Metatheory and Reflection in Theorem Proving: A Survey and Critique

John Harrison
University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
England
jrh@cl.cam.ac.uk

15th February 1995

Abstract

One way to ensure correctness of the inference performed by computer theorem provers is to force all proofs to be done step by step in a simple, more or less traditional, deductive system. Using techniques pioneered in Edinburgh LCF, this can be made palatable. However, some believe such an approach will never be efficient enough for large, complex proofs. One alternative, commonly called *reflection*, is to analyze proofs using a second layer of logic, a *metalogic*, and so justify abbreviating or simplifying proofs, making the kinds of shortcuts humans often do or appealing to specialized decision algorithms. In this paper we contrast the fully-expansive LCF approach with the use of reflection. We put forward arguments to suggest that the inadequacy of the LCF approach has not been adequately demonstrated, and neither has the practical utility of reflection (notwithstanding its undoubted intellectual interest). The LCF system with which we are most concerned is the HOL proof assistant.

The plan of the paper is as follows. We examine ways of providing user extensibility for theorem provers, which naturally places the LCF and reflective approaches in opposition. A detailed introduction to LCF is provided, emphasizing ways in which it can be made efficient. Next, we present a short introduction to metatheory and its usefulness, and, starting from Gödel's proofs and Feferman's transfinite progressions of theories, look at logical 'reflection principles'. We show how to introduce computational 'reflection principles' which do not extend the power of the logic, but may make deductions in it more efficient, and speculate about their practical usefulness. Applications or proposed applications of computational reflection in theorem proving are surveyed, following which we draw some conclusions. In an appendix, we attempt to clarify a couple of other notions of 'reflection' often encountered in the literature.

The paper questions the too-easy acceptance of reflection principles as a practical necessity. However I hope it also serves as an adequate introduction to the concepts involved in reflection and a survey of relevant work. To this end, a rather extensive bibliography is provided.

1 Extending theorem provers

Computer theorem provers typically implement a certain repertoire of inference mechanisms. For example, they may solve tautologies, do first order reasoning by resolution, perform induction, make simplifying rewrites and allow the user to invoke other more delicate logical manipulations.

Certain systems (e.g. fast tautology checkers) are only useful for applications in a quite restricted field. Others, by virtue of the generality of the logic they implement and the theorem proving procedures they provide, are of wider applicability, whether or not such wider applicability was an original design goal. HOL was originally intended mainly for the verification of hardware, but has subsequently been applied to software and protocol verification, work in other embedded formalisms like temporal logic and CCS, and even in pure mathematics.

When a theorem prover is of such general applicability, it is difficult for its supplier to provide a basic repertoire of theorem proving facilities which is adequate for all purposes. The most desirable policy is to make the theorem prover extensible, i.e. provide a facility for augmenting this basic repertoire. At its simplest, this might consist of a macro language to automate certain common repeated patterns of inference. However this does not address the question of implementing radically different proof procedures from those already included. In particular a macro language is likely to have limited facilities for direct construction of terms and formulas, since one has to demarcate the valid inferences somehow — one can't allow arbitrary formulas to be nominated as 'theorems'. So what are the options?

1. If some new inference rule proves useful, simply augment the theorem prover's basic primitives to include it.

2. Allow a full programming language to manipulate the basic rules, so that users may write arbitrarily complex inference rules which ultimately decompose to these primitives.

3. Incorporate a principle of reflection, so that the user can verify within the existing theorem proving infrastructure that the code implementing a new rule is correct, and add that code to the system.

We should say that some theorem provers adopt a mixture of these policies. For example Nuprl has 'sacred' and 'profane' forms. In its sacred incarnation, all inference is done by decomposing to the primitive rules of Martin-Löf type theory. However in practical applications, various additional facilities such as binary decision diagrams and arithmetic decision procedures have sometimes been grafted onto it. Furthermore, the third option has been investigated by Nuprl researchers over the course of many years; we shall discuss this further below.

If the first of the three options is taken, then considerable care should be exercised to ensure that the proposed extension is sound. If users are allowed to tailor the system themselves, what are we to make of claims that a theorem has been proved using that system? If a user's bespoke variant is unsound, they are no longer using the original theorem prover at all, but some incorrect mutation of it. The supplier will be unwilling to concede that the proof has been performed to a satisfactory standard. It can hardly be held up as meeting an objective standard of correctness. In a small user community (e.g. within a single research establishment) consultation with peers may help to achieve a considered view of how sound an enhancement is. However in a wider community with people at work on personal projects, this is not realistic.

In practice then, for sociological reasons, major changes must be implemented by the supplier. This may be enforced by making the system 'closed' (no source code available, for example). However the suppliers still need to exercise careful judgement over which new facilities to add. It's very tempting to add just one more 'obviously sound' extension, and reason by induction that the addition of n such 'obviously sound' extensions will not compromise the system. Experience shows that such 'obviously sound' extensions are frequently not sound at all. Perhaps the suppliers will attempt to verify, informally or formally, that the code with which they are augmenting the system is correct. If this is done formally, then the approach looks similar to the third (reflection), but reflection has the distinguishing feature that the proof is conducted *in the existing theorem prover*.

Even with careful thought, this does not seem a principled way of making a reliable system. Nevertheless we do not mean to suggest that it is an intellectually disreputable policy. On the contrary, if the purpose of theorem proving is to highlight flaws in informal reasoning, it may be most efficient to add lots of facilities in an ad hoc way in order to get quickly to the interesting and error-prone parts of the reasoning. Certainly, a final assertion of correctness means less than it otherwise might, but the bugs that are found may emerge more quickly. For example, interesting work on a 'hybrid' system including HOL and a fast symbolic model checker is reported by Joyce and Seger (1993b). This allows one to tackle leading-edge problems in hardware verification while still taking advantage of the higher level of abstraction that HOL permits.

Such a policy has been argued for by Rushby (1991), and one can point to similar ideas elsewhere. For example, some work on floating point verification at ORA reported by Hoover and McCullough (1992) aims to prove correctness asymptotically as the precision of the arithmetic tends to infinity. Though this is useless at giving true error bounds, experience shows that it does quickly highlight many bugs in software. Another example: computer algebra systems are widely used (much more so than theorem provers) despite sometimes giving wrong answers.

2 The LCF approach to theorem proving

The Edinburgh LCF system, described by Gordon, Milner, and Wadsworth (1979), was developed by Milner and his research assistants in the mid 70s, in response to dissatisfaction both with highly automatic provers and with low-level proof checkers. A prover of the latter kind was developed as Stanford LCF by Milner (1972) and provided much of the motivation. Edinburgh LCF was ported from Stanford LISP to Franz LISP by Huet and formed the basis for the French 'Formel' research project. Paulson (1987) reengineered and improved this version, resulting in Cambridge LCF. The original system implemented a version of Scott's Logic of Computable Functions (hence the name LCF), but as emphasized by Gordon (1982), the LCF approach is applicable to any logic.

In LCF-like systems, the ML programming language is used to define data types representing logical entities such as types, terms and theorems. The name ML is derived from 'Meta Language'; effectively LCF systems *do* have a kind of metalogic, but an explicitly algorithmic one, where the only way to demonstrate that something is provable is to prove it. Since its LCF-related origin, the ML language has achieved a life of its own. It is a higher-order functional programming language with polymorphic types together with a typesafe exception mechanism and some imperative features. A reasonably formal semantics of the Standard ML core language has been published — see Milner, Tofte, and Harper (1990) and Milner and Tofte (1991).

A number of ML functions are provided which produce theorems; these implement primitive inference rules of the logic. For HOL there are 8 such rules; Nuprl, which implements Martin-Löf's constructive type theory, requires well over a hundred. For HOL, these primitive inference rules have been proved sound via a set-theoretic semantics devised by Andy Pitts and published in Gordon and Melham (1993). Pottinger (1992) has also proved that they are complete with respect to Henkin's general models[1]. 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. Simple proof checkers are not to be scorned, since they have been used to check substantial parts of mathematics. A pioneering example was the work by Jutting (1977), formalizing the famous book by Landau (1966) in the AUTOMATH system of de Bruijn (1980). More recently, a large and disparate body of mathematics has been checked in the Mizar system described by Rudnicki (1992), and there is even a journal 'Formalized Mathematics' devoted to Mizar formalizations[2].

Nevertheless, it is attractive to be able to direct proofs at a higher level, rather than perform proofs of trivial tautologies or facts of arithmetic explicitly. This is even more important in verification applications than in pure mathematics, since the proofs, though shallower, tend to be much more involved and intricate because there are few abstraction mechanisms to hide the layers of complexity; though see Melham (1993).

This can be done in LCF provers: using ML programming, 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 and flexibility of a more sophisticated prover, provided someone is prepared to put in the work required to provide useful derived inference rules.

For the sake of completeness, we should add that 'morally', a true LCF implementation has rather simple primitive rules. Primitives in HOL range in complexity from reflexivity of equality to simultaneous parallel substitution. They do not include rewriting or arithmetic decision procedures. An alternative policy, exemplified by the PVS system described by Owre, Rushby, and Shankar (1992), is to make the primitives powerful. This means that less programming is required to build up a suitable set of high-level operations, and that these high-level operations may be more efficient. However it has the defect that it's much harder to be confident about the correctness of such complex primitives, and in any case which primitives are useful can depend on context (for example, users of an embedded formalism

[1]Actually this was without considering polymorphism, but from a proof-theoretic perspective, HOL's polymorphism is obviously a conservative extension since any type instantiations can be floated back up the proof tree.

[2]For more information about this journal, which is surprisingly inexpensive, contact: Fondation Philippe le Hodey, MIZAR Users Group, Av. F. Roosevelt 134 (Bte7), 1050 Brussels, Belgium, fax +32(2)6408968.

may require unusual proof procedures). On the other hand, one can argue that HOL goes too far in the direction of parsimony, e.g. in insisting that arithmetic be done in the logic. Indeed, ICL ProofPower[3] relaxes this restriction and takes numeral addition as a primitive rule.

Readers should be aware that the word 'tactic' is widely used in the theorem proving literature to refer to what we have been discussing, i.e. a compound proof step which ultimately decomposes to some given primitives. In HOL, as in the original LCF system, the word 'tactic' is reserved for cases where the high-level proof step works in a backward (goal-directed) manner. To avoid ambiguity and irrelevant distinctions, we refer to 'derived rules', regardless of whether the steps are forward or backward. However in reading some of the quotes below, the more general use of the word 'tactic' should be borne in mind.

The advantages of LCF

One obvious advantage of the LCF approach is that the user can feel a good degree of confidence that a purported theorem really is a theorem. The critical code is confined to that implementing the primitive rules (and any support functions they use).

