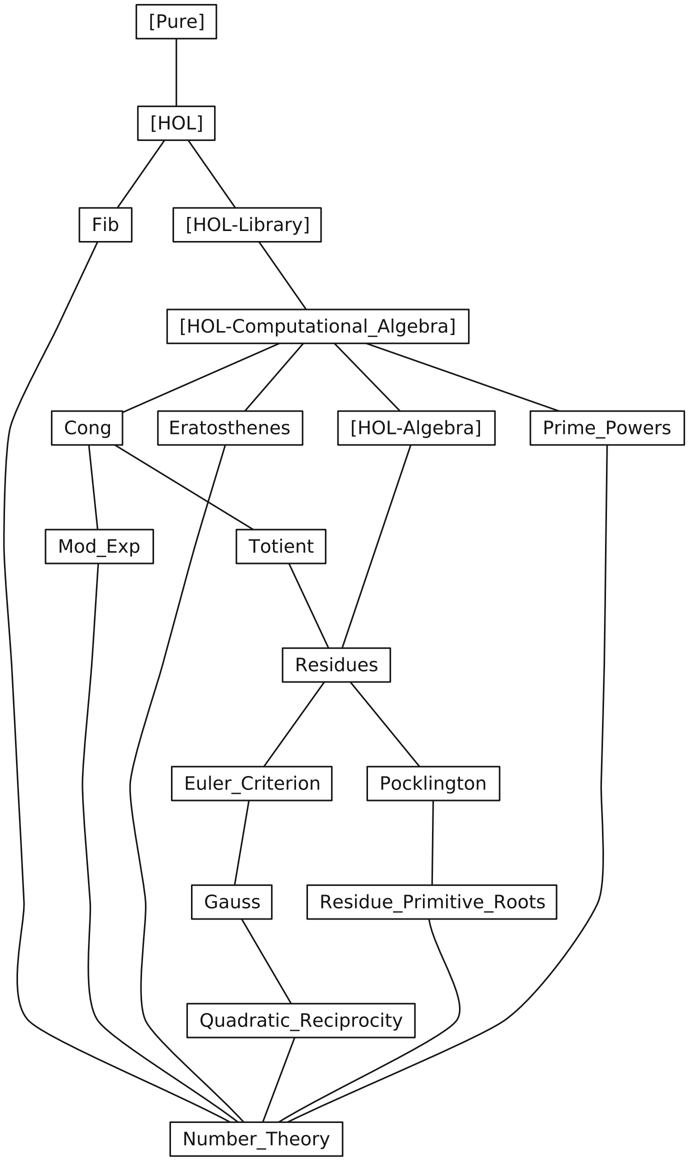


Fig. 2. The theory dependencies of session *HOL-Number_Theory*

The subsequent proof body needs to solve that in the context, potentially with further nesting of local contexts and auxiliary claims. The final result extends the enclosing theory (Sect. 8.2) by the new fact.

Proof tools may access the `Proof.context` value at each point. It contains the logical content (parameters, assumptions, proved facts), local syntax, and hints to guide proof strategies. Thus the universal `Proof.context` replaces Isabelle's earlier tool-specific contexts like `simpset` for the simplifier or `claset` for the classical reasoner. This uniformity is important to combine tools: e.g. the simplifier uses the context to extract its own information (rewrite rules and auxiliary proof procedures); during the simplification process it augments the context and passes it on to other procedures, which in turn extract their own information from it (e.g. for classical reasoning). Users can even extend the context with additional components to support proof procedures specific to their own applications. Contexts provide extensibility along with modularity.

In summary, a proof context in Isabelle represents a local situation derived from the enclosing theory. Nested contexts appear and disappear; only exported results remain. For proof tools, the context is a universal environment for storing tool-specific information.

8.4. Structured specifications: locales

Isabelle locales provide an infrastructure for structured specifications: definitions, statements and proofs of a theory may depend on *local parameters* (type and term variables) and *local premises* (hypotheses). The subsequent example specifies partial orders axiomatically and defines a derived operation and theorem in that specification context:

locale *partial_order* =
  fixes *le* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sqsubseteq$ *50*)
  assumes *refl*: $x \sqsubseteq x$
   and *trans*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
   and *antisym*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$

This introduces a new locale, called *partial_order*. The locale declares *le*, effectively a constant to which we attach infix syntax. The locale asserts what are effectively axioms. But in reality, a locale abbreviates a predicate taking in this case a single argument, asserting that it satisfies the given assumptions. The point is to allow this small specification to be imported simply by quoting the name *partial_order*.

definition (**in** *partial_order*)
  less :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sqsubset$ *50*)
  where $x \sqsubset y \longleftrightarrow x \sqsubseteq y \wedge x \neq y$

theorem (**in** *partial_order*)
  less_le_trans: $x \sqsubset y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubset z$ $\langle proof \rangle$

Now we can make definitions and prove theorems with respect to this locale, its constants and assumptions visible within these declarations.

Locales may be combined via *locale expressions*, to rename or instantiate parameters and merge contexts. Locale *interpretation* imports a given instance of a locale expression into an application context: after proving the locale assumptions as theorems, all conclusions of the locale context become available as facts. Here is an example that instantiates abstract partial orders to natural numbers:

interpretation *nat*: *partial_order* $(\leq) :: nat \Rightarrow nat \Rightarrow bool$
  rewrites *nat.less* $\equiv (<) :: nat \Rightarrow nat \Rightarrow bool$ $\langle proof \rangle$

Thus *le* is instantiated as the standard order on type *nat*, and the derived operation *less* is identified with the corresponding strict order on *nat* (imposing a proof obligation that needs to be proven together with the other locale axioms). Afterwards, all conclusions from the *partial_order* context become available in terms of these native signatures on *nat*.

