Strategic Principles in the Design of Isabelle

Lawrence C. Paulson
lcp@cl.cam.ac.uk

Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, England

Abstract. Interactive proof assistants can support proof strategies, if the right primitives have been included. These include higher-order syntax, logical variables and a choice of search primitives. Such a system allows experimentation with different automatic proof methods, even for constructive logics, new variable-binding operators, etc. The built-in unification and search make proof procedures easy to implement, typically using tableau methods. Against subgoals that arise in practice, even straightforward heuristics turn out to be powerful.

1 Introduction

This paper describes the role of strategic principles in proof assistants, taking Isabelle as an example. I use 'strategic' in two senses: first, in the sense of proof strategies, and second, in the sense of decisions that have long-term consequences. Building a proof assistant is a major investment, and we cannot foresee at the outset precisely how it will be used. So we must identify those features that seem likely to be of potential benefit, and (if we are in the academic world) of scientific interest.

Key objectives are *automation* and *flexibility*. Why automation? Because Isabelle is interactive, and formal proofs are long; obvious formulas, at least, should be proved automatically. Why flexibility? Because users will have different problems and goals from those we have foreseen. They need not just a flexible syntax but control over deep properties of their formalism: it might be first-order or higher-order, constructive or classical, etc. A flexible system gives users a choice of tools to use for automating their chosen formalism.

Soundness is obviously necessary: a proof assistant should deliver correct results and should behave in a way that inspires trust. It should also be efficient in space and time, capable of supporting realistic proofs in a range of applications.

The desire for automation and flexibility in Isabelle has resulted in the inclusion of three basic features:

- higher-order syntax
- logical variables and unification
- search primitives based on lazy lists

These have interesting implications, especially when combined in one system, as we shall see. Automation and flexibility have also affected the design of Isabelle's more specialized tactics, such as the rewriter.

PVS [7] provides automation in the form of special-purpose decision procedures. These are valuable, but the great majority of theorems cannot be proved using decision procedures alone. The rest of the system must provide sufficient automation to deal with more general problem domains.

2 Higher-Order Syntax

Most provers support quantifiers, and a few go further. A *higher-order* syntax allows users to define new variable-binding constructs such as $\mathsf{least}\ n\ P(n)$. A higher-order syntax will typically be based upon some form of typed λ-calculus.

Most research still seems to be devoted to first-order systems and, if I may be provocative, this is a curious state of affairs. First-order logic is not expressive: consider how powerful resolution provers are, and how few applications they have. Adding set theory to first-order logic yields an expressive system [14, 16], but formalizing concepts such as set comprehension, $\{x \in A \mid P(x)\}$, and general union, $\bigcup_{x\in A} B(x)$, requires a higher-order syntax. Approaches based on first-order syntax are not attractive—we might have a clumsy language of combinators or be forced to define an auxiliary function every time we want to write a comprehension.

In spite of its strengths, set theory is unfashionable. Higher-order logic has an established track record, especially for hardware verification; it is the basis of interactive verification tools such as PVS and HOL-Light [2]. But it is important to remember that higher-order syntax does not force us to use higher-order logic. It merely allows us to define variable-binding constructs, and that provides flexibility.

3 Logical Variables and Unification

By *logical variables* I mean that certain variables in formulas can be automatically instantiated using unification. The automated reasoning community is well aware of their benefits:

- *don't know* parts of a goal can be left unspecified until later
- unification can complete those parts automatically
- proof procedures (such as free-variable tableaux) can exploit unification

Logical variables have a further advantage that is relevant to Isabelle. They allow inference rules to be represented declaratively, as a generalization of Horn clauses. For example, Isabelle/HOL defines the bounded quantifier $\forall_{x\in A} P(x)$ to be $\forall x\,[x \in A \to P(x)]$ and derives the rule

$$\frac{\forall_{x\in A} P(x) \quad a \in A}{P(a)}$$

A declarative representation supports many operations on rules. They can be displayed. They can be used forwards (to derive theorems from theorems) or

backwards (to derive subgoals from goals). Proof tools can classify and transform rules automatically. The cost of applying rules is independent of the computational effort expended to prove them in the first place.

Other systems represent rules procedurally, as code to transform formulas or subgoals. The best they can do is to exploit higher-order quantification, expressing the rule as a formula:

$$\forall P\, \forall a\, \forall A\, [\forall_{x \in A} P(x) \wedge a \in A \to P(a)]$$

This form is declarative, but it only works in higher-order logic.

