Interactive Theorem Provers from the perspective of Isabelle/Isar

Makarius Wenzel

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France makarius@sketis.net

Abstract. Interactive Theorem Provers have a long tradition, going back to the 1970s when interaction was introduced as a concept in computing. The main provers in use today can be traced back over 20–30 years of development. As common traits there are usually strong logical systems at the bottom, with many layers of add-on tools around the logical core, and big applications of formalized mathematics or formal methods. There is a general attitude towards flexibility and open-endedness in the combination of logical tools: typical interactive provers use automated provers and dis-provers routinely in their portfolio.
The subsequent exposition of interactive theorem proving (ITP) takes Isabelle/Isar as the focal point to explain concepts of the greater "LCF family", which includes Coq and various HOL systems. Isabelle itself shares much of the relatively simple logical foundations of HOL, but follows Coq in the ambition to deliver a sophisticated system to end-users without requiring self-assembly of individual parts. Isabelle today is probably the most advanced proof assistant concerning its overall architecture and extra-logical infrastructure.
The "Isar" aspect of Isabelle refers first to the structured language for human-readable and machine-checkable proof documents, but also to the Isabelle architecture that emerged around the logical framework in the past 10 years. Thus "Isabelle/Isar" today refers to an advanced proof assistant with extra structural integrity beyond the core logical framework, with native support for parallel proof processing and asynchronous interaction in its Prover IDE (PIDE).

1 Introduction

▷ *Briefly introduce your kind of proof production tool (e.g. Sat-solvers, SMT-solvers, First-order ATPs, Higher-order ATPs, Proof Assistants, ...). Try to give an overview encompassing all your community. If possible, do not to restrict yourself only to the tool(s) you develop.*

Isabelle is one of a handful of *Interactive Theorem Provers* or *Proof Assistants* that have reached a sufficient level of sophistication over some decades to support substantial applications. Today we see large formalization efforts like L4.verified [23], and the *Archive of Formal Proofs* [1] with entries for Isabelle/HOL.

Other notable systems in this category are Coq [47, §4], the HOL family [47, §1], PVS [47, §3], ACL2 [47, §8]. The volume [47] is generally useful as a reference to various interactive (and non-interactive) provers, although it reflects the state-of-the-art from 10 years ago.

Due to lack of expertise on PVS and ACL2, I need to restrict the perspective of ITP to the main European systems in the LCF tradition, namely the HOL family, Coq, and Isabelle. All of them share much heritage from LCF, such as strongly-typed foundations and programming in ML. In contrast, the US American prover tradition is strongly influenced by LISP, concerning syntax and type-system of the formal languages.

Coq is probably the most popular and publicly visible ITP system at the moment. Its implementation and bootstrap language is OCaml, but user input uses a separate "vernacular" (Gallina) for specifications and tactic scripting language (Ltac) for proofs. Since Coq is based on constructive logic with a built-in notion of computation, it is customary to build tools for Coq within the logical language of Coq itself, which are then executed by its byte-code interpreter for efficient

[1] http://afp.sf.net

normalization of proof terms. This trend to "internalize" all aspects of formal reasoning into one big language for logic and programming is specific to Coq.

▷ *What are the tools that pioneered proof production in your area? What are the tools that currently produce proofs? How widespread is the feature of proof production among tools in your area? How was the historical evolution of the proof production feature in your area?*

The HOL family, Coq, and Isabelle are all descendants of LCF, which was developed in Edinburgh and Cambridge in the 1980s by R. Milner. The LCF line was continued by M. Gordon (later HOL), L. C. Paulson (later Isabelle), and G. Huet (later Coq). Paulson and Huet were actually collaborating on the last version of Cambridge LCF, before the split into Isabelle and Coq happened. The core algorithm in Isabelle86 for higher-order unification is actually due to Huet.

Milner can be credited as the inventor of the key ideas for interactive theorem proving and the ML programming language, with the now famous Hindley-Milner type-inference [9] to make a statically-typed language almost as convenient as an untyped one. The so-called "LCF-approach" to interactive theorem proving means the following:

1. **Strong logical foundations.** Some well-understood logical basis is taken as starting point, and mathematical theories are explicitly constructed by reduction to first principles. This follows the tradition of "honest toil" in the sense of B. Russel: results are not just postulated as axioms, but derived from definitions as proper theorems.
2. **Free programmability and extensibility.** Derived proof tools can be implemented on top of the logical core, while retaining its integrity. This works thanks to the strong type-safety properties of the ML implementation platform of the prover.

Originally, Isabelle was introduced as yet another *Logical Framework* [31, 32] when that idea was popular: this was motivated by frequently changing versions of Martin-Löf Type-Theory that Paulson wanted to support on the machine. Later that flexibility of the Isabelle/Pure framework was used for Isabelle/ZF, which is a version of classical Zermelo-Fraenkel set-theory on top of first-order logic [33, 34].

In the past 15 years, Isabelle/HOL [30] has become the largest application of the Isabelle framework, with numerous add-on tools: advanced proof search, support for specific theory reasoning, integration with external ATPs and SMTs, counter-example generation etc. Moreover, any advanced specification mechanism like **datatype**, **inductive fun** in Isabelle/HOL are implemented in the hard way, with proper definitions and proofs produced in ML. For example, some recursive function specification given by the user is turned by the system into suitable low-level definitions, and the expected characterization is derived as theorems under program control.

Conceptually, we have an alternation of theory definitions and ML modules that gradually build up a library of formalized mathematics. There is no theoretical limit to that, but in practice the library is eventually delivered to end-users as theory *Main* of Isabelle/HOL, and further libraries and applications require very little Isabelle/ML programming.

The subsequent examples give some taste of Isabelle/HOL for end-users. The theory snippet below defines an algebraic datatype of finite sequences, with a function for sequence concatenation, and proves some elementary facts about that.

```
datatype 'a seq = Empty | Seq 'a ('a seq)

fun concat :: 'a seq ⇒ 'a seq ⇒ 'a seq
where
  concat Empty ys = ys
| concat (Seq x xs) ys = Seq x (concat xs ys)

theorem concat-empty: concat xs Empty = xs
  by (induct xs) simp-all

theorem conc-assoc: concat (concat xs ys) zs = concat xs (concat ys zs)
```

by (*induct xs*) *simp-all*

On first sight that might look like a functional programming language in the style of ML or Haskell, with extensions for logic and proof. This is in fact the manner how Isabelle/HOL is introduced in the tutorial [29]. As a second approximation, it could be seen as theory axiomatization, as in algebraic specification or preludes of first-order automated theorem proving. In reality, the above is merely some surface syntax for a *definitional theory* in a double-sense: the language elements **datatype**, **fun**, even **theorem** and the proof tools *induct* and *simp-all* are defined somewhere in the library, or the bootstrap environment of Isabelle/Pure and Isabelle/HOL.