Locales are ubiquitious in Isabelle theory development today, but the majority are actually defined as type classes (Sect. 4). These concepts started out independently, but were unified by Haftmann and Wenzel [HW06]. This is another application of the locale interpretation concepts, which are due to Ballarin [Bal06].

There are many ways of structuring mathematical developments via locales. One is simply to package up a group of related assumptions that are needed in a long series of proofs: this avoids cluttering up the theorem statements with this common material. Moreover, locales are more general than type classes. A type can belong to a class in only one way, so duality arguments (reversing the direction of a partial order for example) are out of the question; locales however can be instantiated in multiple ways at the same time. And while the type class for groups imposes the group structure on entire types, a locale for groups can have the carrier of the group as an explicit component of the locale; it can therefore be an arbitrary set. This generality is necessary to develop algebra properly. Isabelle's *HOL-Algebra* session develops a substantial amount of group theory using this approach.

Ballarin [Bal14] provides a comprehensive overview of the concepts, and a tutorial on locales is part of the standard Isabelle documentation. Many people have participated in the development of locales over almost 15 years. The original work by Kammüller, Wenzel, and Paulson [KWP99] directly uses primitives of the Isabelle framework; important abstractions on top of proof contexts (Sect. 8.3) were introduced by Haftmann and Wenzel [HW09].

LCF did not have any concept comparable to locales. But the need for structuring mechanisms has been evident for a long time, and a variety of ideas were tried in systems as diverse as AUTOMATH, Coq and HOL. The original locale concept was inspired by experiments (since abandoned) that had been done in HOL using higher-order predicates. Coq has a richer logic with a built-in notion of "sections": the original Isabelle locales from 1999 [KWP99] were inspired by that, but without augmenting the logical framework of Isabelle. This design principle of building higher concepts without extending the existing logic was later adopted by Coq: its type classes are built on top of existing concepts like records, predicates, and implicit arguments.

8.5. ML within the logical context

LCF and HOL can be seen as being nothing but libraries of proof procedures written in ML. The user can call those procedures via the ML toplevel but does not have to prove theorems and could instead, say, calculate π to 10,000 digits. In this sense, using LCF or HOL is the same as working with any other subroutine library.

This type of ML toplevel no longer exists in Isabelle. Instead, ML has become a sub-language of the framework. Syntactically, ML expressions (or whole modules) may appear within the theory and proof language of Isabelle/Isar: for example, in the command **method_setup** to define a new proof method. Semantically, program snippets may depend on symbolic entities from the logical context: types, terms, facts, etc. The ML compiler is invoked at run-time within an augmented environment; it refers to logical entities as well as its own environment (for ML types, values, modules). The static result is an updated ML environment within the context: thus it also conforms to the parallel evaluation model of Isabelle/ML.

Specialised proof procedures can be implemented in this manner within the normal theory document, using the regular Prover IDE (see Sect. 9). This integration of programming with logic works without augmenting the logic: like in original LCF, ML has access to the implementation of the object-logic, but is not part of the logical formalism. This is in contrast to Coq, where users often implement proof tools inside the logical language itself (with correctness proofs), but genuine extensions in OCaml need to be assembled outside the system as "plugins".

9. Document-oriented interaction: the prover IDE

The now ubiquitous WIMP interface (windows, icons, menus, pointer) emerged in the late 1970s, around the same time as LCF. Many observers suggested that the tedium of theorem proving could be addressed by involving those new ideas. However, typical suggestions did not address the real difficulties. A favourite was to let users point to terms that must be rewritten. But in most situations it is infinitely more effective to execute such steps automatically, driven by an algorithm. Think of dragging 10,000 files one-by-one to a trash can icon when they could be deleted by a single command specifying a pattern. Many suggestions were geared to the needs of novices rather than to the professionals who would be the main users.

Useful interfaces for theorem provers would not appear for 20 years. What users really needed, it seems, was the ability to survey the situation around them: what has been proved, what remains to be proved, what theorems are available, where and how they were proved, etc.

9.1. Prover interfaces: the early days

The original LCF proof assistant from 1979 used a line-oriented terminal or teletype. This model is known as a read-eval-print loop (or REPL): the user types one command after another, reacting on output printed by the prover. When computer screens and multiple windows arrived, there was often a split into two areas: the editor to work on a growing "proof script" and the terminal with the REPL to update its state, using manual copy-paste operations from the editor.

Around 1998, the highly influential Proof General interface for Emacs was released for the first time [Asp00], including support for Coq and Isabelle. Here a refined model of copy-paste and state-synchronization is baked into the editor (which is freely programmable in Lisp): the user can move a frontier of already checked text either *forwards* (apply command) or *backwards* (undo command); only the unchecked part may be freely edited. Proof General requires suitable undo operations of the prover, and for robustness it is better to have a restricted command-language instead of arbitrary ML. Both are missing in HOL (any version), which consequently still uses the original prover REPL, with some support through Emacs.

Today, Coq remains as the main back-end for Proof General Emacs, there is also a popular Proof General clone written in OCaml: CoqIde. In contrast, Isabelle discontinued both the REPL and its Proof General mode in 2014: interaction now works exclusively via the document-oriented Prover IDE.

9.2. The Isabelle prover IDE (PIDE)

From 2008, multi-threaded ML programming became routinely available in Isabelle, for parallel processing of theories and proofs within a single Poly/ML process. This posed some challenges to the robustness and performance of the prover engine, addressed subsequently by Matthews and Wenzel [Wen13b].

