Finding proofs and checking proofs

John Harrison
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14a, 20520 Turku, Finland
`jharriso@abo.fi`
`http://www.abo.fi/~jharriso/`

Abstract. The classic dichotomy between proof discovery and proof presentation resurfaces in automated deduction. The LCF style of theorem proving offers a promising way to integrate formal deductive proof with higher-level algorithmic or heuristic forms of mathematical activity. One way of dealing with the potential inefficiency of a reduction to a formal deductive system is to exploit the separation of proof finding and proof checking. This can be relevant both to the implementation difficulty of computer proof assistants, and the speed with which they run. Exploiting the separation makes it practical to use many of the usual algorithmic and heuristic approaches to mathematical problem-solving while maintaining the soundness of a formal deductive system. There is an interesting analogy with the NP and co-NP complexity classes.

1 Forms of mathematical activity

Classical deductive presentations of mathematics, the style of which goes back to Euclid, often show little trace of how the results and the proofs were arrived at. Sometimes the process of discovery is very different, as shown in the study of Euler's theorem on polyhedra by Lakatos [19]. For example, Newton is popularly believed to have arrived at most of the theorems of *Principia* using calculus, but to have translated them into a purely geometric form so that they would be more easily understood, or more readily accepted as rigorous, by his contemporaries. Often the reasons are connected with the aesthetics of the deductive method; for example Gauss compared other considerations besides the deductive proof to the scaffolding used to construct a beautiful building, which should be taken down when the building is finished.

This situation is rather strange. Mathematics is distinguished from other sciences by its emphasis on deductive logic rather than inductive, abductive or empirical reasoning. But it seems that creative and problem-solving activities in mathematics are often more successful if they do not need to be shoehorned into a deductive presentation. Such a presentation is regarded as useful principally for two reasons: for the avoidance of error, and for the systematization and collation of mathematical knowledge.

Indeed in recent times, formalizability in some system such as ZF set theory has become widely accepted as *the* ultimate arbiter of correctness for mathematical reasoning. Bourbaki [3] clearly says that 'the correctness of a mathematical text is verified by comparing it, more or less explicitly, with the rules of a formalized language', and Mac Lane [20] says almost the same thing (p. 377):

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

We will not pause to debate this position, which in view of Gödel's theorems certainly seems to demand some justification. We will simply observe that in practice, mathematicians almost never attempt such complete formalization — in fact, probably relatively few of them could even recite the axioms of ZF set theory. Probably the most sustained formal development of mathematics is the *Principia Mathematica* of Whitehead and Russell [32], but this book is large and difficult to read, while still only containing relatively elementary mathematics. Possibly mathematicians do have a highly developed intuition about when a mathematical proof could *in principle* be formalized completely. But in practice mathematicians often think in a quite different way. In particular:

- Mathematical creativity is often disconnected from any notion of formal deduction, and motivated by physical pictures or amorphous daydreams.
- In mathematical problem-solving, e.g. differentiating and integrating, there is a tendency to rely on well-known algorithms and heuristics which are assumed sound and useful, without analyzing them on each occasion.

Therefore in attempting to analyze mathematical activity, it is interesting to consider the relationship between these different forms of thinking. And if it is desired to provide computer support for them all, we must ask how to mix them together. One possibility is to insist on fully formal deductive proofs at all times, but delegate the production of such proofs to the computer (whose speed and accuracy well equip it for the task), while allowing humans to interact with the computer at a more appealing level. We will consider in detail one fairly successful method of achieving this.

2 LCF-style theorem proving

There is a traditional divide in the world of computer theorem proving between 'automated theorem provers', which attempt to prove theorems without user assistance, and 'interactive theorem provers' or 'proof checkers' which merely assist the user in the production of a formal proof, acting as a humble clerical assistant.