This strongly definitional approach is shared by all members of the LCF family, with the exception of LCF itself, which was still based on axiomatic specifications of types and constants, as a prelude to proper proofs. The introduction of Higher-Order Logic as new foundation for HOL [15, 14] made it feasible to apply the rigorous principles of "honest toil" (B. Russel) to mechanized reasoning on the machine. Note that Coq has its own variations on the same theme: its Calculus of Inductive Constructions starts as a relatively strong axiomatic basis for user theories that are usually definitional, except for situations where well-known axioms are added in applications that want to escape the constructive defaults of Coq.

In the example above, proofs merely consist of terminal steps of the form **by** *initial-method terminal-method*, where the second argument is optional. That is the shortest possible structured Isar proof: a double-step to split the problem initially (e.g. by induction) and solve the remaining situation (e.g. by simplification). The **by** command ensures the structural integrity of this reasoning scheme: there is no way to escape from the nested proof structure, which is explicit in Isabelle/Isar. Coq has recently introduced "bullets" inspired by the Coq/SSReflect [13] extensions to impose more structure on tactic scripts, while retaining unstructured proof states.

Isabelle/Isar is sufficiently flexible to imitate unstructured proof scripts within its structured proof language, by so-called "improper language elements" like **apply** and **done**:

theorem *concat-empty′*: *concat xs Empty* = *xs*
  apply (*induct xs*)
   apply *simp*
  apply *simp*
  done

theorem *conc-assoc′*: *concat* (*concat xs ys*) *zs* = *concat xs* (*concat ys zs*)
  apply (*induct xs*)
   apply *simp*
  apply *simp*
  done

Such liberality in the Isar proof language, outside its main scope for human-readable structured proofs, allows to import old Isabelle tactic scripts easily or to port applications from other tactical proof assistants. The price for that is some occasional confusion of users, who might mistake Isar proof methods like *simp* as "tactics", and the Isar proof text as some kind of "proof script". The primary input of Isabelle/Isar is better understood as a format for "proof documents", and in fact there is built-in support to render the result nicely in LaTeX, see also [39, chapter 4]. The present article is an example for that: it is a formal theory document in the context of theory *Main* from Isabelle/HOL, with a lot of unproven prose text and a few formally proven examples.

The subsequent example shows more structure, both in the specification context (via **class**) and the proofs (via calculational reasoning elements **also** and **finally**):

class *group* = *times* + *one* + *inverse* +
  assumes *group-assoc*: (*x* ∗ *y*) ∗ *z* = *x* ∗ (*y* ∗ *z*)
    and *group-left-one*: 1 ∗ *x* = *x*
    and *group-left-inverse*: *inverse x* ∗ *x* = 1

theorem (**in** *group*) *group-right-inverse*: *x* ∗ *inverse x* = 1

```
proof −
  have x ∗ inverse x = 1 ∗ (x ∗ inverse x)
    by (simp only: group-left-one)
  also have ... = 1 ∗ x ∗ inverse x
    by (simp only: group-assoc)
  also have ... = inverse (inverse x) ∗ inverse x ∗ x ∗ inverse x
    by (simp only: group-left-inverse)
  also have ... = inverse (inverse x) ∗ (inverse x ∗ x) ∗ inverse x
    by (simp only: group-assoc)
  also have ... = inverse (inverse x) ∗ 1 ∗ inverse x
    by (simp only: group-left-inverse)
  also have ... = inverse (inverse x) ∗ (1 ∗ inverse x)
    by (simp only: group-assoc)
  also have ... = inverse (inverse x) ∗ inverse x
    by (simp only: group-left-one)
  also have ... = 1
    by (simp only: group-left-inverse)
  finally show ?thesis .
qed

theorem (in group) group-right-one: x ∗ 1 = x
proof −
  have x ∗ 1 = x ∗ (inverse x ∗ x)
    by (simp only: group-left-inverse)
  also have ... = x ∗ inverse x ∗ x
    by (simp only: group-assoc)
  also have ... = 1 ∗ x
    by (simp only: group-right-inverse)
  also have ... = x
    by (simp only: group-left-one)
  finally show ?thesis .
qed
```

The emphasis on structured proof texts — beyond mere "scripts" — goes back to the beginnings of Isar in 1999 [40] when *human-readable proof documents* were the main motivation. Rich structure helps to present tiny examples and big applications nicely: the user is taken seriously as proof author, although there is an extra effort to produce readable formal proofs in the first place. That investment is usually returned when big proof libraries need to be maintained, or extended in the scope of application: adapting well-structured proofs to new situations usually works nicely.

Structured proofs also help the machine to process large formalizations efficiently. This is in contrast to the occasional misunderstanding of "high-level proofs" as "automated proofs". An important lesson learned from Isar is that meaningful human-readable structured proofs require only modest proof automation (existing higher-order unification of Isabelle/Pure), while arbitrary proof tools of different reasoning strength can be adjoined as terminal proof steps. A related principle can be seen in Mizar [47, §2]: its mathematical proof language rests on a relatively weak notion of "obvious inferences" performed by the machine, and it even lacks the possibilities to appeal to add-on tools within the system.

Another performance advantage of structured proofs is due to parallel proof processing on multi-core hardware, which is now ubiquitous. Initially, the strict modularity of Isar proofs was merely motivated by aesthetic principles. The advent of multicore hardware in the mass market introduced the demand to make high-performance applications work in parallel. This required some reforms of existing Isabelle/Isar document processing, to gain significant speedup of batch-processing of large formalizations [41, 28, 44]. For example, re-checking the *Archive of Formal Proofs* used to be an overnight job, but is today finished in $\approx$ 1h on 8 cores with hyperthreading, even though the archive has grown substantially in recent years.

Apart from parallel batch processing, interaction had to be reconsidered as well, to fit into the non-sequential paradigm. This is addressed in Isabelle/PIDE, by a timeless and stateless *document model* that integrates all aspects of proof editing and add-on tools within a uniform framework [42, 43, 45]. These advanced concepts of interactive theorem proving are specific to Isabelle: the HOL family still uses plain TTY sessions with copy-paste, while Coq adheres to the traditional model of Proof General [1], either directly in that Emacs mode or indirectly in CoqIde.