Parallel processing is also in conflict with the traditional interaction model: the REPL acts like a single focus of single-threaded command application. In order to remove many built-in assumptions of sequential evaluation from the interaction model and to provide rich semantic information in proof authoring process, Wenzel introduced the document-oriented Prover IDE (PIDE) approach [Wen11, Wen13a, Wen14, Wen19]. The main principles of PIDE are as follows:

- The **prover** supports *document edits* and *markup reports* natively. Interaction works via protocol commands (like `Document.update`) that take regular prover commands as data (e.g. **definition**, **theorem**). It has its own policies to process proof documents in parallel, according to the structure of the text.
- The **editor** connects the physical world of *editor input events* and *GUI painting* to the mathematical document-model of the prover. There are pipelines to stream input and output events asynchronously, with explicit identification of document versions.
- Add-on **tools** may participate in the ongoing document processing by conventional means, as isolated functions from input to output that are managed by PIDE. External tools merely need to ensure that interrupts work correctly: this is required when the user continues editing and old versions of the document are discontinued eventually.

Unlike Proof General, PIDE never locks the source text: edits by the user may lead to instantaneous updates by the prover, or significant delays for slow proof tools. Thanks to overall performance improvements of Isabelle and its underlying Poly/ML implementation, this ambitious interaction model works smoothly.

Isabelle/PIDE is delivered to end-users as a fully integrated desktop application called Isabelle/jEdit: it is based on the Java-based text editor jEdit (see http://www.jedit.org). This explains the initial motivation to use the Java platform for the outwards facing side of PIDE, which is implemented in Scala (on the JVM). Non-Java editors (e.g. VSCode) may be connected to PIDE via an extra socket connection that exchanges JSON records. There is also a *Headless PIDE* server with a similar protocol; this allows PIDE to run under control of another program.

Generally speaking, Isabelle/ML works best for pure applications of mathematical logic inside the prover, but Isabelle/Scala allows us to connect to the physical world: IDE front-ends, database engines, TCP services etc. Such technologies are not available in the same quality in Standard ML, nor even in OCaml (which underlies HOL Light and Coq).

10. The archive of formal proofs

Proof libraries are of enormous importance to formal verification. They are analogous to software libraries, facilitating reuse and eliminating the need to construct everything from scratch.

The Isabelle distribution already comes with a basic collection of more than 700,000 lines (38 MB) of Isabelle/HOL theories. On top of it sits the *Archive of Formal Proofs* (AFP, see https://www.isa-afp.org), a large online collection of proof developments contributed by the Isabelle community, all of them (as of this writing) for Isabelle/HOL. Each entry or *article* is a collection of Isabelle theories. It is the sole shared library of the Isabelle community. The AFP was launched in 2004 and at the time of writing contains 480 articles written by 322 authors. These comprise 2,250,000 lines (152 MB) of Isar text proving 134,000 theorems and supporting lemmas. It covers both computer science and mathematics.

As with a scientific journal, there is a small editorial board and submissions are reviewed for proof style and relevance. The AFP is an online resource and therefore more dynamic than a normal scientific journal. Articles can and do evolve. This conflicts with the purpose of archiving entries as they have been submitted and with the purpose of providing a stable interface to users. However, true preservation requires ensuring that entries continue to work as Isabelle itself evolves. The AFP deals with this conflict as follows. For each Isabelle release there is a corresponding AFP release. There is a separate development version of the AFP that is updated by Isabelle developers and AFP authors: Isabelle developers maintain all entries to be up to date with the current Isabelle development version; authors can update their articles monotonically by adding further material while ensuring that all entries that depend on theirs still work.

As Isabelle evolves, the self-imposed requirement to maintain all AFP articles in working order puts a burden on the Isabelle developers. At the same time, it acts as a reality check, helping developers to evaluate the impact of their changes on the user community. Isar's structured proofs assist the maintenance effort by localising the impact of changes. Sledgehammer is also invaluable when fixing unfamiliar proofs.

The model for the AFP is the *Mizar Mathematical Library* [Miz, BBG+18] which was started in 1989. The statistics today are similar: 250 authors, 1300 articles, 60,000 theorems, 3,000,000 lines (97 MB) of text. Of course one has to take into account that the languages and logics are different and Mizar has less proof automation.

11. Postscript: synergy between ideas

We have looked at a wide variety of ideas: a proof kernel written in a functional language; a logical framework to support multiple formalisms; polymorphism and type classes; advanced forms of automation; a structured proof language; a unique Prover IDE. Having seen the ideas in isolation, it's worth looking at how they work in combination.

Sometimes one idea led to another straightforwardly. Unification and backtracking were included by design to support future automation. The low-level primitives of the logical framework (implication and universal quantification) provided the right foundation for the Isar language precisely because they were the most fundamental logical concepts.

In other cases, the synergy between ideas could not have been predicted. It's remarkable that the early decision to adopt a polymorphic functional programming language (ML) turned out to be crucial 40 years later: the Isabelle Prover IDE is intertwined with pervasive parallelism. The Isabelle code base was not purely functional, but it was close enough, and the few sections that were necessarily imperative could be isolated easily. This yields an impressive speed up in multicore environments, a machine architecture nobody could have expected in 1975.

Another example of synergy is between automation (notably Sledgehammer) and the Isar language. The latter allows us to write a derivation as a chain of simple steps, which can be proved automatically by the former. If a link of this chain needs a different sort of proof, such as induction, a nested scope can be inserted on the spot. This technique is particularly helpful to beginners, who would otherwise have to learn a great many specialised proof methods for transforming one formula into another, as with other proof assistants. But even experts don't have to think so hard when writing proofs, which are also easy to read because the chain of steps is written out explicitly. Powerful automation means the proofs don't have to be too detailed.