However it should be admitted that there is also a dependence on the correctness of the ML implementation, in particular the correctness of its type system. If for this or other reasons one finds the assertion of trustworthiness unconvincing, it is quite easy to change the system to record proofs. This has actually been done in HOL by Wong (1993), so that each primitive inference is logged. (Indeed some LCF-style provers such as Nuprl and Coq already store proof trees or lambda-term witnesses.) Such a proof may then be checked by a simpler external program, written in any chosen programming language. For an overview of various pieces of research in HOL connected with these ideas, see Gordon, Hale, Herbert, von Wright, and Wong (1994). In particular, it may be feasible to prove a simple proof checker correct. An important part of such a project is to analyze carefully what constitutes a HOL proof, and to this end the notion of HOL proof has itself been formalized in HOL by von Wright (1994). In the process a few errors and obscurities in the logical core were uncovered.

Whether or not one personally considers it worthwhile, such proof checking is recommended by certain procurement standards for safety critical software, e.g. MOD (1991):

> 32.3.1 In practice, it is very unlikely that Formal Proofs of any size will be created by hand. Instead, they will be developed using theorem proving assistants, which are interactive programs that carry out symbol manipulation under the guidance of a human operator. But theorem proving assistants are large programs whose correctness cannot readily be demonstrated by Formal Proof. It is, however, possible to remove the reliance on the correctness of the theorem proving assistant from the case for correctness of an application by arranging that a version of the final proof (omitting all history of its construction) is passed from the theorem proving assistant to a proof checker. For reasonable languages, such a proof checker could be a very simple program (perhaps ten pages in a functional programming language) that could be developed to the highest level of assurance.

Naturally, it is possible for non-LCF systems to provide a low-level proof script — for example see Kromodimoeljo and Pase (1994) for a discussion of adding proof

[3] ProofPower is a trademark of International Computers Limited.

logging to the NEVER system. However in making a theorem prover capable of producing primitive inferences, one is effectively writing a second, LCF-style mode for an existing theorem prover. Furthermore, if the primitive inferences are complex, their correctness can still be questioned, so perhaps even LCF's decomposition to *simple* primitives will need to be emulated. It seems much more elegant to adopt the LCF approach from the start.

The LCF approach offers great flexibility to ordinary users, who may extend the system with customized derived rules. Slind (1991) has remarked:

> From a certain point of view, the LCF approach to theorem proving is Socialist and hence deserves its own *Manifesto*: 'The user controls the means of (theorem) production'.

The rules added by a user do not need to be verified; if there is an error, then the proof procedure may fail, but because of the ML type discipline, will never produce an invalid 'theorem'. Users do not need to provide theoretical justification for adding a new derived rule, still less prove the correctness of the code implementing it. They simply have to design it to decompose to primitive inferences. In simple cases this is straightforward; in more complex cases it is not, but arguably it is normally easier than performing correctness proofs of the code.

The inherent inefficiency of LCF?

These advantages are undeniable; however it appears that there is a heavy price to pay: every derived rule and proof procedure must be forced into the straitjacket of decomposing to primitive rules. Apart from seeming rather unnatural in some instances, it is hardly likely to be as efficient as a hand-coded proof procedure. This of course is not necessarily a problem; the key question is whether it is a serious constraint in practice.

Indeed, the view firmly embedded in theorem proving folklore is that it is a serious constraint, making the LCF approach ultimately untenable for practical examples. LCF has certainly been influential, but it is subsidiary features of it, such as the use of tactics to implement backward proof, or the general stress on interactive rather than fully automatic proving, which have been most widely imitated. Systems which adhere to the pure LCF philosophy are rare. In fact, of those widely used in verification examples, HOL is arguably the only system which does. Characteristic criticisms of LCF are these from Davis and Schwartz (1979):

> ...the Edinburgh LCF system ...employs the device of "tacticals" to obtain a modest degree of extensibility. However an LCF tactical is limited to a fixed combination of existing rules of inference. This has the virtue that no correctness proof in our sense is needed but also have [sic] the obvious limitation that no really new inference rule can be adjoined.

from Armando, Cimatti, and Viganò (1993):

> The major drawback of this approach is that each proof procedure (even the most sophisticated ones) must ultimately invoke the basic inference rules. In many cases this turns out to be both unnatural and ineffective. Indeed, most of the decision procedures for decidable theories (e.g. the truth table method for propositional calculus) can be hardly rephrased as proof strategies based on the inference rules of the corresponding calculus. Moreover, the performance of the translated proof strategies is much less effective than a direct implementation of the original procedure.

from Reif and Schönegge (1994):

> In our opinion the currently available validation mechanisms are not powerful enough for large applications, for example in software verification. ... For example, in HOL (and most of the other systems above) extensions are correct per construction but the application of large tactics is inefficient. For practical applications this is a bad compromise.

and from Basin (1994):

> ..., I believe that tactics alone are insufficient to provide a proper level of reasoning for interactive theorem proving and additional mechanisms for metatheoretic extensibility are required. For example, we should be able to extend a prover with new kinds of simplifiers and decision procedures that are not hampered by the need to produce a proof with primitive rules.

When encountering such trenchant criticism one naturally expects to find theoretical justification or the results of case studies to lend support to the assertions. However here one searches in vain for any such thing. Indeed, in the same conference Arthan (1994) writes:

> I would feel a lot more confident in predicting the way theorem prover architecture should go, if more was known about the complexity issues both theoretical and practical that we face. ... Are there natural theorem proving problems with a reasonably tractable decision procedure but with no tractable means of finding proofs from primitive inferences?

It appears that nobody has ever answered this last question convincingly. But it is of central importance. Such is the power and intellectual simplicity of the LCF approach that it seems foolish to enter more nebulous and complicated areas like reflection unless there is some good reason for supposing the LCF approach practically insufficient[4]. It may be too much to expect a rigorous theoretical answer to Arthan's question, since complexity theory often reveals the difficulty in proving apparently simple facts; witness the status of the $P = NP$ problem. But we should be able to say something useful.

The sufficiency of LCF

We will maintain that the inefficiency of the LCF approach has often been overplayed. Our remarks will apply to HOL, and to some extent the validity of our remarks will depend on the logic's being higher order, as well as other details perhaps. First there are two obvious practical counterarguments:

- HOL is currently a widely used theorem proving system. Experience shows that in the overwhelming majority of cases, the difficulty in proving a theorem in HOL is not due to inefficiency, and few users are constrained by such inefficiency. Rather, the difficulty is in providing a proof at all; the key time factor is user thinking time! This means that quite substantial changes (for better or worse) in the efficiency of the basic proof procedures may have minimal impact on the productivity of users. One of the lessons of LCF is that proofs can be guided at quite a low level without its becoming too

[4]Apart of course from sheer intellectual curiosity. This is a laudable motivation but it should not be confused with the dictates of practice. To parody Dijkstra, it is important to distinguish the research fields of 'theorem provers' and 'theorem proving'.

tedious, provided a few well-chosen derived rules are provided. Only when running derived decision procedures on large examples does efficiency become important. This argument is less strong in the case of automatic provers, though, and might have less force for interactive provers too if users come to rely on more sophisticated derived rules.

- Computers are getting faster all the time, and it may be that what today is an efficiency problem ceases to be one tomorrow. The author has heard anecdotal evidence of rewrites taking half an hour in very early versions of LCF. Today, for a rewrite in HOL to take even one second is unusual. Of course, if the size of typical proofs expanded in time with the development of machines, this argument would no longer hold. But such a correlation seems improbable. In its early days, Unix ran a little more slowly because it was written in C rather than assembler. Even at the time the slowdown was modest considering the organizational advantages of a higher level language. Nowadays nobody considers it (and almost nobody writes assembler).

There are several more specific arguments. These are contentions which seem hard to justify by theoretical musings; rather they need case studies. However we believe current experience provides some strong support for them. For a detailed examination of sources of inefficiency in HOL, see Boulton (1993).

1. Most other proof procedures may be implemented in a fairly natural way using primitive inferences. Instead of plugging terms together in an ad hoc manner, such transformations may be justified at each stage by matching against a suitable theorem, requiring only a few primitive inferences each time to instantiate the theorem and perform, say, a Modus Ponens step.

2. In most cases of sophisticated inference rules, inference is (or may be construed as) only a small part. Search often dominates, which can be done without requiring any primitive inferences; this need not even be written in ML or performed by HOL itself.

3. Much of the inefficiency of derived rules in HOL arises from poor (or excessively simpleminded) programming. Using more careful or sophisticated programming, many of the efficiency problems disappear.

4. There are many implementation improvements that can be applied to ML systems. Questions of primitive inference aside, ML is often an order of magnitude slower than C in comparable applications.

Before looking at these matters in more details, let us give an example to illustrate some of the above assertions. HOL's rewriting rules and tactics take a set of equational theorems and repeatedly replace instances of the left hand sides by corresponding instances of the right hand sides in another term. The subterms to be rewritten and the number of repeated rewrites may be precisely controlled, but by default the term is searched for matching subterms in top-down order, repeatedly until no more are possible.

The initial search for matching subterms can be implemented quite independently of the primitive inferences required to perform the rewrite. In HOL, term nets [5] are used to achieve fast lookup, and only the apparent matches thrown out by this matcher are actually tried. This part appears to dominate rewriting's efficiency, especially when there are a large number of rewriting theorems. Perhaps more sophisticated algorithms such as those given by Hoffmann and O'Donnell (1982) could

[5] A well-known indexing technique for tree structures, originally added to Cambridge LCF by Paulson.

be tried instead; the point is that the matching is unaffected by any need to perform inference. The rewriting of a subterm is performed using a few instantiations, and the buildup of the whole term done by iterating congruence rules for equality — not much slower than simply plugging the terms together in an ad hoc way. In fact Boulton (1993) minimized the rebuilding of unchanged subterms by exploiting failure, an optimization which is orthogonal to the question of performing inference and indeed should perhaps be applied to the implementation of term substitution in HOL's core.

Using theorems to justify inference

The contention above was that most patterns of inference carried out ad hoc may be implemented in terms of primitive inference in a fairly straightforward fashion and with only a moderate slowdown. To some extent, this is particularly true of HOL, since higher order logic provides the power to encode quite sophisticated structures in theorems. This idea has long been used by HOL experts; for an early example see Melham (1989). In recent work, Harrison (1993) defines a generic representation of univariate polynomials using lists:

$$\begin{array}{l} (poly \;\; [] \;\; x = 0) \wedge \\ (poly \;\; (CONS\; h\; t) \;\; x = h + x \; \times \; (poly \;\; t \;\; x) \end{array}$$

Now it is possible to prove (by list induction), for example that every polynomial is differentiable. Then for any specific instance it is necessary only to instantiate this 'proforma' theorem and perform some rewrites to unwind the recursive definition. It is not necessary to repeat the whole proof every time. Using similar proformas quite sophisticated inferences are encoded, for example:

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

meaning that all of a finite set (list) of polynomials are 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.

In fact, examples like this awaken one to the fact that HOL's logic incorporates a simple functional programming language, and many operations which could be implemented outside can alternatively be internalized in the logic and executed by primitive inference. Of course this is quite a lot slower, but it seems unlikely to mark the difference between tractability and intractability except right at the boundary of what is possible using special proof procedures. One would expect the slowdown to be linear.