▷ *Is there an open-source and minimalistic* proof-producing *version of your kind of tool that would be particularly suitable for beginners to look at and modify?*

HOL-Light[2] by J. Harrison is best-known for its a very small code base, which greatly helps to give some impression how a classic LCF-style prover is implemented, with minimal system infrastructure. It is possible to browse through HOL-Light sources within a few hours, and learn many things from it. One needs to keep in mind, though, that HOL-Light lacks fundamental principles of modern ITP systems, and the implementation of the LCF kernel is not bullet-proof due to weaknesses of its implementation language OCaml.

Isabelle is at the opposite end of the spectrum of technical sophistication and complexity. Its multi-layered kernel architecture is meant to provide certain guarantees to users, at the cost of considerable code complexity. The Isabelle/ML sources of the core context management have about the same size as the HOL-Light inference kernel, but that "nano kernel" of Isabelle introduces extra structural integrity that is absent in HOL-Light. The explicit context management of Isabelle is essential to allow unhindered parallel execution on multicore hardware, for example.

The CakeML[3] project is notable, since one of its many aspects is to reconstruct a fully verified HOL-Light kernel [24] in a safe variant of Standard ML, instead of unsafe OCaml. If a complex proof assistant like Isabelle would export all its reasoning to such a fully verified minimal system, we could gain levels of confidence beyond the classic LCF tradition so far.

2 Proof Systems

▷ *Please list the (most common) proof systems (e.g. resolution, superposition, tableaux, sequent calculus, natural deduction, ...) underlying state-of-the-art tools of your kind.*

ITP systems usually don't commit to particular forms of mechanized reasoning, but the LCF family favours certain forms of λ-calculus and Natural Deduction. Since the systems are freely programmable in ML, arbitrary proof tools with different logical calculi may be implemented as well. Users and tool developers have done that routinely in the past decades, so all of the above-mentioned proof systems may be seen in the prover libraries. A classic example in Isabelle is the generic tableau prover *blast* [36], which is popular until today.

▷ *Please show the inference rules of (some of) these proof systems explicitly.*

The subsequent inference systems are for Isabelle/Pure, which is the most elementary in the LCF family. It is based on *Minimal Higher-Order Logic*, i.e. a reduced version of classic HOL to support arbitrarily nested natural deduction proofs, without the axiomatic basis that is required to bootstrap a library of formalized mathematics. Thus Isabelle/Pure remains pure from logical presuppositions, and can be used for Isabelle/ZF as well. Isabelle/HOL adds its own bootstrap axioms, and commits to a particular classic interpretation of the logical framework.

Isabelle/Pure consists of three levels of λ-calculus: one for syntax and two for logical reasoning with the connectives $\bigwedge$ and $\Longrightarrow$. Figure 1 illustrates the syntactic formation rules of the language of *types* and *terms*. Types have a simple first-order structure of type variables and type constructors, e.g. *bool*, *nat*, *'a list*. Terms are simply-typed λ-terms, with atoms (variables, constants), application, and abstraction.

[2] `https://code.google.com/p/hol-light`
[3] `https://cakeml.org`

type *fun* :: (*type*, *type*)*type*	function space $\alpha \Rightarrow \beta$
const *all* :: $('a \Rightarrow prop) \Rightarrow prop$	universal quantification $\bigwedge x{::}\alpha.\ B\ x$
const *imp* :: $prop \Rightarrow prop \Rightarrow prop$	implication $A \Longrightarrow B$

$$\frac{}{a_\tau :: \tau} \qquad \frac{t :: \sigma}{(\lambda x_\tau.\ t) :: \tau \Rightarrow \sigma} \qquad \frac{t :: \tau \Rightarrow \sigma \quad u :: \tau}{t\ u :: \sigma}$$

Fig. 1. Isabelle/Pure: abstract syntax with simple types

Figure 2 illustrates the formal treatment of *context* in Isabelle/Pure. The background theory Θ helps to organize big libraries of formalized mathematics in a graph-structured manner, with merge and extend operations. The foundational order of logical specification is preserved: a theory graph may always be flattened into a linear order of primitives. In contrast to programming languages like Java or Haskell, there is *no* mutual recursion of modules.

Results proven in one theory may be transferred to a bigger theory, according to the sub-theory relation that is formally maintained by the system. LCF and the HOL family leave this context implicit in the ML runtime environment, which means there is only a single monotonically growing theory state, without the possibility to go back (undo) nor the possibility to let the system perform inferences within different theories at the same time. Coq does not have a theory context Θ, because its primary context Γ is sufficiently expressive to take over this role.

In Isabelle/Pure, the context Γ serves as local proof context, with slightly different characteristics than the background theory Θ. It allows to build a local situation for specifications and proofs, with operations to *export* (abstraction) and *interpret* (application) results. The system provides a notion of *morphism* for that, to transform arbitrary items along the structure of contexts, but morphisms are not part of the logic.

background theory Θ:	polymorphic types, constants, axioms (definitions)
proof context Γ:	fixed variables, assumptions
merge and extend:	$\Theta_3 = \Theta_1 \cup \Theta_2 + \tau + c :: \tau + c \equiv t$
sub-theory relation:	$\Theta_1 \subseteq \Theta_2$
transfer of results:	if $\Theta_1 \subseteq \Theta_2$ and $\Theta_1,\ \Gamma \vdash \varphi$ then $\Theta_2,\ \Gamma \vdash \varphi$

Fig. 2. Isabelle/Pure: formal context

Figure 3 shows the rules for primitive inferences of Isabelle/Pure, within the formal theory and proof context. Apart from bookkeeping wrt. the context (via *axiom* and *assume*), the remaining rules are standard introductions and eliminations for the logical connectives $\bigwedge$ and $\Longrightarrow$. This is minimal Higher-Order Logic presented as Natural Deduction inference system.

$$\frac{A \in \Theta}{\vdash A}\ (\mathit{axiom}) \qquad \frac{}{A \vdash A}\ (\mathit{assume})$$

$$\frac{\Gamma \vdash B[x] \quad x \notin \Gamma}{\Gamma \vdash \bigwedge x.\ B[x]}\ (\bigwedge\text{-}\mathit{intro}) \qquad \frac{\Gamma \vdash \bigwedge x.\ B[x]}{\Gamma \vdash B[a]}\ (\bigwedge\text{-}\mathit{elim})$$

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B}\ (\Longrightarrow\text{-}\mathit{intro}) \qquad \frac{\Gamma_1 \vdash A \Longrightarrow B \quad \Gamma_2 \vdash A}{\Gamma_1 \cup \Gamma_2 \vdash B}\ (\Longrightarrow\text{-}\mathit{elim})$$

Fig. 3. Isabelle/Pure: primitive inferences

Isabelle/Pure also provides a notion of equality, which is axiomatized as $\alpha\beta\eta$-congruence of λ-calculus, and later used as foundation for definitional tools and equational reasoning.