The earliest design decisions, dating from Edinburgh LCF, still make sense 40 years on. Our choice of a polymorphic functional language, a minimal proof kernel and no stored proofs yields good performance with a minimum risk of soundness errors; contrast that with alternative choices in many other automated theorem provers. Our focus on pervasive automation and readability must be contrasted with the prevailing tendency for low level, "write only" proofs. Our varied ideas have produced a system that looks like a unified whole, despite being the product of many people's contributions[6] over several decades.

Acknowledgements

We thank the referees, Jasmin Blanchette, Michael Norrish and Andrei Popescu for valuable comments on drafts of this paper. The work reported above was funded by the British EPSRC, the German DFG and various European Union funding agencies.

[6] Included in each release is a file entitled CONTRIBUTORS, but it only goes back to 2005 and has many omissions.

References

[AH10] Antoy S, Hanus M (2010) Functional logic programming. Commun ACM 53(4):74–85

[Art16] Arthan R (2016) On definitions of constants and types in HOL. J Autom Reason 56(3):205–219

[Asp00] Aspinall D (2000) Proof general: a generic tool for proof development. In: Graf S, Schwartzbach M (eds) European joint conferences on theory and practice of software (ETAPS), vol 1785 of LNCS. Springer

[Bal06] Ballarin C (2006) Interpretation of locales in Isabelle: theories and proof contexts. In: Borwein JM, Farmer WM (eds) 5th international conference mathematical knowledge management, MKM 2006, vol 4108 of Lecture notes in computer science. Springer, pp 31–43

[Bal14] Ballarin C (2014) Locales: a module system for mathematical theories. J Autom Reason 52(2):123–153

[Bar77] Barwise J (1977) An introduction to first-order logic. In: Barwise J (ed) Handbook of mathematical logic. North-Holland, pp 5–46

[BBG$^+$18] Bancerek G, Bylinski C, Grabowski A, Kornilowicz A, Matuszewski R, Naumowicz A, Pak K (2018) The role of the mizar mathematical library for interactive proof development in Mizar. J Autom Reason 61(1–4):9–32

[BBH09] Berghofer S, Bulwahn L, Haftmann F (2009) Turning inductive into equational specifications. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) Theorem proving in higher order logics, vol 5674 of LNCS. Springer, pp 131–146

[BBN11] Blanchette JC, Bulwahn L, Nipkow T (2011) Automatic proof and disproof in Isabelle/HOL. In: Tinelli C, Sofronie-Stokkermans V (eds) Frontiers of combining systems (FroCoS 2011), vol 6989 of LNCS. Springer, pp 12–27

[BBP13] Blanchette JC, Böhme S, Paulson LC (2013) Extending Sledgehammer with SMT solvers. J Autom Reason 51(1):109–128

[BC04] Bertot Y, Castéran P (2004) Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer

[BKH$^+$08] Bulwahn L, Krauss A, Haftmann F, Erkök L, Matthews J (2008) Imperative functional programming with Isabelle/HOL. In: Mohamed OA, Muñoz CA, Tahar S (eds) 21st international conference theorem proving in higher order logics, TPHOLs 2008. vol 5170 of Lecture notes in computer science. Springer, pp 134–149

[BL18] Brunner J, Lammich P (2018) Formal verification of an executable LTL model checker with partial order reduction. J Autom Reason 60(1):3–21

[Bla12] Blanchette JC (2012) Automatic proofs and refutations for higher-order logic. PhD thesis, Technical University Munich

[Bla13] Blanchette JC (2013) Relational analysis of (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. Softw Qual J 21(1):101–126

[BM79] Boyer RS, Moore JS (1979) A computational logic. Academic Press

[BN02] Berghofer S, Nipkow T (2002) Executing higher order logic. In: Callaghan P, Luo Z, McKinna J, Pollack R (eds) Types for proofs and programs (TYPES 2000), vol 2277 of LNCS. Springer, pp 24–40

[BN04] Berghofer S, Nipkow T (2004) Random testing in Isabelle/HOL. In: Cuellar J, Liu Z (eds) Software engineering and formal methods (SEFM 2004). IEEE Computer Society, pp 230–239

[BN10] Blanchette JC, Nipkow T (2010) Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann M, Paulson LC (eds) Interactive theorem proving, vol 6172 of LNCS. Springer, pp 131–146

[Bul12a] Bulwahn L (2012) Counterexample generation for higher-order logic using functional and logic programming. PhD thesis, Technical University Munich

[Bul12b] Bulwahn L (2012) The new quickcheck for Isabelle: random, exhaustive and symbolic testing under one roof. In: Hawblitzel C, Miller D (eds) Certified programs and proofs, vol 7679 of LNCS. Springer, pp 92–108

[Bul12c] Bulwahn L (2012) Smart testing of functional programs in Isabelle. In: Bjørner N, Voronkov A (eds) Logic for programming, artificial intelligence, and reasoning, vol 7180 of LNCS. Springer, pp 153–167

[CDKM11] Chamarthi HR, Dillinger PC, Kaufmann M, Manolios P (2011) Integrating testing and interactive theorem proving. In: Hardin D, Schmaltz J (eds) 10th international workshop on the ACL2 theorem prover and its applications, ACL2 2011, vol 70 of EPTCS, pp 4–19.