Despite the advantages of logical variables, many new tools lack them. Unification is hard to provide in the presence of higher-order syntax. Isabelle performs *higher-order unification*, using Huet's procedure [3]. It is curious that such an important and useful procedure has received so little attention. Perhaps higher-order unification's theoretical limitations are to blame. It is semidecidable and sometimes yields infinitely many unifiers. But Isabelle seldom encounters these hard cases. *Pattern unification* is a refinement that handles only the easy cases: there is a fairly simple decision procedure, which yields at most one unifier. Miller introduced pattern unification in his logic programming language L_λ [4]. Isabelle uses pattern unification [5] to reduce the number of calls to Huet's procedure. One could base a proof assistant on pattern unification alone: Nipkow has found that it covers 97% of unifications in Isabelle.

The HOL family of provers [2] follows a strict philosophy intended to ensure soundness. In part, it demands that all proof steps should go through a trusted kernel of minimal size. Isabelle follows a similar philosophy (otherwise you get a tool in which users prove 0=1 every few months), but allows the kernel to contain a sophisticated unification algorithm.

The difficulty of combining higher-order syntax with logical variables reminds us that that distinct goals (flexibility and automation) may be hard to achieve simultaneously.

4 Search Primitives Based on Lazy Lists

Many automated reasoning systems employ search, but there are many different strategies. A programmer could hardly cope with a version of Prolog that used anything other than depth-first search. Stickel's PTTP [18] brought depth-first iterative deepening into fashion. Some Isabelle reasoning tactics use best-first search. Flexibility requires giving the user a choice of search strategies.

A *lazy list* is a list (possibly infinite) whose elements are computed upon demand rather than all at once. Lazy lists provide a uniform interface to an arbitrary search strategy. (Most functional programmers know this; for examples see my ML book [9, Chap. 5].) The idea is to implement your search strategy so that it yields up its results as a lazy list. The recipient of the list, just by looking through the elements, incrementally invokes the search strategy. Isabelle uses lazy lists to return, in a prescribed order, possibly infinitely many higher-order unifiers.

In Isabelle, proof strategies are called *tactics*. An Isabelle tactic can generate a lazy list of results. The simplest tactics consist of the application of a single rule to a goal. Users can combine tactics using simple control structures such as THEN and ORELSE. The latter tactical introduces choice, thereby defining a search space. They can explore this search space using primitives such as REPEAT, DEPTH_FIRST, BEST_FIRST, ITER_DEEPEN, etc. Nor is this menu fixed: Isabelle is coded in ML, and users can code their own search strategies. For example, Norbert Völker has implemented the A^* search strategy [17], which is a form of best-first search.

5 The Benefits

The applications of strategic design choices can't be predicted precisely. But it is obvious that a system that combines search strategies and unification will have some potential for automatic proof search. Add interaction and higher-order syntax, and you have an ideal test-bed for experimentation with different proof strategies.

In early work I focussed on intuitionistic logic, independently discovering Dyckhoff's techniques [1] (which he went on to prove complete). Turning to classical logic, it was not hard to build simple tableau provers. But such provers are practically useless, since users don't work in pure first-order logic: they use set theory and other complex notions of their own. Further experimentation led to Isabelle's *classical reasoner* [10]. Its tactics, such as `fast_tac`, achieve the necessary integration—users can give it any reasonable inference rules, represented declaratively as discussed in Sect. 3. The details do not concern us here; let me just stress that its heuristics are elementary by modern standards, and that its power comes from the combination of higher-order syntax, unification and search primitives.

Recently I have built a new tactic, `blast_tac` [11]. It finds proofs using its own generic tableau prover, which is coded directly in ML for speed. Because it cuts corners, it converts the resulting proof into tactics for verification using Isabelle's unification and search primitives. Perhaps one could add such a tactic to a proof assistant that lacked logical variables, but the job would be harder.

Though it cannot prove challenge problems that feature in theorem prover competitions, `blast_tac` is valuable against subgoals that arise in practice [6]. The majority of systems entered in such competitions could not even parse, let alone prove, these subgoals, since they typically involve complex variable binding. Often the proof involves concepts (such as inductively defined relations) that cannot be expressed in first-order logic. Consider this example, formulated in ZF set theory [8]:

$$C \neq \emptyset \to \bigcap_{x \in C} [A(x) \cap B(x)] = (\bigcap_{x \in C} A(x)) \cap (\bigcap_{x \in C} B(x))$$

Back in 1988, proving it required manually replacing $C \neq \emptyset$ by $\neg(C \subseteq \emptyset)$, then applying two repetitive tactics. Now, it takes `blast_tac` half a second. I suspect it is still a hard challenge for other set theory provers.

Isabelle supports higher-order logic as well as set theory, and the same classical reasoner works for both. Many set theory problems become easier to prove in higher-order logic, but perhaps they are still challenging. Some examples appear in my `blast_tac` paper [12].

I have also implemented model elimination (MESON) on Isabelle's proof engine. Compared with PTTP it is absurdly slow, but it crushes `blast_tac` on harder first-order challenge problems. Unfortunately, one seldom encounters pure predicate logic tasks that can be given to the MESON tactic. Because it is generic, `blast_tac` finds uses throughout the Isabelle case studies. This paradox illustrates the irrelevance of traditional challenge problems to real-world verification.