Treatment of equality is particularly important for Type-Theory provers like Coq. Instead of naive "mathematical equality" seen in the HOL-based systems, constructive type-theory cares about more detailed distinction of computable vs. non-computable logical objects, observable vs. non-observable equality etc. In addition to regular $\alpha\beta$-congruence (without η), Coq has built-in notions of $\delta\iota$-conversion to expand definitions (recursive functions over inductive types) inside the formal system. Isabelle and the HOL family perform the latter by more conventional equational reasoning, using tools for rewriting and normalization implemented outside the logical kernel.

Reconsidering the main inference systems of Isabelle/Pure (figure 1, figure 2, figure 3), it is important to understand that little of that is directly exposed to end-users, for a variety of reasons. The general principle is that logical foundations are merely foundational, and additional system infrastructure outside the logic takes over responsibilities to provide convenient access to high-level concepts.

For example, formation of well-typed terms (figure 1) works via Hindley-Milner type-inference [9], even though the logic has merely schematic type variables at the outer theory level, and lacks proper polymorphism. Thus terms can be written with very little type information, and the system reconstructs the rest. Isabelle/HOL even provides an add-on module for the generic type-infrastructure of Isabelle, to work with *coercive sub-typing* [38]: suitable conversion functions are inserted to adjust to different types, e.g. to turn some non-fitting argument of type *nat* into *int*.

Even the main inference system of the Isabelle/Pure kernel (figure 3) is hardly encountered by users in that form. It mainly serves as foundation to more abstract concepts, such as block-structured reasoning in nested contexts. The vacuous example below illustrates this in Isar notation:

notepad
begin — Fresh proof context Γ starts here.

{
fix $a\ b\ c :: {}'a$ — Lets fix some arbitrary hypothetical items ...
assume $a = b$ **and** $b = c$ — ... and assume some propositions.

In the current context we ...

have $a = c$ **using** ‹$a = b$› **and** ‹$b = c$› **by** (*rule trans*)
}

After leaving that context, the system performs some $\Longrightarrow$-*intro* and $\bigwedge$−*intro* reasoning for us, so that the context elements above are no longer fixed, but arbitrary. This means, we ...

have $\bigwedge a\ b\ c.\ a = b \Longrightarrow b = c \Longrightarrow a = c$ **by** *fact*

end

That structured reasoning in Isar is not another logical inference system, but the operation of an interpreter for a proof language that reduces proof text to primitive steps of the logical kernel.

▷ *Why are these proof systems particularly useful for your automated deduction tools? Which features make them suitable? What is not so convenient in them?*

One needs to understand that the implementation, maintenance, and application of major interactive theorem provers is an ongoing effort of several decades. That means the original logical foundations are somewhat accidental, according to certain intentions of the original authors and "latest trends" from a long time ago. Such starting conditions are hard to change later, when a system has reached a critical mass of tools and applications that depend on exactly these foundations. Developers of ITP systems and tools are used to cope with that routinely, and usually manage to turn seemingly old-fashioned structures into exciting new applications.

So in practice there is a combination of flexibility and stability: Whatever happens to be implemented as initial foundations is sufficiently open to be adapted gradually over time to changing demands and new requirements — otherwise the prover in question would have died out already. Significant changes of the logical foundations are unlikely, just due to the weight and gravity of and implemented system with its existing applications.

For example, the notion of local context Γ in Isabelle/Pure started out as a modest concept of minimal Natural Deduction in 1989, but acquired further roles in structured Isar proofs and structured specifications in 1999, until it became the main infrastructure that underlie the *local theory* specifications of Isabelle today (since 2007). Local theories [17] provide generic infrastructure to implement module concepts in Isabelle, outside of the core logic. This unifies locales [22], locale interpretation [2], type classes and class instantiation [16], adhoc overloading etc. and allows to combine them with derived definitional principles like **definition**, **inductive**, **datatype**, **fun**, **function**, **primrec**, **primcorec** etc.

▷ *What are the trends w.r.t. proof systems for your kind of tool?*

I see two contradictory trends happening at the same time.

1. **Convergence of different systems.** The demands from realistic applications and large-scale formalization efforts encourages quite different proof assistants to overcome their foundational biases and arrive at similar user-space tools and libraries. For example, Coq has built-in notions for inductive types and recursive functions over them, but they were not sufficient for general recursion so that was eventually re-implemented as add-on without changing the already complex kernel. Isabelle/HOL had nothing like that from the outset, so everything had to be implemented as add-on in user-space, leading to some complex tool suite in the library that achieves almost the same as Coq today.
2. **Divergence within the same system.** Despite the huge efforts to push significant changes of the logical foundations through the real implementations, and the libraries in particular, there are occasionally ambitions to invent new ways to do formal logic on the computer. Coq is presently faced with the movement towards *Homotopy Type Theory*[4], which overthrows certain starting conditions of the original Calculus of Inductive Constructions as foundation for Coq. HOL is generally less interesting from theoretical viewpoint, and thus more robust against experiments with alternative axiomatizations, but it sometimes happens on private side-branches of HOL [20].

3 Proof Search

▷ *Which algorithms are used to search for a proof/refutation? Is the procedure able to find (counter-)models as well?*

LCF-style proof assistants have traditionally ignored the question of proof search, and merely provided the logical core with free programmability (in ML) to let the user implement arbitrary proof search tools. After some decades, numerous tools have accumulated. The standard distribution of Isabelle/HOL today provides plenty of possibilities to apply proof search in practice, without asking the user to do ML programming again.

The classic portfolio of Isabelle proof tools may be categorized as follows:

- Single rule application, with application-specific declarations and concrete syntax, notably:
  - method *rule* for generic Natural Deduction (with higher-order unification);
  - method *cases* for elimination, syntactic representation of datatypes, inversion of inductive sets and predicates;
  - methods *induct* and *coinduct* for induction and coinduction of types, sets, and predicates.