[CH00] Claessen K, Hughes J (2000) QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky M, Wadler P (eds) Fifth ACM SIGPLAN international conference on functional programming (ICFP '00). ACM, pp 268–279

[CM87] Clocksin WF, Mellish CS (1987) Programming in prolog, 3rd edn. Springer

[DHT03] Dybjer P, Haiyan Q, Takeyama M (2003) Combining testing and proving in dependent type theory. In: Theorem proving in higher order logics, vol 2758 of LNCS. Springer, pp 188–203

[DJK$^+$18] Divasón J, Joosten SJC, Kuncar O, Thiemann R, Yamada A (2018) Efficient certification of complexity proofs: formalizing the Perron–Frobenius theorem (invited talk paper). In: Andronick J, Felty AP (eds) 7th ACM SIGPLAN international conference on certified programs and proofs, CPP 2018. ACM, pp 2–13

[DJTY19] Divasón J, Joosten SJC, Thiemann R, Yamada A (2019) A verified implementation of the Berlekamp–Zassenhaus factorization algorithm. J Autom Reason. Published online

[dMB08] de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan C, Rehof J (eds) Tools and algorithms for the construction and analysis of systems, vol 4963 of Lecture notes in computer science. Springer, pp 337–340

[dMKA$^+$15] de Moura LM, Kong S, Avigad J, van Doorn F, von Raumer J (2015) The Lean theorem prover (system description). In: Felty AP, Middeldorp A (eds) Automated deduction—CADE-25, volume 9195 of Lecture notes in computer science. Springer, pp 378–388.

[DW88] Debray SK, Warren DS (1988) Automatic mode inference for logic programs. J Log Program 5(3):207—229

[Ebe15] Eberl M (2015) A decision procedure for univariate real polynomials in Isabelle/HOL. In: 2015 conference on certified programs and proofs, CPP '15. ACM, pp 75–83

[ELN$^+$13] Esparza J, Lammich P, Neumann R, Nipkow T, Schimpf A, Smaus J (2013) A fully verified executable LTL model checker. In: Sharygina N, Veith H (eds) 25th international conference computer aided verification, CAV 2013, vol 8044 of Lecture notes in computer science. Springer, pp 463–478

[FGJM85] Futatsugi K, Goguen JA, Jouannaud J-P, Meseguer J (1985) Principles of OBJ2. In: 12th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL), New York, NY, USA. ACM, pp 52–66

[GAB$^+$17] Giesl J, Aschermann C, Brockschmidt M, Emmes F, Frohn F, Fuhs C, Hensel J, Otto C, Plücker M, Schneider-Kamp P, Ströder T, Swiderski S, Thiemann R (2017) Analyzing program termination and complexity automatically with AProVE. J Autom Reason 58(1):3–31

[GH98] Griffioen D, Huisman M (1998) A comparison of PVS and Isabelle/HOL. In: Grundy J, Newey M (eds) Theorem proving in higher order logics: TPHOLs '98. Springer, pp 123–142

[GKMB17] Gomes VBF, Kleppmann M, Mulligan DP, Beresford AR (2017) Verifying strong eventual consistency in distributed systems. Proc ACM Program Lang 1(OOPSLA):109:1–109:28

[GKN15] Grabowski A, Korniłowicz A, Naumowicz A (2015) Four decades of Mizar. J Autom Reason 55(3):191–198

[GM93] Gordon MJC, Melham TF (eds) (1993) Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press

[GM10] Gonthier G, Mahboubi A (2010) An introduction to small scale reflection in Coq. J Formaliz Reason 3(2)

[GMW79] Gordon MJC, Milner R, Wadsworth CP (1979) Edinburgh LCF: a mechanised logic of computation. LNCS 78. Springer

[Gog79] Goguen JA (1979) Some design principles and theory for OBJ-O, a language to express and execute algebraic specification for programs. In: Blum EK, Paul M, Takasu S (eds) Mathematical studies of information processing, vol 75 of LNCS. Springer, pp 425–473

[Gor86] Gordon MJC (1986) Why higher-order logic is a good formalism for specifying and verifying hardware. In: Milne G, Subrahmanyam PA (eds) Formal aspects of VLSI design. North-Holland, pp 153–177

[Gor00] Gordon MJC (2000) From LCF to HOL: a short history. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language, and interaction: essays in honor of Robin Milner. MIT Press, pp 169–185

[Gor15] Gordon MJC (2015) Tactics for mechanized reasoning: a commentary on Milner (1984) The use of machines to assist in rigorous proof. Philos Trans R Soc Ser A 373(2039)

[HAB$^+$17] Hales T, Adams M, Bauer G, Dang TD, Harrison J, Hoang LT, Kaliszyk C, Magron V, Mclaughlin S, Nguyen TT et al (2017) A formal proof of the Kepler conjecture. Forum Math Pi 5:e2

[Haf09] Haftmann F (2009) Code generation from specifications in higher order logic. PhD thesis, Technische Universität München

[Har96] Harrison J (1996) HOL light: a tutorial introduction. In: Srivas MK, Camilleri AJ (eds) Formal methods in computer-aided design: FMCAD '96, LNCS 1166. Springer, pp 265–269

[HHP93] Harper R, Honsell F, Plotkin G (1993) A framework for defining logics. J ACM 40(1):143–184

[HIH13] Hölzl J, Immler F, Huffman B (2013) Type classes and filters for mathematical analysis in Isabelle/HOL. In: Blazy S, Paulin-Mohring C, Pichardie D (eds) 4th international conference interactive theorem proving , LNCS 7998. Springer, pp 279–294