There are numerous automated provers for first order logic based on tableaux, resolution and model elimination, as well as NQTHM

© 1996 G. Granger and J. Smith
ECAI 96. *12th European Conference on Artificial Intelligence*
Workshop on Representation of Mathematical Knowledge
Edited by H. Stoyan, K. Homann, S. Jacob, M. Kerber

[4] which proves theorems in quantifier-free first order arithmetic. These have been fairly successful: first order provers have produced new mathematical results in certain fields [1], while NQTHM has been used for an enormous range of tasks including real-life system verification as well as checking theorems from pure mathematics. However in order to tackle real problems, it is necessary to lead automated provers to results via a carefully graded series of lemmas; in general these systems are difficult to control and integrate with specialized algorithms without access to their internals. Moreover, given the complicated and often highly optimized innards of these systems, their correctness can be difficult to guarantee.[1]

Notable interactive provers are Automath [7], Stanford LCF [23] and Mizar [31]. These have often proved successful vehicles for the formalization of mathematics, especially Mizar, which has amassed an unrivalled corpus of machine-checked mathematics. However these systems are relatively labour-intensive to use, since the user must direct the proof at quite a low level. In particular, no facilities are provided for automating special-purpose inference patterns, which often makes proofs tedious.

The Edinburgh LCF system [12] was an attempt to combine the best features of both, and moreover, to make it possible for ordinary users to incorporate their own domain-specific extensions without compromising the security of the logic. The original system implemented Scott's Logic of Computable Functions (hence the name LCF), but as emphasized by Gordon [10], the LCF approach is applicable to any logic. There are now numerous descendants such as Coq and Nuprl which implement constructive type theories, and HOL [11], the system we ourselves are involved with, which implements classical higher order logic. A slightly more radical member of the LCF family is Isabelle [24] which allows multiple object logics embedded in a higher order metalogic.

In LCF-like systems, the ML programming language is used to define data types representing logical entities such as types, terms and theorems.[2] A number of ML functions are provided which produce theorems; these implement primitive inference rules of the logic. Thus, provided only the primitive ML inference rules are used, any theorem resulting will be 'correct'. Such adherence to the primitive rules is enforced by encoding theorems as an ML abstract datatype whose only constructors are the primitive inference rules of the logic. Anything of type `thm` must have arisen by applying the primitive rules.

This trustworthiness is attractive, but proofs of nontrivial facts in terms of primitive inferences are often extremely tedious, and the approach so far offers few, if any, advantages over a simple proof checker. However since a full programming language is available, one can program complicated patterns of inference, provided they ultimately decompose into primitives. Hence one can write derived inference rules which work at a higher level. In order to exploit them, a user may just call the functions, without understanding how they decompose into primitives. Only the original implementor of the derived rule need understand that. Because of the abstract datatype, a user can have equal confidence in the correctness of the resulting theorems, since ultimately they must arise by primitive inference. The user or derived rule is still free to decompose and rebuild terms and formulas in any way desired in order to decide on proof strategy etc. — and the extremely simple term structure of HOL makes this very convenient. Nevertheless all *theorems* must be built by primitive inference.

In this way, the LCF approach can offer the reliability and controllability of a low level proof checker together with the power of a more sophisticated prover. For example, HOL has derived rules for rewriting, associative-commutative rearrangement, linear arithmetic, tautology checking, inductive definitions and free recursive type definitions, among others. Moreover it can support different proof styles such as goal-directed proof, all ultimately decomposing to primitives, and can easily be linked to or incorporated within other systems because everything is accessible via ML — yet because of the abstract type `thm`, no false 'theorems' can arise. Hence LCF seems an ideal vehicle for the integration of deductive reasoning with specialized algorithmic and heuristic methods, which can be built on top in an elaborate edifice.

However, the LCF approach forces *all* reasoning to decompose to primitives. What grounds do we have for believing that this is practical even for a computer? It's well-known that computers are amazingly fast, and getting faster, but it's equally well-known that their speed cannot defeat problems which have an inherent combinatorial explosion. Can algorithms be written in this style, and are the resulting proofs of a size and complexity which is tractable? Perhaps surprisingly, it often is practical — a detailed analysis and a comparison with other techniques such as *reflection* is given by Harrison [15]. We can identify two clear reasons why the apparent difficulties are often not really serious.