[4] `http://homotopytypetheory.org`

– Equational reasoning by the Simplifier, with possibilities for add-on tools: simplification procedures, loopers, solvers. The main entry points are *simp* and *simp-all*, see also [39, §9.3].
– Classical reasoning with simple proof search procedures, based on classification of rules in the library: *intro*, *elim*, *dest*. Due to this instrumentation, relatively simple proof tools work quite effectively, even with large theory libraries. There are two standard implementations available.
  1. The Classical Reasoners with variations on search strategy via *fast*, *safe*, *clarify*, *best*, see also [35] and [39, §9.4].
  2. The Classic Tableau Prover *blast* [36].
– Various combinations of the Simplifier and Classical Reasoner tools, notably *auto*, *force*, *fastforce*.
– Metis[5] as medium-strength ATP inside Isabelle/HOL, with full proof reconstruction like the other standard tools. This is particularly important to Sledgehammer [5], which uses arbitrary external ATPs (and SMTs) for proof search, but needs to produce a proper LCF-style proof in the end. Note that *metis* is *not* accepting any instrumentation of the library: it works on a (limited) collection of facts that are given on the spot, usually produced by Sledgehammer for inlining into the proof text.
– Special tools for special situations, with some (semi)decision procedures for well-known theories like Presburger Arithmetic (methods *arith* or *presburger*).
– Integration of Z3[6] as general-purpose SMT, via the proof method *smt*, which performs explicit proof reconstruction for a large subset of Z3 proofs [7], using standard Isabelle tools like *simp*, *blast*, *auto* internally.

To get an overview of the zoo of Isabelle proof methods, one needs to keep the general tool frameworks behind them in mind, and avoid confusion of tools of similar names in other provers. For example, `simpl` in Coq refers to the builtin notion of computation of inductive definitions and recursive functions over them, which is a relatively powerful principle, but not as flexible as the Isabelle Simplifier method *simp*. Moreover, `auto` in Coq refers to elementary intuitionistic reasoning with a few rules, while *auto* in Isabelle is a relatively strong combination of the Simplifier and Classical Reasoner with additional instrumentation provided by the library.

There is further potential for confusion concerning the syntactic (and conceptual) categories of language elements. Isabelle/Isar clearly distinguishes *theorem attributes* (for small forward rules or context declarations), *proof methods* (for structured backwards refinement), and *proof commands* (for primary proof structure). In Coq, almost everything is just a "tactic", including `apply` — but in Isabelle **apply** is a proof command to apply an arbitrary proof method in an unstructured situation.

Tools for **disproving** pending statements have become increasingly important in recent years. Current Isabelle routinely provides Quickcheck [8] and Nitpick [6]. These tools are also integrated into the Prover IDE, such that the system can provide information about faulty statements (with proposals for alternatives) produced asynchronously, while the user is working on the proof document in the vicinity.

▷ *What are the trends w.r.t. proof search methods for your kind of tool?*

The general trend, which has already a long tradition in Isabelle, is to combine many tools in the library: automated provers and dis-provers, general proof search and special decision procedures, connection to external proof tools with explicit proof reconstruction.

This leads to a considerable distance of Isabelle/HOL experienced by end-users, and the foundations of Isabelle/Pure sketched above (§2).

[5] `http://www.gilith.com/software/metis`
[6] `http://z3.codeplex.com/`

4 Proof Formats

▷ *Briefly describe the actual text formats used to output proofs.*

The classic *input* format for theories and proofs in LCF-style proof assistants is just the ML code that constructs the intended results. By storing the sources and replaying them eventually, the user can return to some particular situation in the formal context. Thus the input format produced by one user also serves as output format given to other users.

An alternative, but less portable format, is the "dumped world image" that the ML runtime system might provide. Dumped images were once popular in the LISP world (later also in Smalltalk), and might see renewed popularity due to some trends in general IDE design to support "live programming" with direct manipulation of the running system while it is developed incrementally.[7]

▷ *Are there standard formats in your community? If not, why not?*

The OpenTheory[8] project aims at a well-defined independent exchange format for HOL-based proof assistants [21]. This may be understood as HOL "object-code", so it lacks high-level structures of the original sources. It is also difficult to exchange proof tools, but there are some moves in that direction [25].

Fixed formats for interactive proof assistants are rare, due to the openess of the LCF approach. Although both Coq and Isabelle have some surface syntax, this "vernacular" of theory specifications and proofs is merely some default notation, with unlimited possibilities for extensions. In Isabelle, this open-endedness is particularly extreme: only the commands **theory** and **ML-file** are primitive, to start a new theory and load ML modules, and all other Isabelle/Pure and Isabelle/HOL language elements are bootstrapped from that.

In principle, Isabelle theory content is just a sequence of semantic operations to update the toplevel state consecutively (in a purely functional manner). The surface syntax is part of the command implementation, using computationally complete mechanisms (e.g. parser combinators) instead of fixed grammars.

This flexibility makes it difficult to build tools that operate on theory sources of LCF-style proof assistants. The situation is particularly difficult for the HOL family, where the input consists of unrestricted ML code. Nonetheless, in the past 5 years, the Isabelle/PIDE infrastructure has managed to provide systematic access to important aspects of the prover and its internal data structures, with continuous feedback in the text editor, e.g. see [43]. The PIDE approach of Isabelle/jEdit gives the user some illusion of direct manipulation of formal source text, despite the conceptual distance of the concrete syntax to the abstract logical entities behind it.

▷ *If possible and if not too large, please copy-paste the grammar of the proof format(s) here.*

It is always possible to invent specific formats for specific tools, and implement them with the means provided by ML. Current Isabelle actually uses a hybrid (two-legged) approach: Isabelle/ML for hard-core logical tools, and Isabelle/Scala for systems-programming on the JVM. For example, a simple XML-based exchange-format or parsers for concrete syntax could be implemented either in Isabelle/ML or Isabelle/Scala, using existing libraries that are available on both sides.

Bypassing official standards, the exchange of tree-structured data in Isabelle works conveniently via YXML/XML/ML data representation[9], which is already available for SML, OCaml, and Scala, and may be easily ported to other languages on the spot. So if there is any standard exchange format in Isabelle, it is YXML as transfer syntax for untyped XML trees [46, §6.11], which is used to encode algebraic datatypes. Actual content needs to be defined by the tools themselves.

[7] E.g. see the blog of Bret Victor `http://worrydream.com` — it also covers old Smalltalk concepts to some extent.

[8] `http://www.gilith.com/research/opentheory`

[9] `https://bitbucket.org/makarius/yxml`

▷ *Please copy-paste a small but interesting example proof here. Try to select an example that shows (most of) the peculiarities of your proof format. Use a verbatim environment (like the listings package, for example).*

We have already seen various Isabelle/Isar proof texts in §1 of this document, which is a pretty-printed version of a genuine theory. This is a proper PDF-LaTeX document with relatively high type-setting quality, not a verbatim listing.