[HKKN13] Haftmann F, Krauss A, Kunčar O, Nipkow T (2013) Data refinement in Isabelle/HOL. In: Blazy S, Paulin-Mohring C, Pichardie D (eds) 4th international conference interactive theorem proving, vol 7998 of LNCS. Springer, pp 100–115

[HKMS17] Hunt WA, Jr, Matt K, Strother MJ, Anna S (2017) Industrial hardware and software verification with ACL2. Philos Trans R Soc Ser A 375(2104)

[HN10] Haftmann F, Nipkow T (2010) Code generation via higher-order rewrite systems. In: Blume M, Kobayashi N, Vidal G (eds) Functional and logic programming (FLOPS 2010), vol 6009 of LNCS. Springer, pp 103–117

[HN18] Hupel L, Nipkow T (2018) A verified compiler from Isabelle/HOL to CakeML. In: Ahmed A (ed) European symposium on programming (ESOP 2018), vol 10801 of LNCS. Springer, pp 999–1026

[HO82] Hoffmann Christoph M, O'Donnell Michael J (1982) Programming with equations. ACM Trans Program Lang Syst 4(1):83–112

[Hol97] Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295

[Hue75] Huet GP (1975) A unification algorithm for typed λ-calculus. Theor Comput Sci 1:27–57

[HW06] Haftmann F, Wenzel M (2006) Constructive type classes in Isabelle. In: Altenkirch T, McBride C (eds) Types for proofs and programs, vol 4502 of LNCS. Springer, pp 160–174

[HW09] Haftmann F, Wenzel M (2009) Local theory specifications in Isabelle/Isar. In: Berardi S, Damiani F, de Liguoro U (eds) Types for proofs and programs, TYPES 2008, vol 5497 of LNCS. Springer

[Jac06] Jackson D (2006) Software abstractions. Logic, language, and analysis. MIT Press

[KAE$^+$10] Klein G, Andronick J, Elphinstone K, Heiser G, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2010) sel4: formal verification of an operating-system kernel. Commun ACM 53(6):107–115

[Kal91] Kalvala S (1991) HOL around the world. In: Archer M, Joyce JJ, Levitt KN, Windley PJ (eds) International workshop on the HOL theorem proving system and its applications. IEEE Computer Society, pp 4–12

[KMNO14] Kumar R, Myreen MO, Norrish M, Owens S (2014) CakeML: a verified implementation of ML. In: Jagannathan S, Sewell P (eds) The 41st annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '14. ACM, pp 179–192

[KP18] Kunčar O, Popescu A (2018) Safety and conservativity of definitions in HOL and Isabelle/HOL. PACMPL 2(POPL):24:1–24:26

[KP19] Kunčar O, Popescu A (2019) A consistent foundation for Isabelle/HOL. J. Autom Reason 62(4):531–555

[KWP99] Kammüller F, Wenzel M, Paulson LC (1999) Locales: a sectioning concept for Isabelle. In: Bertot Y, Dowek G, Hirschowitz A, Paulin C, Thery L (eds) Theorem proving in higher order logics: TPHOLs '99, vol 1690 of LNCS. Springer

[Lam13] Lammich P (2013) Automatic data refinement. In: Blazy S, Paulin-Mohring C, Pichardie D (eds) 4th international conference interactive theorem proving ITP 2013, vol 7998 of Lecture notes in computer science. Springer, pp 84–99

[Lam14] Lammich P (2014) Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein G, Gamboa R (eds) 5th international conference interactive theorem proving ITP 2014, vol 8558 of Lecture notes in computer science. Springer, pp 325–340

[Lam16] Lammich P (2016) Refinement based verification of imperative data structures. In: Avigad J, Chlipala A (eds) 5th ACM SIGPLAN conference on certified programs and proofs. ACM, pp 27–36

[Lam17] Lammich P (2017) Efficient verified (UN)SAT certificate checking. In: de Moura L (ed) Automated deduction— CADE-26, vol 10395 of Lecture notes in computer science. Springer, pp 237–254

[Lam19] Lammich P (2019) Refinement to imperative HOL. J Autom Reason 62(4):481–503

[LB11] Lochbihler A, Bulwahn L (2011) Animating the formalised semantics of a Java-like language. In: van Eekelen Marko CJD, Geuvers H, Schmaltz J, Wiedijk F (eds) Second international conference interactive theorem proving ITP 2011, vol 6898 of Lecture notes in computer science. Springer, pp 216–232

[Ler09] Leroy X (2009) A formally verified compiler back-end. J Autom Reason 43:363–446

[LPP19] Li W, Passmore GO, Paulson LC (2019) Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. J Autom Reason 62(1):69–91

[LPY97] Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a nutshell. STTT 1(1–2):134–152

[LS19] Lammich P, Sefidgar SR (2019) Formalizing network flow algorithms: a refinement approach in isabelle/hol. J Autom Reason 62(2):261–280

[LT12] Lammich P, Tuerk T (2012) Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer L, Felty AP (eds) Third international conference interactive theorem proving ITP, 2012, vol 7406 of Lecture notes in computer science. Springer, pp 166–182

[Mil85] Milner R (1985) The use of machines to assist in rigorous proof. In: Hoare CAR, Shepherdson JC (eds) Mathematical logic and programming languages. Prentice-Hall, pp 77–88

[Mil91] Miller D (1991) A logic programming language with lambda-abstraction, function variables, and simple unification. J Log Comput 1(4):497–536

[Miz] The Mizar Mathematical Library. http://mizar.org

[ML84] Martin-Löf P (1984) Constructive mathematics and computer programming. Philos Trans R Soc Ser A 312(1522):501–518

[MMW16] Matichuk D, Murray TC, Wenzel M (2016) Eisbach: a proof method language for Isabelle. J Autom Reason 56(3)