First, complex patterns of inference can often be encoded as single theorems, proved once and for all. These can then be instantiated for the case in hand rather cheaply. This idea has long been used by HOL experts; for an early example see [22]. A more recent, and complex, example is the following [13] justifying a step in a quantifier-elimination procedure for the reals; it states that a finite set (list) of polynomials are all strictly positive throughout an interval if and only if they are all positive at the middle of the interval and are nonzero everywhere in it.

$$\begin{array}{ll}\vdash \forall l, a, b. & a < b \wedge \\ & (\forall x.\, a < x \wedge x < b \Rightarrow \text{FORALL (POS } x)\ l) \\ & = \;\; a < b \wedge \\ & \quad\;\; \text{FORALL (POS } (\frac{a+b}{2}))\ l \wedge \\ & \quad\;\; \neg\exists x.\, a < x \wedge x < b \wedge \text{EXISTS (ZERO } x)\ l\end{array}$$

Many special-purpose algorithms can be understood as a natural process of transformation. It is not difficult to store theorems which justify the individual steps; the algorithm can often be implemented almost as one would do without the primitive inference steps, and generally with a modest linear slowdown (there are a few delicate points over the efficiency of equality testing, but that's more or less true). This applies, for example, to HOL's implementation of rewriting and definition of inductively defined relations, as well as numerous special-purpose rules that users write in the course of their work, e.g. the conversion to differentiate algebraic expressions by proof described by Harrison [14].

The second reason harks back to the introduction above: the processes of proof discovery and proof checking can be quite different, not just for the community of human mathematicians, but for a theorem proving system like HOL. It is not necessary for each step of a higher-level algorithm or heuristic to be decomposed lock-step to a series of deductive steps. It's quite possible to use a completely separate checking phase.

[1] In May 1995, there was a public announcement that the 'Robbins conjecture' had been proved using the REVEAL theorem prover. This was subsequently traced to a bug in REVEAL, and the conjecture is still open.

[2] The name ML is derived from 'Meta Language'; following Tarski [29] and Carnap [6] it has become usual in formal logic to enforce a strict separation between the object logic and the metalogic (which is used to reason about the object logic).

3 Finding versus checking

From the perspective of computer theorem proving, there are interesting differences between proof finding and proof checking. For example, it is very straightforward and cheap to multiply together two large prime numbers on a computer, for example:

$$3490529510847650949147849619903898133417764638493387843990820577$$

and

$$32769132993266709549961988190834461413177642967992942539798288533$$

It's not even prohibitively hard to do it by hand. However going from the product (which we will call r to save space) back to the above factors seems, without prior knowledge of those factors, difficult. The security of certain cryptographic schemes depends on that (though perhaps not *only* on that). In fact the above factorization was set as a challenge ('RSA129'), and was eventually achieved by a cooperative effort of around a thousand users lavishing spare CPU cycles on the task.

There are two contrasts between the tasks of finding the factorization and checking it, both significant for the business of implementation in computer theorem provers. The first is in computational complexity. We've already discussed the *practical* complexity. From a theoretical point of view, multiplying n-digit numbers even in the most naive way requires only n^2 operations. But factorizing an n-digit number is not even known to be polynomial in n, and certainly seems likely to be worse that n^2.[3]The second contrast is one of implementation complexity. Writing an adequate multiprecision arithmetic routine is not a very challenging programming exercise. But present-day factorization methods are rather complex — it's a hot research topic — and often rely for their justification on fairly highbrow mathematics.

The crucial point is that even under the LCF strictures, a result can be arrived at it any way whatsoever provided that it is *checked* by a rigorous reduction to primitive inferences.

In general, we want the search stage to produce some kind of 'certificate' that allows the result to be arrived at by proof with acceptable efficiency. In the case of the above factorization, the certificate was simply the factors. Often the certificate can be construed simply as the 'answer' and the checking process a confirmation of it. But as we shall see, other certificates are possible.