This does however presuppose an important condition. If a transformation from $\vdash E[t]$ to $\vdash E'[t]$ is to be justified by appeal to a pre-proved theorem $\vdash \forall x.\; E[x] \Rightarrow E'[x]$, the Modus Ponens step must check that $E[t]$ and $E[t]$ are equal. In a good ML implementation, this will be an efficient operation since internally the two t's are actually the same *as pointers* (i.e. are `EQ` in LISP parlance). Thus a traversal of the two trees will terminate once t is reached, and the efficiency only depends on the structural complexity of $E[x]$ with respect to x (as would any applicative implementation). If a full traversal were required each time, it would be a severe efficiency bottleneck. Unfortunately, current versions of HOL use α-equivalence, not equality, as the condition in most derived rules, and the α-equivalence test does not succeeds quickly when terms are `EQ` (though HOL88 performs a single `EQ` test, it does not repeat this test as it traverses subterms). It is our opinion that this should be changed, e.g. by performing an equality test first. (Standard ML does not provide something like LISP's `EQ`, so the more obvious optimization of putting an `EQ` test inside the α-equivalence function is not possible.)

With this proviso, it is close to being plausible that anything implementable in ML using purely functional programming can also be implemented inside the logic with only a moderate constant factor slowdown. Cases where this is not true are likely to be where the type system constrains the generality of a proforma theorem; for example one might want to state a theorem for arbitrary iterations of the function space constructor. (In more general type systems like Nuprl's, even this may be unproblematical.)

For example, it is useful to be able to justify inductive definitions using the Tarski fixpoint theorem, but to interpret 'monotonicity' not just for unary relations (sets) but relations of arbitrary arities; i.e.

$$(\forall x_1 \ldots x_n.\, R\, x_1 \ldots x_n \Rightarrow S\, x_1 \ldots x_n) \Rightarrow (\forall x_1 \ldots x_n.\, E[R]\, x_1 \ldots x_n \Rightarrow E[S]\, x_1 \ldots x_n)$$

This particular case can be dealt with in various ways while maintaining an instantiation linear in the number of variables (indeed repeating the proof every time satisfies the constraint and does not use so many primitive inferences each time).

Separating search from inference

A lot of theorem proving procedures involve substantial amounts of search, but once the search is complete, it may be simple to produce a proof in terms of primitive inferences. For example, a resolution proof may involve searching through thousands, even millions, of clauses for a refutation. Once found, however, the path to a refutation is usually quite short, and provided the clauses and unifiers are recorded, can easily be transformed into a HOL proof using a little instantiation and propositional reasoning. Other tableau-based decision procedures for first order logic have actually been developed in HOL in this way; see work by Kumar, Kropf, and Schneider (1991) and Schneider, Kumar, and Kropf (1992). Similar remarks apply to many arithmetic decision procedures, and some of these have actually been implemented in HOL as very useful practical tools by Boulton (1993).

Some other intersting procedures are those like the factorization of polynomials (or numbers!), the finding of antiderivatives and the solution of equations. These are difficult, computationally intensive or require sophisticated heuristics. Nevertheless, once a putative answer is found, it is a relatively straightforward matter to check its correctness — all that is required for a formal proof. Based on this observation, Harrison and Théry (1993) discuss using a link between HOL and the Maple computer algebra system. More general than using external oracles to produce checkable *answers* is the idea of delegating all the proof-finding to an external 'proof planner'. This idea has been explored by Bundy, van Harmeleu, Hesketh, and Smaill (1991).

Other optimizations

There are a number of other techniques which may make the LCF approach more efficient. For example, in many situations (Nuprl typechecking obligations are a good example) the same trivial theorem is proved over and over again. It may be that by cacheing (aka 'memoizing') previous theorems, a substantial gain in efficiency could be obtained. An example of this situation for arithmetic theorems is described by Boulton (1993).

A more general technique for optimizing LCF implementations has been proposed by Boulton (1993): lazy theorems. The idea is to delay the inference phase of rules until later, accumulating a function closure during 'inference' and only executing it when and if the final theorem is needed (as already happens in HOL *backward* proof — it is exactly how tactics work). This offers some potential for making interactive proof more efficient, while postponing costly inferences until later (even overnight perhaps). If that were the only gain, the idea could more simply be achieved by 'slow' and 'fast' modes, where no inference is done in the latter. However it may be that lazy theorems can help to manage the separation of search and inference automatically, allowing the programmer to employ a more free and easy style, using inference rules during exploratory search where convenient. Furthermore there may be special situations where inferences can be more efficiently decided on after examining the whole 'proof'; for an example of this phenomenon, see the work on BDDs in HOL by Harrison (1995).

Partial evaluation

There is a substantial research area of 'partial evaluation' which aims to optimize functional programs by precomputing parts of them. The idea is similar to the well-known idea of constant folding in compilers, but much more sophisticated. A very nice summary is the following, from Bjørner, Ershov, and Jones (1988):

> In the large, the goal of PE is to construct, when given a program and some form of restriction on its usage (e. g. knowledge of some but not all of its input parameter values), a more efficient new or "residual" program that is equivalent to the original program when used according to the restriction.

Some HOL derived rules have been manually optimized to perform computation before they see all their arguments. For example, the rewriting rules canonicalize the rewrites and set up the term net before they are applied to the term to be rewritten. There may be considerable potential for similar optimizations to be made automatically. Ideally, whole sequences of primitive inferences might be folded down to something much more efficient. Work on applying partial evaluation to HOL is being undertaken by Welinder (1994), and preliminary results are promising. We should point out that the use of imperative features, in particular proof recording, at the level of primitive inferences, is likely to all but destroy the potential optimizations.

Incidentally, papers by Danvy (1988) and Talcott and Weyhrauch (1988) explore some possible connections between partial evaluation and reflection.

Difficult cases

It seems, then, that the cases where a tractable proof by primitive inference is hard to find are likely to be algorithms making essential use of imperative features (arrays, shared data structures) and not allowing a cheap checking process. Binary Decision Diagrams as described by Bryant (1992) constitute just such an algorithm, and so present an interesting challenge. Harrison (1995) implements BDDs as a derived rule, encountering a slowdown of something like 40 times over a direct Standard ML implementation. This is a significant factor, but not outrageously large considering that BDDs were chosen precisely because they were a challenge. Furthermore the second of the palliatives discussed above applies to some extent, since very often a well chosen variable ordering makes a tremendous difference to the efficiency of a BDD-based tautology check. Deciding on a good variable ordering can be done without requiring primitive inference.

There are other special theorem proving algorithms which might be at least as hard to implement satisfactorily in an LCF-style system. For example the method invented by Wu (1978) for solving geometrical problems is remarkably powerful — see Chou (1988) for some impressive examples. This involves manipulation of polynomials of very high degree, so much so that complicated but asymptotically fast FFT-based algorithms for multiplication are a *sine qua non* for large examples. It remains to be seen whether a reasonably fast implementation could be done in the LCF style.

There is a somewhat different example where a proof by primitive inference is likely to be dramatically less efficient: arithmetic. (And more generally, any situation where we end up duplicating a hardware facility inside the logic, but this seems the only real example.) Rather than using the machine arithmetic we must perform inference inside the logic even to add a couple of integers. We can use a binary or decimal representation inside the logic, and consequently the speed of arithmetic operations on numbers of size n is likely to be $O(log_2(n))$ or $O(log_2(n)^2)$. From a purely theoretical point of view of course, machine arithmetic is no different when one allows arbitrary precision. However from a practical perspective, the constant factor difference in speed may be enormous. How serious this problem is depends on whether manipulation of large numbers occurs frequently in typical applications. Replicating in HOL the work described by Boyer and Yu (1992) could be quite difficult, though of course one might follow ProofPower in softening the LCF ideal slightly and using the bignums provided by (or written in) the implementation language to perform arithmetic in the logic.

Finally, it is worth noting that in cases where the terms involved become very large, there may arise hidden overheads; for example when instantiating a theorem with a very large assumption, a complete traversal of the assumption term is required to check that the instantiated variable is not free. It may even be that a decomposition to primitive inferences forces a retraversal many times, whereas metaknowledge might convince us that one check suffices. Sometimes this can be achieved, paradoxically, by making certain aspects of the theorem explicit in the logic, rather than relying on the way the primitive rules work, as discussed by Harrison (1995). It is not clear whether it represents a serious difficulty in any practical cases. An analogy can be drawn with the way some languages or compilers insert bounds checks into all array references. The programmer may know for a fact that they are unnecessary because of the detailed structure of the program. This is one of the reasons why C, which does not perform such checks (and in general cannot, because array dereferencing may be applied to arbitrary pointers), often outperforms superficially similar languages.

In conclusion, it is not clear that there are any useful practical proof procedures which are impossible to accommodate in the pure LCF approach. Indeed from a more theoretical (and unrealistic) perspective it has not been demonstrated that any proof procedure suffers more than a constant factor slowdown when implemented in the LCF style. Certainly it does not seem that many applications naturally demand such procedures. The most likely exceptions are fast model-checking algorithms like BDDs which are much used in hardware verification. It is not clear that software verification makes similar demands, and likely that pure mathematics does not — more justification of this last claim may be found below. In any case, if reflection is to represent a substantial practical advance, justifying its greater difficulty, verification of essentially imperative code is likely to be required.

3 Metatheory

Various antinomies have been discovered since antiquity, which were classified by Ramsey (1926) into 'logical' and 'epistemological' contradictions. The classic example of a logical paradox is Russell's[6] which arises by considering "the set of all sets which are not members of themselves", or the equivalent using predication instead of set membership. These paradoxes all seem to involve some kind of self-reference inside the logical system, and have been avoided by adding a notion of type, either explicitly as in Type Theory or implicitly as in hierarchical set theories [7] or those based on stratified formulas.

Typical examples of epistemological paradoxes are the Liar ("this statement is false") and Santa Claus[8] ("if this statement is true then Santa Claus exists"). These also involve a kind of self reference, but involving questions which go beyond mathematics — for example a *sentence* referring to itself. In order to avoid them it is usual in formal logic, following Tarski (1936) and Carnap (1937), to enforce a strict separation between the object logic and the metalogic (which is used to reason about the object logic).

In a formal logical system, it may sometimes happen that the most natural form of reasoning is metareasoning. For example, inferences in Hilbert-style proof systems are greatly eased by using the Deduction Theorem, which states that $A \vdash p \Rightarrow q$ if and only if $A \cup \{p\} \vdash q$.

Metatheorem or higher order theorem?