[MQP06] Meng J, Quigley C, Paulson LC (2006) Automation for interactive proof: first prototype. Inf Comput 204(10):1575–1596

[MW10] Matthews D, Wenzel M (2010) Efficient parallel programming in Poly/ML and Isabelle/ML. In: ACM SIGPLAN workshop on declarative aspects of multicore programming (DAMP 2010)

[Nip91a] Nipkow T (1991) Higher-order critical pairs. In: Proceedings 6th IEEE symposium logic in computer science. IEEE Press, pp 342–349

[Nip91b] Nipkow T (1991) Higher-order unification, polymorphism, and subsorts. In: Kaplan S, Okada M (eds) Proceedings 2nd international workshop conditional and typed rewriting systems, vol 516 of LNCS. Springer

[Nip93a] Nipkow T (1993) Functional unification of higher-order patterns. In: Proceedings 8th IEEE symposium logic in computer science, pp 64–74

[Nip93b] Nipkow T (1993) Order-sorted polymorphism in Isabelle. In: Huet G, Plotkin G (eds) Logical environments. Cambridge Uiversity Press, pp 164–188

[NK14] Nipkow T, Klein G (2014) Concrete semantics with Isabelle/HOL. Springer, 298 pp. http://concrete-semantics.org.

[NM16] Nagele J, Middeldorp A (2016) Certification of classical confluence results for left-linear term rewrite systems. In: Blanchette JC, Merz S (eds) 7th international conference interactive theorem proving ITP, 2016, vol 9807 of Lecture notes in computer science. Springer, pp 290–306

[NO80] Nelson G, Oppen DC (1980) Fast decision procedures based on congruence closure. J ACM 27(2):356–364

[NP92] Nipkow T, Paulson LC (1992) Isabelle-91. In: Kapur D (ed) Automated deduction—CADE-11, vol 607 of LNCS. Springer, pp 673–676

[NP93] Nipkow T, Prehofer C (1993) Type checking type classes. In: Principles of programming languages, POPL '93, New York, NY, USA. ACM, pp 409–418

[NP98] Nipkow T, Prehofer C (1998) Higher-order rewriting and equational reasoning. In: Bibel W, Schmitt P (eds) Automated deduction—a basis for applications. Volume I: foundations, vol 8 of Applied logic series. Kluwer, pp 399–430

[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic. Springer, Online at http://isabelle.in.tum.de/dist/Isabelle/doc/tutorial.pdf.

[NS91] Nipkow T, Snelting G (1991) Type classes and overloading resolution via order-sorted unification. In: Hughes J (ed) Proceedings 5th ACM conference functional programming languages and computer architecture, vol 523 of LNCS. Springer, pp 1–14

[Obu06] Obua S (2006) Checking conservativity of overloaded definitions in higher-order logic. In: Pfenning F (ed) Term rewriting and applications, vol 4098 of LNCS. Springer, pp 212–226

[O'D77] O'Donnell MJ (1977) Computing in systems described by equations, vol 58 of LNCS. Springer

[Ove75] Overbeek R (1975) An implementation of hyper-resolution. Comput Math Appl 1:201–214

[Owr06] Owre S (2006) Random testing in PVS. In: Workshop on automated formal methods (AFM). http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf

[Pau86] Paulson LC (1986) Natural deduction as higher-order resolution. J Log Program 3:237–258

[Pau89] Paulson LC (1989) The foundation of a generic theorem prover. J Autom Reson 5(3):363–397

[Pau90] Paulson LC (1990) Isabelle: the next 700 theorem provers. In: Odifreddi P (ed) Logic and computer science. Academic Press, pp 361–386

[Pau93] Paulson LC (1993) Isabelle's object-logics. Technical report 286, Cambridge University Computer Laboratory

[Pau94] Paulson LC (1994) Isabelle—a generic theorem prover (with contributions by T. Nipkow), vol 828 of Lecture notes in computer science. Springer

[Pau98] Paulson LC (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6(1–2):85–128

[Pau99] Paulson LC (1999) A generic tableau prover and its integration with Isabelle. J Univers Comput Sci 5(3):73–87

[Pau03] Paulson LC (2003) The relative consistency of the axiom of choice—mechanized using Isabelle/ZF. LMS J Comput Math 6:198–248. http://www.lms.ac.uk/jcm/6/lms2003-001/

[Pau04] Paulson LC (2004) Organizing numerical theories using axiomatic type classes. J Autom Reason 33(1):29–49

[Pau18] Paulson LC (2018) Computational logic: its origins and applications. Proc R Soc Lond A Math Phys Eng Sci 474(2210)

[Pel96] Peled DA (1996) Combining partial order reductions with on-the-fly model-checking. Form Methods Syst Des 8(1):39–64

[PG96] Paulson LC, Grabczewski K (1996) Mechanizing set theory: cardinal arithmetic and the axiom of choice. J Autom Reason 17(3):291–323

[PHD$^+$15] Paraskevopoulou Z, Hritcu C, Dénès M, Lampropoulos L, Pierce BC (2015) Foundational property-based testing. In: Urban C, Zhang X (eds) Interactive theorem proving, vol 9236 of LNCS. Springer, pp 325–343

[PS07] Paulson LC, Susanto KW (2007) Source-level proof reconstruction for interactive theorem proving. In: Schneider K, Brandt J (eds) Theorem proving in higher order logics: TPHOLs 2007, LNCS 4732. Springer, pp 232–245