The subsequent example of the Knaster-Tarski fixed-point theorem shall illustrate the different notions of proofs further. According to the textbook [10, pages 93–94], the Knaster-Tarski fixpoint theorem is as follows:

> **The Knaster-Tarski Fixpoint Theorem.** Let L be a complete lattice and $f\colon L \to L$ an order-preserving map. Then $\bigsqcap\{x \in L \mid f(x) \leq x\}$ is a fixpoint of f.
>
> **Proof.** Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigsqcap H$. For all $x \in H$ we have $a \leq x$, so $f(a) \leq f(x) \leq x$. Thus $f(a)$ is a lower bound of H, whence $f(a) \leq a$. We now use this inequality to prove the reverse one (!) and thereby complete the proof that a is a fixpoint. Since f is order-preserving, $f(f(a)) \leq f(a)$. This says $f(a) \in H$, so $a \leq f(a)$.

This proof is formalized in Isabelle/Isar as follows, streamlining the reasoning a bit to achieve structured top-down decomposition of the problem at the outer level, while only the inner steps of reasoning are done in a forward manner.

```
theorem Knaster-Tarski:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes mono f
  shows ∃ a. f a = a
proof
  let ?H = {u. f u ≤ u}
  let ?a = ⋂ ?H
  show f ?a = ?a
  proof (rule order-antisym)
    show f ?a ≤ ?a
    proof (rule Inf-greatest)
      fix x
      assume x ∈ ?H
      then have ?a ≤ x by (rule Inf-lower)
      with ⟨mono f⟩ have f ?a ≤ f x ..
      also from ⟨x ∈ ?H⟩ have ... ≤ x ..
      finally show f ?a ≤ x .
    qed
    show ?a ≤ f ?a
    proof (rule Inf-lower)
      from ⟨mono f⟩ and ⟨f ?a ≤ ?a⟩ have f (f ?a) ≤ f ?a ..
      then show f ?a ∈ ?H ..
    qed
  qed
qed
```

This format is called *primary proof text* in Isabelle/Isar jargon. It is notable that the reasoning merely composes some elementary rules of lattice theory into a structured proof outline, according to Natural Deduction rule composition in Isabelle/Pure. The only "proof automation" used here is unification, to search for suitable rules and instantiations from the context (the background theory or current proof).

The internal *proof term*, which is normally not seen nor stored in memory, shows the conciseness of the formal reasoning in its low-level form:

```
λ(H: -) Ha: -.
 exI · (λx. ?f x = x) · - · (thm · H) ·
```

```
(order-antisym · - · - · (thm · H) ·
  (complete-lattice-class.Inf-greatest · - · - · H ·
    (λx Hb: -.
      order-trans-rules-23 · - · - · - · (thm · H) ·
       (order-class.monoD · ?f · - · - · (thm · H) · (thm · H) · Ha ·
         (complete-lattice-class.Inf-lower · - · - · H · Hb)) ·
       (iffD1 · - · - · (thm.Set.mem-Collect-eq · - · (λu. ?f u ≤ u) · (thm · H)) · Hb))) ·
  (complete-lattice-class.Inf-lower · - · - · H ·
    (iffD2 · - · - · (thm.Set.mem-Collect-eq · - · (λa. ?f a ≤ a) · (thm · H)) ·
      (order-class.monoD · ?f · - · - · (thm · H) · (thm · H) · Ha ·
        (complete-lattice-class.Inf-greatest · - · - · H ·
          (λx Hb: -.
            order-trans-rules-23 · - · - · - · (thm · H) ·
             (order-class.monoD · ?f · - · - · (thm · H) · (thm · H) · Ha ·
               (complete-lattice-class.Inf-lower · - · - · H · Hb)) ·
             (iffD1 · - · - · (thm.Set.mem-Collect-eq · - · (λu. ?f u ≤ u) · (thm · H)) · Hb)))))))
```

Proof terms have little practical relevance today, so users don't mind to use stronger proof tools, even if the internal reasoning becomes less aesthetic. Isar proof texts allow to retain the clarity of a proof outline, independently of the proof tools that are used.

▷ *What are the guiding principles involved in the design of the format? Is it intended to be human-readable? easy to parse? easy/efficient to verify? as small as possible? In case of automated proofs, is it intended to recover intuition of the proofs?*

The guiding principles of the primary proof format — the Isar proof text — are as follows:

- Input produced by one user coincides with the output given to other users.
- Human-readability is emphasized, but it should coincide with efficient machine-checking.
- Structure is more important than strong automation, but arbitrary proof tools may be included on demand.
- Proofs are documents (with proper type-setting), not "scripts".

▷ *What are the trends?*

I see a general trend towards more structure. The Isar proof language was a relatively significant reform of the main Isabelle input 15 years ago, but Coq/SSReflect and classic Coq/Ltac have gradually provided a bit more explicit static information about the intended meaning of proof sources, beyond traditional "scripts".

Another trend is to go beyond static "proof formats", and understand the proof development process as genuine interaction of the user with the prover, which needs to be supported explicitly by some document-model in PIDE. Internal aspects of the prover are externalized on the spot, to let the editor participate. Isabelle/jEdit [43] is an important step towards that, but advanced structural editing of proof documents is still missing.

5 Proof Production

The LCF approach demands proper foundations of all reasoning performed in the system: all proof tools need produce some explicit proof record, and have the core logic accept it. There are two main approaches to that problem.

1. **Explicit replay.** The proof tool reduces its reasoning at run-time to primitive inferences of the LCF kernel. The results of the proof search are turned into a proof trace of some form, and passed through the abstract datatype *thm* to infer proper theorems (*without* storing a proof trace by default). The separation of proof discovery versus justification is important: finding a solution is usually much harder than checking it by explicit inferences. We see here a practical consequence of "*NP* vs. *P*" from complexity-theory.

2. **Reflection.** The proof tool is implemented within the logical language of the prover itself and formally proven correct for all input values. Tool applications then turn some symbolic representation of the problem into a theorem, but without performing explicit inferencing again. The proof is finished beforehand, to establish the correctness of the implemented algorithm (e.g. by induction).

Explicit replay is the standard approach of LCF, the HOL family, and Isabelle: tools themselves are not proven, but produce explicit proof steps at run-time. Thus the tool might fail or diverge unexpectedly, but cannot return wrong results.

Reflection is routinely used in Coq, since its stronger type-theory facilitates that and the access to ML as tool implementation language is more difficult than in the other systems.

After many years of both approaches side-by-side, there is no clear "winner". Explicit proof replay might sound less efficient in theory, but usually performs well by in practice. For reflection there is first the hurdle to produce a formal correctness proof of the implemented tool (including the parts that merely search for a solution). Afterwards its symbolic evaluation needs to be performed by built-in normalization of the logical language, using byte-code interpretation instead of native ML code.