4 Relationship to NP problems

The classic definition of the complexity class NP is that it is the class of problems solvable in polynomial time by a nondeterministic Turing machine (one which can explore multiple possibilities in parallel, e.g. by replicating itself). However, many complexity theory texts give another definition: it is the class of problems whose solutions may be *checked* in polynomial time on a *deterministic* Turing machine. Of course when they are framed as decision problems, as they usually are, there is no 'answer' to the problems beyond yes/no; checking that is no different from finding it. But in general there exists for each problem a key piece of data called a *certificate* that can be used in polynomial time to confirm the result. Often this *is* the 'answer' to the problem if the problem is rephrased as 'find an x such that $P[x]$' rather than 'is there an x such that $P[x]$?'. But in general, the certificate can be some other piece of data. Indeed, to prove that the two characterizations of NP are equivalent, one uses the fact that the execution trace of a nondeterministic Turing machine can be used as the certificate; checking this can be done simply by interpreting it. The other way is easy: the NDTM can explore all possibilities, running the checking procedure on each.

The equivalence of the two definition is strongly reminiscent of Kleene's normal form theorem in recursion function theory. That says that unbounded search need only be done in one place; here we say that nondeterminism can be concentrated in one place. Indeed, there is a similar *logical* (or 'descriptive') version of both results: Kleene's theorem states that any recursive predicate can be expressed as a first order existential statement with primitive recursive body (Σ^0_1), while Fagin [9] proves that NP problems are precisely those expressible as existential statements in second order logic over finite structures (Σ^1_1).

The close similarity with our wish for efficient proof checking should now be clear. We are interested in cases where a certificate can be produced by an algorithm, and this easily checked. Our version of the idea 'easily checkable' is less pure and more practical, since in the assessment of what can be checked 'easily' we include a number of somewhat arbitrary factors such as the nature of the formal system at issue and the mathematical and programming difficulty underlying the checking procedure. (For example, even if factoring numbers turns out to be in P, it will almost certainly be dramatically harder than multiplying the factors out again.) But the analogy is still strikingly close.

The complementary problems to the NP ones are the co-NP ones. Here it is *negative* answers which can be accompanied by a certificate allowing easy checking. A good example of a co-NP problem is tautology checking, the dual of the NP-complete problem of Boolean satisfiability. Boolean satisfiability admits an easily checked certificate, viz, a satisfying valuation, but no such certificate exists for tautologies, unless $P = NP$. We may expect, therefore, that our analogs of co-NP complete problems will be harder to support with an efficient checking process.

A problem that looks intuitively complementary to the problem of factorization is primality testing. However as shown by Pratt [26], short certificates are possible and so the problem is not only in co-NP but also in NP.[4]An especially strong form of this result is due to Pomerance [25], who shows that every prime has an $O(log\ p)$ certificate, or more precisely that 'for every prime p there is a proof that it is prime which requires for its verification $(\frac{5}{2} + o(1))log_2 p$ multiplications mod p'. Of course whether useful primality testing algorithms can naturally produce such certificates is still an open question. Elbers [8] has been exploring just such an idea, using then LEGO prover to check Pratt's prime certificates.

5 What must be internalized?

We have placed in opposition two extreme ways of implementing an algorithm as an LCF derived rule: to implement it entirely inside the logic, justified by formal inference at each stage, or to perform an algorithm without any regard to logic, yet provide a separate check afterwards. However, there are intermediate possibilities, depending on what is required.

For example, consider the use of Knuth-Bendix completion to derive consequences from a set of algebraic laws. Slind [27] has implemented this procedure in HOL, where at each stage the new equations are derived by inference. However the fact that any particular rewrite system resulting is canonical is *not* proved inside the

[3] The problem is known to be in P if the Extended Riemann Hypothesis holds. Also, it's generally not as hard to prove compositeness nonconstructively, e.g. it's pretty quick to check that $2^{r-1} \not\equiv 1 \pmod{r}$, so by Fermat's little theorem, r is composite.

[4] As already remarked, it's probably in P.

logic. This could be done either in an ad-hoc way for the particular system concerned, or could be done by internalizing the algorithm and proving its correctness. Either would require much more work, and cost much more in efficiency. And if all we want is to prove *positive* consequences of the rewrites, this offers no benefits. On the other hand to prove *negative* results, e.g. that a group exists which does not satisfy a certain equation, then this kind of internalization would be necessary. Such a theme often appears in HOL, where the steps in an algorithm may all be justified by inference, but the overall reasoning justifying its usefulness, completeness, efficiency or whatever are completely external (one might say informal).