[Raj93] Rajan SP (1993) Executing HOL specifications: towards an evaluation semantics for classical higher order logic. In: Claesen Luc JM, Gordon Michael JC (eds) Higher order logic theorem proving and its applications, vol A-20 of IFIP transactions. North-Holland/Elsevier, pp 527–536

[RNL08] Runciman C, Naylor M, Lindblad F (2008) SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In: Gill A (ed) Proceediong of the 1st ACM SIGPLAN symposium on Haskell. ACM, pp 37–48

[Rob65] Robinson JA (1965) A machine-oriented logic based on the resolution principle. J ACM 12:23–41

[RV02] Riazanov A, Voronkov A (2002) The design and implementation of VAMPIRE. AI Commun 15(2):91–110

[Sch04] Schulz S (2004) System description: E 0.81. In: Basin D, Rusinowitch M (eds) Automated reasoning—second international joint conference, IJCAR 2004, LNAI 3097. Springer, pp 223–228

[Sha02] Shankar N (2002) Little engines of proof. In: Eriksson L-H, Lindsay P (eds) FME 2002: formal methods—getting IT right: international symposium of formal methods Europe, LNCS 2391. Springer, pp 1–20

[Sie19] Siegel SF (2019) What's wrong with on-the-fly partial order reduction. In: Dillig I, Tasiran S (ed) Computer aided verification (CAV 2019), LNCS. Springer

[SK07] Spiridonov A, Khurshid S (2007) Automatic generation of counterexamples for ACL2 using alloy. In: Seventh international workshop on the ACL2 theorem prover and its applications

[SN08] Slind K, Norrish M (2008) A brief overview of HOL4. In: Mohamed O, Muñoz C, Tahar S (eds) Theorem proving in higher order logics, TPHOLs 2008, pp 28–32

[Sum02] Sumners R (2002) Checking ACL2 theorems via SAT checking. In: Third international workshop on the ACL2 theorem prover and its applications

[TJ07] Torlak E, Jackson D (2007) Kodkod: a relational model finder. In: Grumberg O, Huth M (eds) Tools and algorithms for the construction and analysis of systems, vol 4424 of LNCS. Springer, pp 632–647

[TS09] Thiemann R, Sternagel C (2009) Certification of termination proofs using CeTA. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) 22nd international conference theorem proving in higher order logics, TPHOLs 2009, vol 5674 of Lecture notes in computer science. Springer, pp 452–468

[Web05] Weber T (2005) Bounded model generation for Isabelle/HOL. In: Ahrendt W, Baumgartner P, de Nivelle H, Ranise S, Tinelli C (eds) Selected papers from the workshops on disproving and the second international workshop on pragmatics of decision procedures (PDPAR 2004), vol 125(3) of Electronic notes in theoretical computer science. Elsevier, pp 103–116

[Web08] Weber T (2008) SAT-based finite model generation for higher-order logic. PhD thesis, Technical University Munich, Germany

[Wei01] Weidenbach C (2001) Combining superposition, sorts and splitting. In: Robinson A, Voronkov A (eds) Handbook of automated reasoning, vol II, chapter 27, pp 1965–2013. Elsevier Science

[Wen97] Wenzel Ma (1997) Type classes and overloading in higher-order logic. In: Theorem proving in higher order logics, vol 1275 of LNCS. Springer, pp 307–322

[Wen07] Wenzel M (2007) Isabelle/Isar—a generic framework for human-readable proof documents. Stud Log Gramm Rhetor 10(23):277–297 From Insight to Proof—Festschrift in Honour of Andrzej Trybulec

[Wen11] Wenzel M (2011) Isabelle as document-oriented proof assistant. In: Davenport JH et al (eds) Conference on intelligent computer mathematics (CICM 2011), vol 6824 of LNAI. Springer

[Wen13a] Wenzel M (2013) READ-EVAL-PRINT in parallel and asynchronous proof-checking. In: Kaliszyk C, Lüth C (eds) User interfaces for theorem provers (UITP 2012), vol 118 of Electronic proceedings in theoretical computer science

[Wen13b] Wenzel M (2013) Shared-memory multiprocessing for interactive theorem proving. In Blazy S, Paulin-Mohring C, Pichardie D (eds) Interactive theorem proving (ITP 2013), vol 7998 of Lecture notes in computer science. Springer

[Wen14] Wenzel M (2014) Asynchronous user interaction and tool integration in Isabelle/PIDE. In: Klein G, Gamboa R (eds) Interactive theorem proving (ITP 2014), vol 8558 of LNCS. Springer

[Wen19] Wenzel M (2019) Interaction with formal mathematical documents in Isabelle/PIDE. In: Kaliszyk C, Brady E, Kohlhase A, Sacerdoti CC (eds) Intelligent computer mathematics (CICM 2019), vol 11617 of LNAI. Springer. https://arxiv.org/abs/1905.01735

[Wie01] Wiedijk F (2001) Mizar light for HOL light. In: Boulton RJ, Jackson PB (eds) Theorem proving in higher order logics, TPHOLs 2001, Berlin, Heidelberg. Springer, Berlin, pp 378–393

[WL18] Wimmer S, Lammich P (2018) Verified model checking of timed automata. In: Beyer D, Huisman M (eds) Tools and algorithms for the construction and analysis of systems, TACAS 2018, vol 10805 of Lecture notes in computer science. Springer, pp 61–78

[Woo18] Wood C (2018) The strange numbers that birthed modern algebra. https://www.quantamagazine.org/the-strange-numbers-that-birthed-modern-algebra-20180906/

Received 25 March 2019
Accepted in revised form 9 August 2019 by Cliff Jones and José N. Oliveira
Published online 2 September 2019