When one discusses a particular formal system, there is a straightforward distinction between an object-level theorem and a metatheorem. Sometimes a metatheorem may say essentially the same as an object level theorem. For example, the metalevel statement $\vdash_{object} \phi$ corresponds to the object level statement ϕ. In general it may be impossible to state an object level theorem with the same import as a metatheorem. Nevertheless it is crucial to note that *what can be stated at the object level depends critically on the nature of the object level formal system.* Consider the following breathtakingly categorical statements from Aiello and Weyhrauch (1980):

> ...if the reasoning system you are dealing with has no capability of explicitly representing metatheoretic knowledge, many of the statements in elementary math books cannot even be expressed. This might not be interesting if such meta-statements never appeared in practice. On the contrary they arise *very* often in mathematics books ...

and Aiello, Cecchi, and Sartini (1986):

> ...in reading a book on algebra one realizes that most of the stated lemmas and theorems are in fact metatheorems.

No instances of these putative metatheorems are cited. However Matthews (1994) is more explicit:

> Mathematics is not done with a proof development system in quite the same way as it is done in a textbook, even when the two look like one another. For instance in a book on algebra one might read:

[6] Zermelo arrived at the same paradox at much the same time but chose to make little of it.

[7] Zermelo's original system was based on Cantor's 'limitation of size' doctrine, which held that collections were paradoxical simply because they were 'too big'. However the subsequent addition of the Axiom of Foundation amounts to an admixture of type theory.

[8] Gödel's proof can be viewed as a formalized version of the Liar, and Löb's of Santa Claus. Interestingly, Löb's theorem predated the informal version of the paradox, which was the suggestion of one of his paper's referees.

'If A is an abelian group, then, for all a, b in A, the equivalence

$$\overbrace{(a \circ b) \circ \ldots \circ (a \circ b)}^{n\ times} = \overbrace{(a \circ \ldots \circ a)}^{n\ times} \circ \overbrace{(b \circ \ldots \circ b)}^{n\ times}$$

holds'

> ...On the other hand, instead of a book, imagine a proof development system for algebra; there the theorem cannot be stated, since it is not a theorem of abelian group theory, it is, rather, a meta-theorem, a theorem *about* abelian group theory.

Underlying this is the implicit assumption that we are talking about an axiomatization in first order logic. In a higher order logic, or in an embedding in set theory (conventional foundational systems for mathematics) iterated operations and algebraic structures are definable, and the argument collapses. Indeed, to take this argument to its extreme, if I allow only arithmetic statements involving ground terms in my 'logic' then anything with a variable in it is a metatheorem. This is not a frivolous point. According to Edwards (1989) a major part of Kronecker's objections to 'Cantorian' mathematics was that for him it made no sense to talk about 'all functions' or even 'an arbitrary function', even though the corresponding statement for a particular instance was something he found perfectly acceptable. A comment of similar import made by the referees of the paper by Aiello and Weyhrauch (1980) is parried thus:

> ...in elementary algebra we are taught how to manipulate equations ...This manipulation of equations (i.e. syntactic expressions) is straightforwardly metamathematical. It is the ability to do this *directly* that makes our formalization attractive.

This seems to be making the weaker claim that proof strategies and manipulative techniques, which connect mathematical results and are used to derive them, can be said to be outside the formal system. This is true — however it is not clear that a purely algorithmic metalogic (i.e. the ML programming language) is inadequate for this, and LCF-style systems have it. In any case, as we shall see below, there is no difficulty in incorporating syntactic techniques into a higher order framework without an explicit metatheory. Is it evident that there is any need for a more elaborate metatheory? Well, one sometimes sees in mathematics books assertions like 'the other cases are similar', 'we may assume without loss of generality that $x < y$', or 'the case of an upper semilattice is treated analogously'. These have the flavour of metatheorems in that they relate similar *proofs* rather than *theorems*. But bear in mind that when presenting proofs informally, this is just one of many devices used to abbreviate them. Certainly, a way of formalizing statements like this is as metatheorems connecting proofs. But another is simply to do (or program the computer to do!) all the cases explicitly. There seems no reason to suppose this is inadequate in practice. Performing similar proofs in different contexts may anyway be a hint that one should be searching for a suitable mathematical generalization to include all the cases. For example, the similarity of the proofs of arithmetical theorems for different kinds of limits (pointwise limits of real functions, limits of real functions at infinity, limits of real sequences) leads to more general limit notions like nets and filters — see for example Dudley (1988).

Of course, we are not denying that there are many fascinating things one can do with proofs. For example, some very interesting work has been done on finding computational content of classical proofs via a generalized double-negation translation in Nuprl; see for example the paper by Constable and Murthy (1990). There are lots of other interesting avenues of investigation, for example translating proofs of analytic number theory into very weak systems of arithmetic, as described by Takeuti (1978). However we claim:

1. Such interests are largely the preserve of logicians rather than 'mainstream' mathematicians, and are of no obvious relevance in verification work.

2. It is unusual to want to feed the results back into the system; their interest arises as an independent piece of mathematics.

3. Many of the manipulations are algorithmic, and can be investigated using only minor modifications of the LCF method (for example, storing proof trees as concrete objects, as already done by several systems including Nuprl).

Metalogical frameworks

In response to the proliferation of theorem provers for different logics, a number of 'generic theorem provers' or 'logical frameworks' have been developed. Well-known examples include the Isabelle system described by Paulson (1994), Lambda Prolog described by Felty and Miller (1988) and the LF system described by Harper, Honsell, and Plotkin (1987). These provide a simple metalogic in which different object logics can be represented. Proof procedures are provided which are intended to be applicable to a wide variety of logics. The objective is to avoid needing to write a new theorem prover from scratch for each new logic one is interested in. Users, though they may in fact be proving facts at the metalevel, often need not be aware of the nature of the metalogic.

Usually the metalogic chosen is rather weak; Isabelle for example uses intuitionistic second order logic based on lambda calculus, without any numbers or recursive data types. This is ideal for formalizing object logics; in particular, binding constructs like universal quantifiers can be implemented using lambda-calculus binding in the metalogic. The framework is designed to support object-level reasoning in a uniform manner. However it is not sufficient to perform much metatheoretic reasoning beyond the fact that a particular object-level statement is a theorem. In fact the metalogic is usually so weak that even proving the Deduction Theorem for a Hilbert-style object axiomatization is impossible.

It is of course possible to extend the metalogic. As was pointed out by Randy Pollack, the LF logic can be embedded as a natural subset of the much more powerful Calculus of Constructions, as implemented in the Coq and LEGO provers. Pollack, in unpublished work, proved the Deduction Theorem for a Hilbert-style propositional logic in the LEGO prover, and Taylor (1988) verified a tactic for LF encodings in LEGO, which proves equations in an associative semigroup using list equality of the fringes of the term trees.

Alternatively one can break away from traditional framework logics. Matthews (1994) explores the use of Feferman's FS_0 system as a metalogic. This has independent interest as FS_0 was proposed theoretically by Feferman (1989) for precisely the purpose of representing formal systems — perhaps even as an explication of what constitutes a formal system. Matthews gives a detailed sketch of how to prove cut-elimination for propositional logic in FS_0 (the algorithm is simple enough, but the usual textbook formulation uses induction over higher ordinals, not directly available in FS_0, for the termination proof). For a general plea for more extensive use of metatheoretic reasoning, see the paper by Basin and Constable (1991).

Metatheoretic proofs share something with verification proofs: they are mostly a detailed and messy technical execution of a fundamentally simple, though often ingenious, idea. Nevertheless, as the reader may have guessed from the examples cited, the existing work on metatheoretic proof in computer theorem provers is surprisingly limited. Probably the most substantial example is the proof of Gödel's first Incompleteness Theorem by Shankar (1994), but apparently this was done without any interest in exploiting it metatheoretically in the theorem prover.

Seamless use of metatheory

The power of metatheoretic reasoning arises from the ability to step back from the constraints of a formal system and exploit syntactic properties, and their connection with semantics. For example the Löwenheim-Skolem theorems prove that there exist groups, rings, fields, Boolean algebras and any first order axiomatizable structures of arbitrary infinite cardinality. By means of a *syntactically based* classification of structures we are able to prove a theorem of attractive generality. A more substantial example is 'Lefschetz's principle' in algebraic geometry, pithily but imprecisely stated by Weil (1946) as:

> There is but one algebraic geometry of characteristic p.

Now, as pointed out by Tarski and noted by Seidenberg (1954) there is quite a simple quantifier elimination procedure for the first order theory of algebraically closed fields. Using routine logical equivalences and the facts that, for any polynomials $p(x)$ and $q(x)$, $p(x) = 0 \wedge q(x) = 0 \equiv gcd(p, q)(x) = 0$ and $p(x) = 0 \vee q(x) = 0 \equiv p(x)q(x) = 0$, it suffices to eliminate the universal quantifier from $\forall x.\ p(x) = 0 \Rightarrow q(x) = 0$. But in an algebraically closed field $p(x)$ splits into linear factors, each of which must therefore divide $q(x)$. Hence the above is equivalent to $p(x)$ dividing $q(x)^d$ where d is the (formal) degree of $p(x)$.

Consequently, once the characteristic is specified (allowing us to decide ground sentences), the first order theory is complete, and all models are elementarily equivalent. So Lefschetz's principle is literally true if we restrict ourselves to first order statements. It is possible to take it further, again using techniques of mathematical logic, as showed by Eklof (1973). These and other examples of applications of mathematical logic to pure mathematics are surveyed by Kreisel (1956), Robinson (1963), Kreisel and Krivine (1971) and Cherlin (1976).

Higher order logic provides sufficient resources to carry out what is essentially metatheoretic reasoning (in the sense that it operates on syntactically demarcated subsets of the logic) without in any way tampering with or extending the simple inference mechanisms. Datatypes representing syntactic objects like terms can be set up, and interpretation functions into the logic defined to connect the internal representation to the corresponding constructs in logic. For example, the usual set-theoretic semantics of first order logic with respect to some interpretation and environment (valuation) can be defined in an obvious manner (bound variables are the only nontrivial consideration, and they are easily dealt with). Indeed, the simple logic of the Boyer-Moore prover, of which more later, allows an evaluation function (`MEANING`) to be defined on terms under a given variable assignment. As Boyer and Moore (1981) emphasize:

> There is nothing magic or "meta" about this function.

For reasons connected with the undefinability of truth demonstrated by Tarski (1936) it is not possible to define a semantics for the *whole* logic inside itself. Nevertheless useful subsets can be dealt with, and in rich type theories such as Nuprl, stratification by universe level allows one to come very close to this ideal. Howe (1988) has actually implemented such a scheme in Nuprl and verified an embedded term rewriting system, including matching algorithms. This appears to go beyond any of the work on reflection proper in Nuprl which we will discuss below. The stress in Howe's work was on verifying theorem proving procedures. More recently, the present author has been experimenting with proving some more 'mathematical' results using similar techniques in HOL.

Let us reiterate: in many cases everything can be done in the formal system as it stands, if that system is higher order or supports set theory. Full internalization of semantics is impossible by Tarski's theorem, which may show itself in practice by the type system providing an obstacle. Nevertheless a lot of interesting things, including more or less all the usual metatheorems about first order logic, can be done in this way.