6 Examples

It would be a very interesting exercise to make a systematic catalogue of algorithms used in all kinds of symbolic manipulations, and analyze whether each one admits an interesting separation of search from checking. If checkability comes to be considered important, then this could provide a fundamentally different criterion for choosing one algorithm over another. We will not attempt to compile such an ambitious list, but we will note some good examples. In particular, these include a large number of algorithms commonly associated with computer algebra systems. This has been exploited, for example, by Harrison and Théry [16] who physically link the HOL prover and the Maple computer algebra system, performing search and checking in completely different systems. Processes admitting a (relatively) easy checking process include:

- Factorizing polynomials (or numbers)
- Finding GCDs of polynomials (or numbers)
- Solving equations (algebraic, simultaneous, differential, . . .)
- Finding antiderivatives
- Finding closed forms for summations

In most cases the certificate is simply the answer. An exception is the GCD, where a slightly more elaborate certificate is better for our purposes. If we ask Maple to find the GCD of $x^2 - 1$ and $x^5 + 1$ using its `gcd` function, for example, it responds with $x + 1$. How can this result be checked? Well, it's certainly straightforward to check that this is *a* common divisor. If we don't want to code polynomial division ourselves in HOL, we can call Maple's `divide` function, and then simply verify the quotient as above. But how can we prove that $x + 1$ is a *greatest* common divisor[5]? At first sight, there is no easy way, short of replicating something like the Euclidean algorithm inside the logic (though that isn't a really difficult prospect).

However, a variant of Maple's GCD algorithm, called `gcdex` will, given polynomials p and q, produce not just the GCD d, but also two other polynomials r and s such that $d = pr + qs$. Indeed, the coefficients in this sort of Bezout identity follow easily from the Euclidean GCD algorithm. For example, applied to $x^2 - 1$ and $x^5 + 1$ we get the following equation:

$$(-x^3 - x)(x^2 - 1) + 1(x^5 + 1) = x + 1$$

This again can be checked easily, and from that, the fact that $x + 1$ is the *greatest* common divisor follows by an easily proved theorem, since obviously any common factor of $x^2 - 1$ and $x^5 + 1$ must, by the above equation, divide $x + 1$ too. So here, given a certificate slightly more elaborate than simply the answer, easy and efficient checking is possible.

Outside the realm of computer algebra there are other situations where checking is much simpler that finding. One obvious example is boolean satisfiability, which is useful for finding bugs in combinational logic circuits and the like. Given a satisfying valuation, checking is easy and mechanical. Although the converse problem of tautology checking is co-NP complete, certain algorithms may still admit relatively easy checking, when they succeed. The present author is currently investigating checking for a patented tautology checking algorithm due to Stålmarck [28], with promising results so far.

More generally, the problem of automated proof search in (usually first order) logic usually admits a very easy checking process — there are several automated tools in the HOL system based on this idea, e.g. see the work of Kumar et al. [18]. The point is that finding the proof may involve trawling through thousands, even millions, of formulas, but the eventual proof found is usually short. In some cases, as in Bundy's "proof plans", the proof search may involve complicated heuristics and AI techniques, but no such exotica is required for checking a proof. Actually the report by Bundy et al. [5] does not support the conclusion that checking is computationally quicker than finding, though it is much easier to implement. In fact they report that 'it is an order of magnitude less expensive to find a plan that to execute it', though this may be due in part to a badly implemented inference engine and the relatively limited problem domain for the planner (inductive proofs).

7 Related Work

Our general theme of checkability has been stressed by a number of researchers, notably Blum [2], who also mentions the GCD example. He suggests that in many situations, checking results may be more practical and effective than verifying code. This argument is related to, in some sense a generalization of, arguments by Harrison [15] in favour of the LCF approach to theorem proving rather than so-called 'reflection'. Mehlhorn et al. [21] describe the addition of verification to routines in the LEDA library of C++ routines for computational geometry (e.g. finding convex hulls and Voronoi diagrams). Our interest is a little different in that it involves checking according to a formal deductive calculus. However it seems that many of the same issues arise. For instance, they remark that 'a convex hull program that delivers a triangulation of the hull is much easier to check than a program that only returns the hull polytope', which parallels our example of a certificate for the GCD consisting of more than just the answer.