▷ *What is the "price" (e.g. in terms of efficiency or memory overhead) paid to generate proofs? How much slower does the tool become? How much more memory does it consume?*

By default, a true LCF-style proof assistant does not need to store proof objects, since all inferences are "correct by construction". An explicit proof record may be requested nonetheless, e.g. for external checking with some independent tool. The cost for that is typically an order of magnitude: a factor 2–10 in time and space requirements. Naive proof storage would require much more space, but extra time is used to maintain reasonably compact proof terms (e.g. see [4]).

▷ *In case of automated proofs, how different is the proof-producing proof search algorithm from the proof search algorithm that simply answers "yes" or "no"?*

For properly implemented proof tools that use the LCF-style inference kernel as intended, there is no difference in the algorithm. It is merely a matter to change some system flags, and to spend more machine resources to turn the platonistic idea of a proof into a materialized proof object.

▷ *Is it usually possible to switch proof production on and off in your kind of tool?*

Yes, there are various degrees of thoroughness in formal proof checking. In Isabelle the main options are as follows:

- **Skip proofs:** the user merely wants to see some checking of theory specifications, without going into proofs.
- **Quick-and-dirty:** tools are allowed to omit explicit inferences or proper definitions, and replace them by ad-hoc oracle invocations or axioms.
- **Default LCF mode:** all inferences go through the LCF kernel (*without* storing proof traces) and definitions are done properly.
- **Proof terms:** LCF kernel inferences are stored as compact λ-terms for later inspection, extraction etc.

▷ *What are the trends?*

With routine support for parallel LCF-style proof checking in Isabelle, the default LCF mode has regained its central role, and the weaker and stronger options are becoming less important. On the one hand, parallel proof checking is fast enough to render most quick-and-dirty shortcuts obsolete. On the other hand, the practical irrelevance of proof terms (their inaccessibility) is important for scalability of really big proof developments. Retrospectively, the classic LCF-approach from 1979 fits nicely to multicore programming with shared memory from 2009, see also [44].

So far there are only Isabelle and ACL2 with realistic support for parallel processing [37]. Hopefully, other proof assistants will manage to catch-up eventually, otherwise that will become a dividing line for slow sequential systems vs. fast parallel ones.

6 Proof Consumption

▷ *Does your kind of tool consume proofs from another kind of tool? (e.g. SMT-solvers using (proofs from) sat-solvers; higher-order ATPs and proof assistants using (proofs from) first-order ATPs; . . .)*

Yes, the most prominent examples for that is the integration of Z3 via the *smt* proof method [7], and Sledgehammer [5]. Both are notable in using actual external tools, and recovering some internal LCF-style proofs from the result.

▷ *If so, is there anything that could be improved in the proofs that are generated by this other kind of tool?*

This question needs to be asked to the people behind particular tool integrations. Whenever such a project is started, there are usually technical problems and genuine doubts about the proof trace of the external tool, which usually require several iterations to incorporate some tool reliably.

▷ *If your tool consumes proofs from other tools, is it only a proof-checker or is it a wider tool also used to build proofs by its own? If it is a wider tool, do you also provide a lighter version of your tool that is dedicated to proof-checking? If not, would it be desirable?*

Tools integrated into an interactive prover like Isabelle are usually for genuine production use in the target system, to support the user in producing big formalizations. So the main purpose is to make an external tool look like an internal one. Independent checking of the external tools could in principle be done as well, but the practical relevance is much lower.

▷ *What are the trends?*

Big ITP systems are becoming more and more a "workbench" for many different proof tools, both internal and external ones. Thus the classic proof assistants are the canonical "sink" for proofs produced by other systems, but there is also the possibility to export their reasoning again as a trace of the inference kernel.

Integration of tools in practice has two aspects: (1) the actual work, and (2) technical side-conditions to glue together quite dissimilar programs into an actual system. The Isabelle infrastructure is particularly polished to accommodate tools stemming from very mixed background, with uniform support for the main operating system families: Linux, Windows, Mac OS X.

7 Proof Applications

▷ *Which application domains have used your kind of tool? Have there been any ground-breaking achievements? Among these application domains, which are particularly interested in the generated proofs? How do they use the proofs? Do they use proofs just for certifying the correctness of the provided answer? Or do they extract other kinds of information (e.g. unsat cores, interpolants, witnesses, programs, . . .) from proofs?*

ITP systems are open-ended and the scope of applications depends on the imagination of their users. Despite some big and publicly visible applications, the main potential is probably still unexplored, due to traditional obscurity and inaccessibility of the ITP systems (which was inherited from LCF). Applications today are typically about formalized mathematics, modelling computer-science concepts, or specification and verification of software / hardware systems. Here is a list of some large-scale proof development projects (project leaders and proof assistant in parentheses).

- The **Flyspeck Project** `https://code.google.com/p/flyspeck` (Tom Hales, HOL-Light): formal proof of Kepler's Conjecture [18, 19].
- The **L4.verified project** `http://ertos.nicta.com.au/research/l4.verified` (Gerwin Klein, Isabelle/HOL): formally correct operating system kernel [23].

– The **Feit-Thompson Odd Order Theorem** `http://www.msr-inria.fr/news/feit-thomson-proved-in-coq` (Georges Gonthier, Coq/SSReflect) [12, 11].
– The **CompCert verified compiler** `http://compcert.inria.fr/doc` (Xavier Leroy, Coq): optimizing C-compiler for various assembly languages, written and proven in the functional language of Coq [26, 27].

There are also some efforts to build libraries of formalized mathematics. The Feit-Thompson proof depends on the *Mathematical Components* library that was developed by the same research group. For Isabelle/HOL, the *Archive of Formal Proof*[10] has accumulated quite substantial material over 10 years, from many different sources and different degrees of quality and sophistication.[11]

▷ *In the context of these applications, when should a proof P of a certain theorem be considered better than a proof Q of the same theorem? For example, in case of double negation transformations, why is it better to get intuitionistic proofs from classical proofs?*

To answer this question from the Isabelle/Isar perspective, recall that there are different notions of formal proof.

– The **primitive proof** (or **proof object**) is some way to ensure that results produced by the system are reliable records of logical reasoning.
The standard LCF approach does not require to record proof objects explicitly, because the abstract datatype of theorems ensures that its elements enjoy certain properties by construction (thanks to strongly-typed virtues of Standard ML).
– The **primary proof** (or **proof document**) is the main source text that is processed by the system to accept a formal proof, and to tell the human reader that the reasoning makes sense. Proof documents are suitable for "archival publication" of formal proofs, and have a chance to be useful for other proof assistants or other versions of the same proof assistant even after some time.