4 Logical reflection

Gödel (1931) showed how quite simple logical theories can act as their *own* metatheory, and derived important metamathematical results from the exercise. It is a straightforward, though tedious, matter to encode formulas, and hence lists of formulas and proofs, as numbers[9]. We will write the Gödel number of a formula ϕ as $\ulcorner \phi \urcorner$. Furthermore, under any sensible encoding, all the important syntactical operations on encoded formulas, e.g. substitution, are recursive (computable), and hence so is provability itself. For a rather more detailed development, which discusses many of the points we touch on here, see Smoryński (1991).

Now even in quite a spartan number theory, the representable relations and functions are precisely the recursive ones, so provided the set of axioms S is recursive, one can define a quantifier-free predicate $Prov$ such that

$$Prov(p, \ulcorner \phi \urcorner)$$

means intuitively that p is the Gödel number of an encoded proof from S of the encoded counterpart to ϕ. (When one wishes to be explicit about the system of axioms, one writes $Prov_S$.) If we make the following abbreviation:

$$Pr(\ulcorner \phi \urcorner) = \exists p.\ Prov(p, \ulcorner \phi \urcorner)$$

then one can prove formally that the following 'derivability conditions' hold.

1. If $\vdash \phi$ then $\vdash Pr(\ulcorner \phi \urcorner)$
2. $\vdash Pr(\ulcorner \phi \urcorner) \wedge Pr(\ulcorner \phi \Rightarrow \psi \urcorner) \Rightarrow Pr(\ulcorner \psi \urcorner)$
3. $\vdash Pr(\ulcorner \phi \urcorner) \Rightarrow Pr(\ulcorner Pr(\ulcorner \phi \urcorner) \urcorner)$

The first of these is easy, since any true existential formula (i.e. one of the form $\exists x.\ P[x]$ with $P[x]$ quantifier-free) must be provable, for if $\exists x.\ P[x]$ is true, there is some n with $P[n]$. This is a purely decidable fact so $\vdash P[n]$ and by elementary logic $\vdash \exists x.\ P[x]$. The second is a routine piece of syntax manipulation since Modus Ponens is usually one of the basic proof rules. The third one is a bit harder. For the converse of 1 to hold, i.e. 'if $\vdash Pr(\ulcorner \phi \urcorner)$ then $\vdash \phi$', it is sufficient to make an additional assumption of 1-*consistency*, i.e. that all provable existential formulas are true, since $Pr(\ulcorner \phi \urcorner)$ is such an existential statement and it is true precisely when $\vdash \phi$.

Now, using a diagonalization argument, Gödel was able to exhibit a statement ϕ which expressed its own unprovability in the system.

$$\vdash \phi \equiv \neg Pr(\ulcorner \phi \urcorner)$$

[9]Similar techniques can be applied to formal systems including hereditarily finite sets or free recursive datatypes, with the possible advantage that the Gödelized form of a formula may be quite readable, instead of just a huge number.

If $\vdash \phi$, then by (1) above, $\vdash Pr(\ulcorner \phi \urcorner)$; but this means $\vdash \neg\phi$ and therefore the system is inconsistent. So assuming the system is consistent, ϕ is unprovable, and furthermore it is *true*, since it asserts precisely that unprovability. This is Gödel's First Incompleteness Theorem. Note that it only depends on the first derivability condition.

Consistency is not sufficient to rule out $\vdash \neg\phi$. However 1-consistency certainly is, since now if $\vdash \neg\phi$ then $\vdash Pr(\ulcorner \phi \urcorner)$, and since all provable existential statements are true, $\vdash \phi$, again contradicting consistency. Gödel's argument itself shows that if S is consistent, so is $S \cup \{\neg\phi\}$, and therefore consistency does not in general imply 1-consistency. Rosser (1936) modified Gödel's proof by defining[10]

$$\overline{Prov}(p, \ulcorner \phi \urcorner) \equiv Prov(p, \ulcorner \phi \urcorner) \wedge \forall q \leq p.\ \neg Prov(q, \ulcorner \neg\phi \urcorner)$$

If $\overline{\phi}$ is the analog of Gödel's sentence ϕ for this new notion of provability, it turns out that just assuming consistency, neither $\overline{\phi}$ nor $\neg\overline{\phi}$ is provable. What's more, assuming the system *is* consistent, Pr and $\overline{Pr}$ are evidently coextensive.

Gödel's argument above may itself be formalized using the provability predicate. This essentially means that instead of (1), we use (3), which can be seen as (1) 'at one remove'. Note also that by combining (1) and (2) we see that if $\vdash \psi \Rightarrow \varphi$ then $\vdash Pr(\ulcorner \psi \urcorner) \Rightarrow Pr(\ulcorner \varphi \urcorner)$. Applying this, and (2) again, to $\vdash Pr(\ulcorner \phi \urcorner) \Rightarrow (\phi \Rightarrow \bot)$, which is true by construction of ϕ (here $\bot$ denotes 'false', and for any ψ, $\neg\psi \equiv \psi \Rightarrow \bot$), we find that $\vdash Pr(\ulcorner Pr(\ulcorner \phi \urcorner) \urcorner) \Rightarrow (Pr(\ulcorner \phi \urcorner) \Rightarrow Pr(\ulcorner \bot \urcorner))$. But by (3) we also have $\vdash Pr(\ulcorner \phi \urcorner) \Rightarrow Pr(\ulcorner Pr(\ulcorner \phi \urcorner) \urcorner)$, and so $\vdash Pr(\ulcorner \phi \urcorner) \Rightarrow Pr(\ulcorner \bot \urcorner)$. On the other hand, $\vdash \bot \Rightarrow \phi$ is trivial, and so again $\vdash Pr(\ulcorner \bot \urcorner) \Rightarrow Pr(\ulcorner \phi \urcorner)$. This shows that $\vdash \phi \equiv \neg Pr(\ulcorner \bot \urcorner)$.

$\neg Pr(\ulcorner \bot \urcorner)$ is an assertion that the system is consistent, and so is usually abbreviated Con (or Con_S when one makes the system explicit). We have thus, assuming all three derivability conditions, deduced Gödel's Second Incompleteness Theorem, that a system of the kind we are considering is unable to prove its own consistency. We have shown, moreover, that the statement of consistency is logically equivalent to the unprovable sentence produced in the proof of Gödel's first theorem.

As noted by Feferman (1960), there is an *intensionality* involved in an assertion of consistency. In fact using Rosser's notion of provability $\overline{Pr}$, we get a notion of consistency $\overline{Con}$ which is (assuming the system *is* consistent) coextensive with Con yet such that $\vdash \overline{Con}$. Indeed manifestly $\vdash \neg\bot$, so we can assume $Prov(p, \ulcorner \neg\bot \urcorner)$ for some p. Now if $q \geq p$ then $\neg\overline{Prov}(q, \ulcorner \bot \urcorner)$ by definition, and since the system is assumed consistent, $\forall q < p.\ \neg\overline{Prov}(q, \ulcorner \bot \urcorner)$. The former is trivially provable in the logic, and the second must be too because it's decidable (only bounded quantification). Putting these together we find $\vdash \forall p.\ \neg\overline{Prov}(p, \ulcorner \bot \urcorner)$.

Of course this coextensiveness cannot, on pain of contradicting Gödel's second theorem, be proved inside the logic, and $\overline{Pr}$ must fail to satisfy one of the derivability conditions, but it nevertheless has some significance as regards the transfinite progressions of theories discussed below. Even using the standard notion of provability, Feferman showed that for theories like ZF and PA which can prove consistency of finitely axiomatized subsystems, one can produce alternative predicates representing the same set of axioms (using a trick reminiscent of the Rosser construction) such that Con becomes provable. This means one needs to distinguish carefully between 'natural' and 'pathological' representations of the same axiom set. Resnik (1974) discusses the philosophical significance of this fact.

[10]Strictly, $\ulcorner \neg\phi \urcorner$ should be read as $Neg(\ulcorner \phi \urcorner)$ where Neg is the negation function on encoded formulas.

Reflection principles and transfinite progressions

While Gödel's theorems show the limitations of formal systems, they also point to a systematic way of making a given system stronger. Given some axiom system S_0, a natural way of strengthening it is the addition of a new axiom amounting to a statement of S_0's consistency.

$$S_1 = S_0 \cup \{Con_{S_0}\}$$

This gives a new system, and a corresponding new provability predicate and assertion of consistency. Now the procedure can be iterated, giving S_2, S_3 and so on. The iteration can even be continued transfinitely:

$$S_\lambda = \bigcup_{\alpha<\lambda} S_\alpha$$

This was first investigated by Turing (1939), who showed that the limiting system (unioning over all constructive ordinals α) arising from Peano arithmetic by repeatedly adding statements of consistency was capable of proving all true universal sentences of number theory (i.e. those of the form $\forall x.\ P[x]$ with $P[x]$ quantifier-free). For example, it could prove Fermat's Last Theorem, if true. Turing's explorations were carried much further by Feferman (1962), who coined the term 'reflection principle' for an assertion, like a statement of consistency, which amounts to an expression of trust in a system of axioms (presumably so called because it arises by 'reflecting upon' those axioms from outside).

> In contrast to an arbitrary procedure for moving from A_κ to $A_{\kappa+1}$, a reflection principle provides that the axioms of $A_{\kappa+1}$ shall express a certain trust in the system of axioms A_κ.

A stronger reflection principle would be an assertion of soundness with respect to some standard model, e.g. the natural numbers for arithmetic theories.

$$\vdash Pr(\ulcorner\phi\urcorner) \Rightarrow True(\ulcorner\phi\urcorner)$$

As it stands this so-called *global reflection schema* cannot be expressed in the logic, since it was shown by Tarski (1936) that there is no definable predicate $True$ corresponding to arithmetic truth. However we can express something intuitively similar by the following schema (in which ϕ is any sentence):

$$\vdash Pr(\ulcorner\phi\urcorner) \Rightarrow \phi$$

This schema, now known as the *local reflection schema*, was also considered by Turing. Since the special case where ϕ is $\perp$ is a statement of consistency, this statement is at least as strong. In fact it was proved by Löb (1955) that an instance of the above schema is provable precisely when the corresponding ϕ is itself already provable. Note that Gödel's second theorem is a special case of Löb's theorem, again setting ϕ to $\perp$.

Turing conjectured that the limiting system from repeatedly adding the local reflection schema would be properly stronger than that resulting from repeatedly adding a statement of consistency. This conjecture was refuted by Feferman: in fact the limiting systems have equal power. However a still stronger schema, the *uniform reflection principle* [11]

$$\vdash \forall n.\ Pr(\ulcorner\phi[n]\urcorner) \Rightarrow \phi[n]$$

[11] $\ulcorner\phi[\mathbf{n}]\urcorner$ is really a shorthand for something like $subst(n, \ulcorner\phi\urcorner)$, i.e. the result of substituting the encoding of numeral n for the unique free variable of ϕ. This function $subst$ is primitive recursive.

was shown by Kreisel and Lévy (1968) to be, with respect to first order number theory among others, equivalent to transfinite induction up to ε_0, which was precisely the additional property used by Gentzen in his consistency proof for number theory[12], and Feferman showed that a transfinite iteration based on it proves all true sentences of number theory.