ACKNOWLEDGEMENTS

The issues discussed in this paper were stimulated by discussions with several people over the last few years; in particular Konrad Slind, Laurent Théry, Norm Megill and Gunnar Stålmarck. While this paper was written I was employed at the Programming Methodology Group at Åbo Akademi University and generously funded by the European Commission under the Human Capital and Mobility Programme. Thanks to Jim Grundy for constructive comments on a draft at extremely short notice.

REFERENCES

[1] Argonne National Laboratories, 'A summary of new results in mathematics obtained with Argonne's automated deduction software'. Unpublished; available on the Web as

[5] The use of 'greatest' is a misnomer: in a general ring we say that a is a GCD of b and c iff it is a common divisor, and any other common divisor of b and c divides a. For example, both 2 and -2 are GCDs of 8 and 10 over $\mathbb{Z}$.

http://www.mcs.anl.gov/home/mccune/ar/new_results/, 1995.

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

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

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

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

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

[7] N. G. de Bruijn, 'The mathematical language AUTOMATH, its usage and some of its extensions', in *Symposium on Automatic Demonstration*, eds., M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, volume 125 of *Lecture Notes in Mathematics*, pp. 29–61. Springer-Verlag, (1970).

[8] Hugo Elbers, 'Construction of short formal proofs of primality'. Preprint, 1996.

[9] Ronald Fagin, 'Generalized first-order spectra and polynomial-time recognizable sets', in *Complexity of Computation*, ed., R. M. Karp, volume 7 of *SIAM-AMS Proceedings*, pp. 43–73. American Mathematical Society, (1974).

[10] Michael J. C. Gordon, 'Representing a logic in the LCF metalanguage', in *Tools and notions for program construction: an advanced course*, ed., D. Néel, pp. 163–185. Cambridge University Press, (1982).

[11] Michael J. C. Gordon and Thomas F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press, 1993.

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

[13] John Harrison, 'A HOL decision procedure for elementary real algebra', In Joyce and Seger [17], pp. 426–436.

[14] John Harrison, 'Constructing the real numbers in HOL', *Formal Methods in System Design*, **5**, 35–59, (1994).

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

[16] John Harrison and Laurent Théry, 'Extending the HOL theorem prover with a computer algebra system to reason about the reals', In Joyce and Seger [17], pp. 174–184.

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

[18] Ramaya Kumar, Thomas Kropf, and Klaus Schneider, 'Integrating a first-order automatic prover in the HOL environment', in *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, eds., Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, pp. 170–176, University of California at Davis, Davis CA, USA, (1991). IEEE Computer Society Press.

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

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

[21] K. Mehlhorn et al., 'Checking geometric programs or verification of geometric structures'. Unpublished; to appear in the proceedings of the 12th Annual ACM Symposium on Computational Geometry, SCG'96. Available on the Web as http://www.mpi-sb.mpg.de/LEDA/articles/programc.dvi.Z, 1996.

[22] Thomas F. Melham, 'Automating recursive type definitions in higher order logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, eds., Graham Birtwistle and P. A. Subrahmanyam, pp. 341–386. Springer-Verlag, (1989).

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

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

[25] Carl Pomerance, 'Very short primality proofs', *Mathematics of Computation*, **48**, 315–322, (1987).

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

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

[28] Gunnar Stålmarck, 'System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula'. United States Patent number 5,276,897; see also Swedish Patent 467 076, 1994.

[29] Alfred Tarski, 'Der Wahrheitsbegriff in den formalisierten Sprachen', *Studia Philosophica*, **1**, 261–405, (1936). English translation, 'The Concept of Truth in Formalized Languages', in [30], pp. 152–278.

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

[31] Andrzej Trybulec, 'The Mizar-QC/6000 logic information language', *ALLC Bulletin (Association for Literary and Linguistic Computing)*, **6**, 136–140, (1978).

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