Isabelle/Isar is mostly concerned about proof documents, i.e. the real sources of some formal reasoning, in a form that is both human-readable and machine checkable (to turn it into an abstract proof object that is not stored).

The relation of "proof document P is better than Q" is not formally defined, but may be understood as a natural continuation of quality standards for program source text, with the following criteria:

– Readability for humans, not just the machine.
– Maintainability and stability: small changes of theory definitions lead to reasonably small changes of proofs.
– Generality: existing proofs for one theory may be easily re-used in a more general theory by reworking the proof text.
– Performance of proof checking: Isabelle/Isar is faster in checking nicely structured proof documents than unstructured tactic scripts.

Of course, people have also applied a standard repertoire from Proof Theory to the internal proof objects of various ITP systems, including Isabelle [4, 3]. This is interesting in its own right, but merely a niche of the full spectrum of ITP applications.

▷ *What are the trends?*

For the near future I see the need for more systematic tool support to maintain the growing repositories of formalized mathematics. There is also a need to make proof assistants more accessible to more users, to allow easy browsing and editing of such libraries.

[10] `http://afp.sourceforge.net/`
[11] At the time of writing (26-May-2014), AFP consists of 176 articles in 1899 source files, which comprise 46 MB total. Checking in batch mode takes approximately 10h CPU time and 1h elapsed time on a solid 8-core Intel Xeon workstation.

Current Isabelle addresses that by its default Prover IDE: Isabelle/jEdit is intended for beginners and experts alike. It allows to edit theory libraries directly, with continuous proof checking on multiple cores. "The ACL2 Sedan"[12] is a similar approach to Eclipse-based IDE support for ACL2, with the notable difference that it is meant as a comfortable "family car", while the traditional Emacs interface remains available as "race car" for experts.

Watching the ITP community over almost 20 years, my impression is that the logical foundations at the bottom matter very little today. For example, Coq started out very constructionistic, but the majority of its users don't understand much of that. In applications, Coq users often include a prelude of classic axioms for their application theory. e.g. for *real* real numbers in classic analysis. Even if the application is as constructive and computational as some floating-point arithmetic algorithms, the mathematics of the problem domain works more smoothly with all the tools of classical reasoning that have accumulated over a long time.

8 Conclusions

▷ *Summarize the most important points of the previous sections.*

Taking even just a single ITP system like Isabelle, it is difficult to pin down its boundaries precisely, to say what it is and what it does. Too many aspects of formal logic and proof development have been added over the decades. Today one could characterize Isabelle and its Prover IDE as an integrative platform for domain-specific formal languages and tools around them, but it depends on many side-conditions how the system is perceived in a particular application scenario. For example, Isabelle/jEdit could be used as IDE for Standard ML, without any theory or proof development.

Each proof assistant has its own tradition and culture. New users should spend some time to develop a sense for more than one accidental candidate, before making a commitment: substantial time will be required to become proficient with any of these systems. Old users should try to learn how other proof assistants actually work, and what are their specific strengths and weaknesses. There are very few people who are proficient with more than one ITP system, and the exchange of ideas within the various brands of proof assistants is often slow and cumbersome.

References

1. D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 1785 of *LNCS*. Springer, 2000.
2. C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, LNAI 4108, 2006.
3. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
4. S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *LNCS*, pages 38–52. Springer, 2000.
5. J. C. Blanchette. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL.* `http://isabelle.in.tum.de/doc/sledgehammer.pdf`.
6. J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[12] `http://acl2s.ccs.neu.edu/acl2s/doc`

7. S. Böhme and T. Weber. Fast LCF-Style proof reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.
8. L. Bulwahn. The new Quickcheck for Isabelle – random, exhaustive and symbolic testing under one roof. In C. Hawblitzel and D. Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.
9. L. Damas and H. Milner. Principal type schemes for functional programs. In *ACM Symp. Principles of Programming Languages*, 1982.
10. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
11. G. Gonthier. Engineering mathematics: the odd order theorem proof. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013.
12. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.
13. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2), 2010.
14. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
15. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
16. F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *LNCS*. Springer, 2007.
17. F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In S. Berardi, F. Damiani, and U. de Liguoro, editors, *Types for Proofs and Programs, TYPES 2008*, volume 5497 of *LNCS*. Springer, 2009.
18. T. C. Hales. Formalizing the proof of the kepler conjecture. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, page 117. Springer, 2004.
19. T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.
20. P. V. Homeier. The HOL-Omega logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2009.
21. J. Hurd. The OpenTheory standard theory library. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods – Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
22. F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *LNCS*. Springer, 1999.
23. G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.
24. R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Hol with definitions: Semantics, soundness, and a verified implementation. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, volume 8558 of *LNCS*. Springer, 2014.
25. R. Kumar and J. Hurd. Standalone tactics using OpenTheory. In L. Beringer and A. P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 405–411. Springer, 2012.
26. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, pages 42–54. ACM, 2006.
27. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
28. D. C. J. Matthews and M. Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010), co-located with POPL*. ACM Press, January 2010.
29. T. Nipkow. *Programming and Proving in Isabelle/HOL*. `http://isabelle.in.tum.de/doc/prog-prove.pdf`.
30. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
31. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
32. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
33. L. C. Paulson. Set theory for verification: I. from foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.
34. L. C. Paulson. Set theory for verification. II: induction and recursion. *J. Autom. Reasoning*, 15(2):167–215, 1995.
35. L. C. Paulson. Generic automatic proof tools. *CoRR*, cs.LO/9711106, 1997.
36. L. C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.
37. D. L. Rager, W. A. Hunt, and M. Kaufmann. A parallelized theorem prover for a logic with parallel execution. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*. Springer, 2013.
38. D. Traytel, S. Berghofer, and T. Nipkow. Extending Hindley-Milner Type Inference with Coercive Structural Subtyping. In H. Yang, editor, *APLAS 2011*, volume 7078 of *LNCS*, pages 89–104, 2011.
39. M. Wenzel. *The Isabelle/Isar Reference Manual*. `http://isabelle.in.tum.de/doc/isar-ref.pdf`.
40. M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *LNCS*. Springer, 1999.
41. M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*. ACM Digital Library, 2009.
42. M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2011)*, volume 6824 of *LNAI*. Springer, 2011.
43. M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, volume 7362 of *LNAI*. Springer, 2012.
44. M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.
45. M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, volume 8558 of *LNCS*. Springer, 2014.
46. M. Wenzel and S. Berghofer. *The Isabelle System Manual*. `http://isabelle.in.tum.de/doc/system.pdf`.
47. F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNAI*. Springer, 2006.