It is possible for a consistent theory to become inconsistent on the addition of the local reflection schema, or even a simple statement of consistency. For example, Gödel's theorem shows that if S is consistent, so is $T = S \cup \{\neg Con_S\}$, but Con_T implies Con_S, so the further addition of Con_T to T yields an inconsistent system. However it follows from Feferman's work that a 1-consistent system remains so even on transfinitely many additions of the uniform reflection schema.

A more recent exposition of such matters is given by Feferman (1991). Since his original coining, the meaning of 'reflection principle' has become slightly specialized towards statements like the local reflection schema which seem to make a connection between a theory and its metatheory of the form 'if ϕ is provable then it is true'. For example, Kreisel and Lévy (1968) say:

> By a "reflection principle" for a formal system S, we mean, roughly, the formal assertion stating the soundness of S: If a statement ϕ (in the formalism S) is provable in S then ϕ is valid.

It may be that a slight shift in the perceived metaphor is behind this changed usage: the Gödelization is a representation which mirrors the actual proof system.

Rather than stressing its role in making a logical system stronger, Kreisel and Lévy exploit the fact that the reflection schema for S is unprovable in S to yield a way of comparing the strengths of logical systems. If a system T can prove the reflection principle for S, then T is properly stronger than S. It is claimed that the intuitive significance of the reflection schema tends to make such proofs easier to find than those for a simple statement of consistency which is, as noted above, a special case and thus in principle weaker. Smoryński (1977) also gives results on the strength of reflection principles.

Finally, we should observe that although the full reflection schema is unprovable, it may happen that by suitably restricting the kinds of provability allowed in $Prov$, the analogous schema becomes provable. In particular, this happens in Peano Arithmetic and Zermelo-Fraenkel set theory, if provability is only allowed from a fixed finite set of the axioms. (We discuss this situation for ZF set theory in an appendix.) Following Troelstra (1973), where the logical complexity of formulas in proofs in Heyting arithmetic is restricted, we refer to these as 'partial reflection schemas'.

5 Computational reflection

In contrast to the reflection principles above, consider the following reflection *rule*:

$$\frac{\vdash Pr(\ulcorner \phi \urcorner)}{\vdash \phi}$$

The addition of this may be inconsistent (again, consider the 1-inconsistent system T above). However 1-consistency guarantees not only that the new system is 1-consistent, but actually has the *same theorems*. Indeed, 1-consistency includes assertions of the form 'if $\exists p.\ Prov(p, \ulcorner \phi \urcorner)$ is provable then it is true', and

[12]Weil referred to Gentzen as the lunatic who justified induction on the natural numbers, ω, using induction on a higher ordinal.

$\exists p.\ Prov(p, \ulcorner\phi\urcorner)$ is true precisely when $\vdash \phi$. For this form of reflection, we might work in two different logical systems S and T. Then the following rule:

$$\frac{\vdash_S Pr_T(\ulcorner\phi\urcorner)}{\vdash_T \phi}$$

is, provided S is 1-consistent, a conservative extension of T. In what follows we will usually assume, for simplicity, that we are dealing with a single logical system. In this case, it is important to note that the provability predicate Pr is an arithmetization of the *original* notion of provability (without the new reflection rule). By assuming 1-consistency, we have noted that the new notion of provability is extensionally the same as the old one. However since we certainly can't prove 1-consistency *inside* the system, we should not expect to be able to duplicate this fact there. So if one wishes to embed applications of the reflection rule inside the formalized proofs, the natural arithmetization gives a new notion of provability Pr_1. Then we have for example:

$$\vdash Pr_1(\ulcorner Pr(\ulcorner\phi\urcorner)\urcorner) \Rightarrow Pr_1(\ulcorner\phi\urcorner)$$

If one then wishes to apply reflection based on Pr_1 inside a formalized proof, yet another provability predicate Pr_2 results, and so on. It is not possible to close up this procedure with a single syntactic notion of provability $\overline{Pr}$ which satisfies the three derivability conditions, for then:

$$\vdash \overline{Pr}(\ulcorner\overline{Pr}(\ulcorner\phi\urcorner)\urcorner) \Rightarrow \overline{Pr}(\ulcorner\phi\urcorner)$$

and by Löb's theorem and 1-consistency, $\vdash \phi$ for any ϕ and the system is inconsistent. However one can work with an infinite tower of provability predicates in several ways — see the later description of the Nuprl work for two examples.

Now, suppose for some predicate DP and a class of formulas ϕ we can prove the following:

$$\vdash DP(\ulcorner\phi\urcorner) \Rightarrow Pr(\ulcorner\phi\urcorner)$$

In that case the following schema is also a conservative extension:

$$\frac{\vdash DP(\ulcorner\phi\urcorner)}{\vdash \phi}$$

This forms the basis for *computational reflection* in theorem proving. DP might be a recursive predicate encoding an efficient decision procedure for formulas whose proofs were otherwise difficult. Now, first we must prove the 'correctness' theorem $\vdash DP(\ulcorner\phi\urcorner) \Rightarrow Pr(\ulcorner\phi\urcorner)$ for some class of formulas ϕ, and thereafter may prove $\vdash DP(\ulcorner\phi\urcorner)$ in order to deduce $\vdash \phi$. In fact, DP may itself include a condition that the formula concerned is in the chosen class, allowing the correctness theorem to be proven without restriction on ϕ.

Generalities

The idea of computational reflection is not to make a formal system stronger, but rather to make its deductive process more efficient by utilizing information which avoids having to construct formal proofs in full detail. The first question to ask is whether we should expect to achieve worthwhile gains in efficiency. Davis and Schwartz (1979), among the earliest advocates, do not marshal any convincing arguments:

> On the basis of ordinary mathematical experience we have every reason to expect that the difficulty (in the precise sense we have defined) of various important theorems will be greatly decreased this way. Although we have been unable to formulate and prove any metatheorems that would serve as a formal demonstration of this conjecture, we can point to some suggestive evidence. It is well known in proof theoretic research that the addition of new rules of inference to so-called cut-free systems can drastically decrease the length of proofs. ...we have seen that the introduction of an appropriate "algebra" rule of inference shortens to 1 the difficulty of a sentence which asserts an algebraic identity.

The appeal to 'ordinary mathematical experience' is vague. Experience with Mizar and HOL in pure mathematics has shown no hint that expansion into primitive inferences is not feasible. Admittedly, only a tiny part of mathematics has been formalized, but there seems no obvious reason to expect any other branches to be different in this regard. This is conjectural — maybe proofs involving a lot of geometric insight will prove hard to formalize for example — but at least our conclusion is supported by *some* practical experience. The Bourbaki project has developed large parts of mathematics based, notionally, on set theory as covered in the first volume. How large the proofs would be if written out in formal detail is an interesting question.

Since cut-free sequent proofs are a theoretical device and nobody would dream of basing a general computer theorem prover on them, the second point is irrelevant — it's hardly surprising that by artificially hobbling the logic one can make the sizes of proofs explode. In a similar way, the Deduction Theorem in Hilbert-style proof systems can dramatically shorten proofs, obviating the need for a lot of intermediate steps, but as conceded by Matthews, Smaill, and Basin (1991) a theorem prover is not likely to be based on a Hilbert-style axiomatization.

Finally, judging efficiency on the basis of the number of inferences, without regard to the computational complexity of those inferences, is not acceptable if the argument is supposed to be one of practical utility. We discuss this in more detail below.

Theoretical potential

From a theoretical perspective, it is certainly true that many theorems have unfeasibly long proofs[13]. For most interesting deductive systems, there can be no (total) recursive bound on the length of the smallest proof of ψ in terms of $\ulcorner\psi\urcorner$, since that would make the logical system decidable, only bounded proof search being required. Indeed, a modification of Gödel's diagonalization argument allows us, given any total recursive function f, to exhibit a sentence ϕ with the following property [14]

$$\vdash \phi \equiv \forall p.\ Prov(p, \ulcorner\phi\urcorner) \Rightarrow length(p) > f(\ulcorner\phi\urcorner)$$

Now, whether or not there are proofs of ϕ whose length is bounded by $f(\ulcorner\phi\urcorner)$ is decidable, and so ϕ is provable iff it is true. But it cannot be false, since that would mean there exists a proof of it (indeed, one within the stated length bounds). Therefore ϕ must be true, and have no proof within the stated bounds; on the other hand it does have some proof. Thus we have exhibited a sentence which is true and even provable, yet such that all proofs are unfeasibly long (given some

[13] Interpreting 'length' as something like the number of symbols in the proof based on a finite alphabet, which is a reasonable measure as far as practical feasibility goes.

[14] This is a slight gloss; though all recursive functions are representable, in general they may be represented by a relation.

suitable f, which might be a very large constant function for example). Gödel (1936) has pointed out that the lengths of proofs of already provable facts may decrease dramatically when a logic is extended to higher orders — for a recent treatment see Buss (1994). We might hope that the reflection rule will allow similar savings.

This however doesn't justify the quote above, which contained the crucial word 'important'. The real question is: are sentences like ϕ just theoretical pathologies, or are we likely to hit a theorem with no feasible proof in the course of using a theorem prover in mathematics or verification? As far as I know, nobody has given any grounds for deciding this question, though it is discussed by a few papers in Clote and Krajíček (1993). At the moment, there are no examples, but that doesn't prove anything. There is an analogy with the incompleteness of Peano arithmetic; theoretical examples of unprovable sentences have been known since Gödel's work, but only very recently have there emerged examples, like those presented by Paris and Harrington (1991), which could conceivably be called mathematically mainstream. It might be very much harder to find an unfeasible statement of say, higher order logic or set theory. Indeed, systems of *natural deduction* are not so-called through some accident, but because it appears they really correspond to how mathematicians prove theorems. As Gentzen (1935) says:

> It is remarkable that in the whole of existing mathematics only very few easily classifiable and constantly recurring forms of inference are used, so that an extension of these methods may be desirable in theory, but is insignificant in practice.

That certainly seems the message of the, admittedly limited, usage of theorem provers in pure mathematics. On theoretical grounds, we cannot bury the LCF approach till someone comes up with an example. If and when they do, it might be so exceptional that a one-off extension of the axiom system is a reasonable response. That such examples should frequently recur seems highly implausible. In any case, adding a reflection principle to help us deal with such instances still yields a recursively enumerable set of theorems — we are after all talking about a machine implementation. So this too has its own unfeasible statements, and why should *they* be any less likely to occur in practice?