6 Conclusions

Isabelle's combination of features—higher-order syntax, logical variables and search primitives—is ideal for trying out new proof strategies. It is less ideal for producing high-performance tools. Implementing `blast_tac` required a direct resort to ML. The same thing happened with the simplifier: early versions ran as logic programs upon Isabelle's proof engine, but it too is now coded in ML for speed. We can expect most proof procedures to be expressed ultimately as algorithms and implemented as straight code.

Of course, the final implementation should retain the new ideas discovered during experimentation. Isabelle's simplifier has unusual features, such as congruence rules [13, Sect. 4.5]. These express contextual rewriting, for instance, given $A \wedge B$, to assume A to be true while rewriting B. Congruence rules were discovered back when the simplifier ran on Isabelle's proof engine.

Beginning users do not conduct research on automatic proof, but even they benefit from Isabelle's combination of features. Higher-order syntax is indispensable in any general-purpose specification language. Logical variables arise naturally, and can be left to be instantiated automatically. Even search can arise naturally. Giving Isabelle a list of rules makes it search for the first one that is applicable, and joining two tactics using THEN can cause a bit of search to find a route through both tactics. Over time, users learn to undertake more adventurous searches.

Acknowledgement. Isabelle is the outcome of many years' collaboration with Tobias Nipkow and his group at the Technical University of Munich. Isabelle has been supported by numerous grants, including the EPSRC grant GR/K57381 'Mechanizing Temporal Reasoning' and the ESPRIT working group 21900 'Types.' Tim Leonard made detailed suggestions on this paper; James Margetson and Norbert Völker also commented.

References

1. Dyckhoff, R., Contraction-free sequent calculi for intuitionistic logic, *Journal of Symbolic Logic* **57**, 3 (1992), 795–807

2. Gordon, M., From LCF to HOL: a short history, In Plotkin et al. [15], In press
3. Huet, G. P., A unification algorithm for typed λ-calculus, *Theoretical Comput. Sci.* **1** (1975), 27–57
4. Miller, D., A logic programming language with lambda-abstraction, function variables, and simple unification, *Journal of Logic and Computation* **1**, 4 (1991), 497–536
5. Nipkow, T., Functional unification of higher-order patterns, In *Eighth Annual Symposium on Logic in Computer Science* (1993), M. Vardi, Ed., IEEE Computer Society Press, pp. 64–74
6. Nipkow, T., Verified lexical analysis, In *Theorem Proving in Higher Order Logics: TPHOLs '98* (1998), J. Grundy M. Newey, Eds., LNCS, pp. ??–??, invited lecture
7. Owre, S., Rushby, J. M., Shankar, N., Srivas, M. K., A tutorial on using PVS for hardware verification, In *Theorem Provers in Circuit Design: Theory, Practice, and Experience* (1995), R. Kumar, Ed., LNCS 901, Springer, pp. 258–279
8. Paulson, L. C., Isabelle: The next seven hundred theorem provers (system abstract), In *9th International Conference on Automated Deduction* (1988), E. Lusk R. Overbeek, Eds., LNCS 310, Springer, pp. 772–773
9. Paulson, L. C., *ML for the Working Programmer*, 2nd ed., Cambridge University Press, 1996
10. Paulson, L. C., Generic automatic proof tools, In *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, R. Veroff, Ed. MIT Press, 1997, ch. 3
11. Paulson, L. C., A generic tableau prover and its integration with Isabelle, Tech. Rep. 441, Computer Laboratory, University of Cambridge, Jan. 1998
12. Paulson, L. C., A generic tableau prover and its integration with Isabelle (extended abstract), In *CADE-15 Workshop on Integration of Deduction Systems* (1998), in press
13. Paulson, L. C., Tool support for logics of programs, In *Mathematical Methods in Program Development: Summer School Marktoberdorf 1996*, M. Broy, Ed., NATO ASI Series F. Springer, Published 1997, pp. 461–498
14. Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *Journal of Automated Reasoning* **17**, 3 (Dec. 1996), 291–323
15. Plotkin, G., Stirling, C., Tofte, M., Eds., *Essays in Honor of Robin Milner*, MIT Press, 1998, In press
16. Quaife, A., Automated deduction in von Neumann-Bernays-Gödel set theory, *Journal of Automated Reasoning* **8**, 1 (1992), 91–147
17. Rich, E., Knight, K., *Artificial Intelligence*, 2nd ed., McGraw-Hill, 1991
18. Stickel, M. E., A Prolog technology theorem prover: Implementation by an extended Prolog compiler, *Journal of Automated Reasoning* **4**, 4 (1988), 353–380