Practical potential

We have seen that as yet there is no evidence for supposing the LCF approach inadequate for proofs in pure mathematics. However, theorem provers are more usually used for verification tasks, and here the theorems tackled are rather different: bigger but shallower. It seems possible that the sheer size of theorems will create new problems for the LCF approach, i.e. that the size of fully-expanded proofs will have much poorer asymptotic behaviour than the complexity of some other decision procedure. Consider the following system of classical biconditional logic. The only logical connective is bi-implication ($\equiv$). The axioms are all substitution instances of:

$$\begin{array}{ll} \vdash & p \equiv p \\ \vdash & (p \equiv q) \equiv (q \equiv p) \\ \vdash & (p \equiv (q \equiv r)) \equiv ((p \equiv q) \equiv r) \end{array}$$

and the sole rule of inference is the following variant of Modus Ponens (aka *Detachment*):

$$\frac{\vdash p \equiv q \quad \vdash p}{\vdash q}$$

Leśniewski (1929) pointed out that a formula is provable in this system if and only if every propositional variable in it occurs an even number of times. He also showed that this is equivalent to being valid in the usual sense, so he same observation obtains for the equivalential fragment of any conventional axiomatization of propositional logic. Using this metatheorem we can justify saying a formula is provable simply by pairing off the propositional variables. A less striking but more realistic example is algebraic simplification using associative and commutative laws and/or cancellation. Given an assertion of the form:

$$a_1 + \ldots + a_n = b_1 + \ldots + b_n$$

we can justify its truth just by showing that the sets $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$ are equal. To produce a proof by primitive inferences, though, we need to delicately rewrite with the associative and commutative laws to make the two sides identical.

But let us look more critically at this example. How are we to compare the sets $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$? Of course it depends on how terms are represented inside the theorem prover. But in any case the problem of testing n-element sets for equality is known — see Knuth (1973) for example — to be $O(n^2)$ in the worst case (assuming a constant-time pairwise equality test is available). Even if the subterms are pairwise orderable somehow, we can't do better than $O(n\ log(n))$. It's not hard to devise algorithms using primitive inferences which have the same complexity. (Such a thing exists in HOL, called `AC_CONV`.) Even if it weren't possible, we could use the techniques explained earlier to write a higher order function operating on 'syntax' without any special metatheory. Of course there may be a significant constant factor difference, but nevertheless, the example is hardly especially persuasive from the perspective of efficiency. Perhaps there is an argument that the LCF-style coding is much less *natural*. To an extent this is true, but then any encoding of an algorithm in a strict formalism like a programming language is 'unnatural' — it's just a question of degree.

It seems, then, that to get worthwhile gains from reflection, we may have to move away from such simple examples, and consider the kinds of complex special-purpose algorithms which are sometimes implemented in provers. But now we want to reason about real programs, and *execute* them as real programs, rather than simulating them inside the logic, or we will encounter a dramatic slowdown, probably worse than sticking to the LCF approach all along! This raises a host of new questions.

Computational reflection and code verification

A reasonable logic is quite capable of representing a wide variety of theorem proving procedures on encoded formulas as recursive functions inside the logic. The higher order type theory of systems like HOL and Nuprl corresponds nicely to an idealized functional programming language — one where all functions are total. Similarly NQTHM's logic corresponds closely to pure LISP. Verification of nontrivial algorithms may already be within reach in this way. For example Aagaard and Leeser (1994) have verified a Boolean simplifier formalized in Nuprl, corresponding to about 1000 lines of Standard ML code. However, if we are going to verify an abstract recursive function in the logic and and then run an implementation in a real programming language (probably the implementation language of the theorem prover — LISP, ML or whatever), we should ask:

1. How exactly do we regiment the process of iteratively adding code to a running implementation? We are trying to repair the hull of a ship while it is sailing, without bringing it into dry dock.

2. How do we justify the correspondence between an idealized mathematical description inside the logic and an implementation in a real programming language? We know that the latter has awkward features like finite limits for arithmetic, complicated evaluation orders and subtle semantics for nontermination and exceptional conditions.

3. How do we represent inside the logic imperative language features like arrays and pointers? As we have argued above, it is important to reason about these features if we want to implement certain important proof procedures effectively.

A satisfactory answer to the first question depends on the implementation language. In languages such as LISP the seamless use of compiled and interpreted code makes it much easier. In an LCF-style system, there are some quite formidable problems; we must somehow rip open an abstract type, tinker with it to add a new constructor, and then close it up again. An alternative is to perform the final addition of code 'informally' after performing the proof, then restart the enhanced system. The user must take responsibility for correctly sequencing the additions and editing the source code. We might call this 'informal reflection'.

A completely satisfactory answer to the second and third points is: don't just verify an abstract version of an algorithm, verify actual code using the formal semantics of the implementation language[15]. This means embedding the syntax and formal semantics of the implementation language in the logic. Here we make several presuppositions, most notably that the implementation language has a suitable formal semantics which is stable and likely to be adequate and tractable for serious proofs. These are properties satisfied by very few, if any, languages. The difficulty of code verification of this kind is serious, and as far as we are aware no substantial examples exist. Finally, porting the theorem prover to another language (e.g. a different ML dialect) becomes much harder, since not only must the system itself be modified, but so must the proofs based on the formalized semantics of the old language.

All existing instances of reflection make the leap from an abstract to a concrete implementation via a naive syntactical transformation without complete formal justification. The correctness proof in abstracto is much easier, and perhaps for most practical purposes the distinction between the abstract description and the concrete implementation is not likely to trip one up. Nevertheless the third point is still problematical; it is our belief that for many efficient proof procedures, imperative code is required, and it is not so easy to associate such constructs with parts of the logic in a convincing way. Arrays can be identified with functions, as in some versions of Floyd-Hoare logic, but care needs to be taken over indexing exceptions, and the problem of aliasing makes any proofs much harder.

Another difficult question is: what do we mean by 'correct' in this context? The minimal requirement is that if a procedure terminates, it always produces something valid. This is a partial correctness condition — we do not prove termination — and may well be the best we can do in many cases. Resolution methods performing unbounded search, or algorithms based on that of Huet (1975) for higher order unification (used with great success in practice in the TPS, Isabelle and LAMBDA systems) may fail to terminate. In the case of complicated heuristic procedures, it

[15]Of course one can still doubt that the compiler correctly implements the language, or that various issues abstracted away, like running out of memory, are really irrelevant; but these are not new problems arising with reflection.

may be quite impossible to demarcate formally those instances where termination is to be expected.

Even if termination can be proved, the proof may require much more complex mathematics than that required to prove partial correctness. We are not aware of any particularly convincing practical instances, but justifying bounds for the solution of Diophantine equations requires deep analytic number theory — see the work of Baker (1975) in this regard. It may even happen that termination is only provable in a stronger logic — following the work of Kreisel (1952) it has become a fashionable research topic to classify logics on the basis of which functions they can prove total. In the unlikely event that we want to actually prove complexity bounds, we might even need to go beyond the resources of present-day mathematics. For example, as a consequence of work by Ankeny (1952), the efficiency of some important number-theoretic algorithms, e.g. for primality testing as discussed by Bach (1990), is apparently dependent on the truth of the Extended Riemann Hypothesis.

The main reason the question of termination should interest us is that most of the abstract versions of proof procedures are modelled in the logic as total functions. Reasoning about partial functions is widely believed to be much more difficult. Nevertheless, a serious attempt to address partiality in reflected proof procedures is surveyed by Giunchiglia, Armando, Cimatti, and Traverso (1994).

A more extreme point is that many of the most efficient algorithms *aren't* correct in a strict sense. They may assume that machine arithmetic will never overflow for example. Now it may be an entirely justifiable assumption on the basis that machine resources would become exhausted before it could happen. But to carry through details like this in a formal proof is much more complicated. To take an analogy with physics: if I want to analyze the dynamics of a person riding a bicycle, I intuitively *know* that I can neglect relativistic effects, and use classical mechanics. But arriving at a general theorem from which this fact can be read off might be extremely difficult — harder than simply applying relativistic mechanics in the first place.

6 Computational reflection in practice

Reflection is a popular topic for investigation in theorem proving, and one might expect large numbers of real systems to have experimented with it. On the whole, experiments have been limited to small projects. We shall examine three significant implementations, and one relevant proposal.

FOL

Perhaps the first actual use of reflection in a formal reasoning system was by Weyhrauch (1980). His FOL system is an implementation of first order logic which allows reasoning in multiple theories. It provides a notion of 'simulation structure', which may be described as a computable, partial model. A full model is unattainable since even facts about ground terms may be uncomputable. However one can associate with a logical system L some simulation structure S giving a restricted part of the information that a model would provide, in the shape of some kind of evaluator. For example, one might associate with the addition symbol an evaluator which rewrites expressions involving addition and other ground terms. This process of association is called 'semantic attachment', because it stands in place of a full model. The resulting couple is called an 'L/S pair' or 'context' (in some related work a context includes the currently proved set of theorems). The standard set-theoretic account of first order logic semantics has been tweaked to use simulation structures rather than full models by Weyhrauch and Talcott (1994).

One of the contexts in FOL is called META, and formalizes the syntax of FOL's own logic, including the structure of formulas and logical derivability. Then a reflection principle is asserted which justifies a transition between $\vdash_{object} \phi$ and $\vdash_{meta} Pr_{object}(\ulcorner \phi \urcorner)$. The principle allows these statements to be interderived both ways; these processes are usually referred to as 'reflection up' (left to right) and 'reflection down' (right to left).

At its simplest, this connects inference in the object theory with computation in the metatheory. The syntax operations and inference rules of the object logic are just function symbols in the metalogic, and may, by semantic attachment, be associated with the natural operations on formulas. This identification of 'theorem proving in the theory' with 'evaluation in the metatheory' (to coin an FOL slogan) is reminiscent of LCF. However since a full logic is available to formalize the metatheory, it is more general. Metatheorems can be proven which justify certain kinds of inference without needing to expand down to the original primitives. A simple illustrative example is given in Weyhrauch (1982) based on a Hilbert-style axiomatization of propositional logic.

The obvious defect of the FOL approach is that there is no check on the user attaching arbitrary actions to function symbols. Logical consistency is not enforced. For example, one might make the evaluator transform $1 + 1$ into 1. This doesn't affect FOL's appeal as an AI project, but for the approach to make inroads into the formal verification community something better is needed.

This is being worked on by a number of researchers in Italy. FOL has been reimplemented and reengineered as GETFOL, and attempts are being made to achieve formal demarcation of acceptable proof strategies rather than permitting arbitrary attachments. An interesting summary is given by Armando, Cimatti, and Viganò (1993). Proof strategies in the object theory are simply terms in the metatheory. It is possible to demarcate, purely syntactically, terms which implement safe proof strategies; so-called 'logic tactics'. These are built from existing primitives using a few simple connectives like the conditional, much as in LCF. However it is possible to use more general metatheoretic reasoning should it prove necessary. The next step being considered is to compile proven proof procedures down to the implementation language (apparently using a naive transliteration) to make them more efficient than interpreting them in the metatheory. This 'flattening' process has already been investigated in distinct but related work by Basin, Giunchiglia, and Traverso (1991).

NQTHM

The NQTHM prover, described in detail by Boyer and Moore (1979), is a fully automatic theorem prover for a quantifier free first order logic. The logic has its roots in Primitive Recursive Arithmetic (PRA), as developed by Skolem and Goodstein (1957), but allows arbitrary recursive types, not just the natural numbers. There is no separate class of formulas. An induction rule is available, but there are no explicit quantifiers. The logic is represented using LISP syntax. The prover has no real interactive features, but the user may direct the prover by choosing a suitable chain of lemmas, each of which can be proved automatically.

One of the earliest practical applications of reflection in a major theorem prover, indeed *the* earliest where the stress was placed on soundness, was the work of Boyer and Moore (1981) in adding a reflection principle to NQTHM. (Note that they do not use the word 'reflection' to describe the process, but rather talk about adding 'metafunctions'.) This was then used to implement a simplifier which performs cancellation in arithmetic equations. Other simplifiers, including a tautology checker, have also been verified. Apparently metatheoretic extension is a facility not widely used in practice, but it is not just a theoretical flight of fancy.

The implementation of reflection in the Boyer-Moore prover does not entail the full internalization of the logic's rules of inference. Rather, a denotation function `MEANING` is defined (just an ordinary recursive function, as has been mentioned already). This gives the value of an encoded term (the coding is rather simple, since the logic is in LISP syntax to start with) under a given assignment to variables.

If the user wishes to introduce a term-transforming function `fn` as a new logical primitive, then first, as usual with definitions in NQTHM, the prover must show that the recursion equations given define a unique total function. However in more recent work, `EVAL` is used instead of `MEANING`, and the former allows *partial* recursive functions. Nevertheless, most users stick to total functions, probably because they are easier to work with.

Then the prover must prove a metatheorem, which states that for any formula `F` and any assignment `A`, then first `fn(F)` is also a formula (the system is untyped, so this is not automatic), and second that `MEANING(A,F) = MEANING(A,fn(F))`. That is, the simplified term is always equal to the original term under any variable assignment. Note that the transformation function may be parametrized by free variables, provided it obeys the above strictures under all assignments.

If these obligations are proved, the system installs compiled INTERLISP code corresponding to `fn`. This jump deserves detailed consideration. INTERLISP does not behave quite like the encoded abstract pure LISP environment. In particular there is a finite limit for integers, with silent wrapping on overflow. Boyer and Moore pay careful attention to these difficulties, using for example a custom addition function `badd1` which fails on overflow. One or two other tweaks are applied, mostly for reasons of efficiency rather than correctness. The approach is careful, but informal. Recent versions of NQTHM are based on Common LISP, which has bignums, obviating the need for many of the precautions.

Another point to note is that what in some systems would be automatic type correctness conditions (for example, that `fn` maps terms to terms), must, since we are in the untyped world of LISP, be proved explicitly. Indeed, if the system has already been extended unsoundly (the user may posit inconsistent axioms, precisely in order to derive an inconsistency!) then the proof may be nonsense and the newly installed metafunction might not just derive falsity, but fail in arbitrary and damaging ways. In more recent work this has been guarded against.

It has to be said that the cancellation function is easily implemented in the LCF style with adequate efficiency. But Boyer and Moore report that the correctness proof was actually rather easy (taking one of the authors just a day), so we should not take this example as truly indicative of the potential state of the art. We are not aware of any verification of imperatively implemented metafunctions, though there is extensive work on imperative program verification in NQTHM, e.g. that described by Boyer and Yu (1992). Recently Moore (1994) has produced an efficient, purely applicative version of the BDD algorithm, whose verification may be tractable, though it does depend on hash tables.

Nuprl

Nuprl, described by Constable (1986), is an LCF-descended theorem prover which supports an extension, including for example inductive types, of a type theory as described by Martin-Löf (1985). It is intended as an environment for constructive mathematics and computer programming, and for exploring their connections. The richness of the Nuprl type theory generates a profusion of additional primitive inferences compared with HOL[16]. Furthermore the system stores proof trees as concrete objects, so invocation of primitive inferences is particularly expensive in space and

[16]It is not clear whether the kinds of optimizations we have been looking at above, in particular cacheing of theorems, would render this less problematic.

hence garbage collection time. Finally, in applied work, decision procedures have often been tacked on to Nuprl, and the user community is more accustomed to them; it is however desirable to place them on a firmer theoretical footing. All these facts make computational reflection especially appealing.

We have already discussed the work of Howe, where the connection between an internalization of derivability and the logic itself is made via a function in the logic. However Nuprl researchers have also experimented with adding explicit reflection rules allowing the deduction of $H \vdash G$ from $\vdash Pr(\ulcorner H \vdash G \urcorner)$. As noted already, there are problems in fixing a single provability predicate Pr allowing embedded instances of the reflection rule. In Nuprl, two possibilities have been pursued.

The first is explained in Knoblock and Constable (1986). The idea is that repeatedly adding the reflection rules gives rise to a sequence of logics PRL^0, PRL^1, PRL^2, ... The reflection rule connects formalized provability at one level to the logic in the level below it. This is rather complex because there are an infinite number of levels. Nevertheless for many purposes 'provability in some PRL^k' can be taken as the standard notion of provability. As far as we are aware, no practical work has been done using this scheme.

The second alternative, described in Allen, Constable, Howe, and Aitken (1990) is to have just one logic and restrict the reflection rule. The reflection rule is parametrized by a natural number called the 'reflection level', and in any instance of the reflection rule, embedded instances must have a lower reflection level. With this feature the reflection rule remains a conservative extension to the logic, although its eliminability from proofs cannot be proven *inside* the logic.

A further step in this direction is described by Howe (1992). He proposes a slight modification of the Nuprl type theory, including for example a 'denotation' (better: 'evaluation') function, which allows a particularly clean internalization of the logic's semantics. (The presence of dependent types blurs the distinction between syntax and semantics, so one shouldn't read the terminology used here too critically.) The semantics is stratified by universe level; at a given level the evaluation semantics of lower universe levels can be completely formalized. Using this internal semantics, Howe was able to derive the reflection rule without extending the logic.

Most of the work in reflection proper appears to be theoretical, and we are not aware of any practical applications. The position of the Nuprl community on some of the issues raised above regarding real programming languages is not entirely clear.

HOL

No work on reflection has actually been done in HOL, but Slind (1992) has made some interesting proposals. His approach is distinguished from those considered previously in two important respects.

First, he focuses on proving properties of programs written in Standard ML using the formal semantics to be found in Milner, Tofte, and Harper (1990). This contrasts with the other approaches we have examined, where the final jump from an abstract function inside the logic to a concrete implementation in a serious programming language which *appears to correspond to it* is a glaring leap of faith.

Second, he points out that absolute program verification is not necessary. It suffices to show that some new piece of code behaves in the same way as another piece of code which is implemented in the normal LCF way as a composition of primitive inference rules. More precisely, if an ML function `f` is a 'safe' derived rule returning a theorem, and we can prove that for some function `g` which returns a term list-term pair, `dest_thm o f = g` then we may safely incorporate `mk_thm o g` as a new rule. Here `mk_thm` and `dest_thm` are the abstraction and representation functions which move between a concrete implementation of sequents and the abstract type `thm` of theorems.

How realistic is this proposal? The more modest requirement for proofs of program equivalence has considerable promise. It seems more suited to simple low-level proof procedures which have a straightforward but inefficient proof in terms of primitive inferences. The use of program equivalence is less appealing in instances where a reasonable proof by primitive inferences must deviate substantially from the 'standard' presentation. Nevertheless, as far as it goes, it may allow reasonably tractable correctness proofs without forcing the practitioner to worry about the low-level codification of the semantics.

It is conceivable that a useful set of equational transformations for reasoning about Standard ML programs could be derived from the operational semantics of Standard ML. On the other hand, the semantics of Standard ML is not trivial, and has certain defects from the point of view of correctness proofs. For example, it says nothing about the behaviour of arithmetic operations; in practice some implementations use machine arithmetic while others use infinite precision arithmetic. If one takes program verification seriously, these issues have to be addressed.

There has been some work in HOL on the formal semantics of ML by Syme (1993), VanInwegen and Gunter (1993) and Maharaj and Gunter (1994). But this is in its early stages, and more than just a formalized semantics is needed to make program verification tractable. One approach might be to isolate a sufficient core of ML for the implementation of HOL, and attempt to produce a formal semantics for it. The difficulty is that, if our arguments are correct, the appeal of reflection is greatest where the verification is of imperative code. This is known to be more difficult; see Mason and Talcott (1992) for some preliminary work in this direction.

Finally, it is interesting to observe how Slind dealt with the problem of inserting compiled code into an LCF-style implementation. His technique makes essential use of first-class environments, a relatively new feature of the New Jersey compiler. It may be that it holds interesting lessons for the incremental extension of abstract data types in general, not just in connection with LCF-style theorem proving.

Conclusion

Extensibility of theorem provers is an important issue, and if demands of rigour are to be taken seriously, the relative merits of reflection and the pure LCF approach should be analyzed carefully.

For many purposes, ostensibly metatheoretic reasoning can be implemented without any logical extensions. In any case, there are various subtly different notions described as 'reflection', and it is important to distinguish them. The most interesting from the point of theorem proving technology is computational reflection used to make inferences more efficient without unprincipled addition of new rules.

Programming a derived rule in LCF requires a certain discipline, and in complex cases, good programming skills are needed. There is certainly an argument that verifying a direct implementation is more natural. It's rather like coding an algorithm which appears naturally imperative in a pure functional language. Nevertheless there seems no convincing evidence that it is fundamentally inadequate from the efficiency point of view. Airy claims about the hopeless inefficiency of LCF-style provers on real examples have limited support in theory and are contradicted by practice (HOL is used!). We have looked at techniques which often render LCF proof procedures quite efficient. It has not clearly been established that there are any efficient proof procedures which cannot be implemented as a HOL derived rule with more than a moderately large constant factor slowdown. This constant factor may well be practically important, but it seldom separates tractability from intractability. As we have remarked, the efficiency of proof procedures is, within reason, not a major issue in interactive theorem proving.

Reflection is an intellectually attractive idea, in that it offers a way of adding efficient proof procedures while maintaining a guarantee of soundness. However the fact that despite all the research reports and proposals it has only once been used in a major practical prover, and even there not much in practice, speaks against it. Furthermore we have argued that the most interesting proof procedures such as Binary Decision Diagrams depend for their efficiency on imperative features such as arrays or shared data structures. Most real implementations are written in C, a difficult language to reason about formally. If reflection principles are to accommodate such programs, then the state of the art in program verification needs to advance, or the correctness proofs will be unbearably difficult. Then the difficulties of dynamically adding code to a running implementation need to be taken seriously. Finally, it seems hard to exploit reflection while at the same time generating a checkable low-level proof log.

There is considerable intellectual and practical benefit in sticking to the pure LCF approach in HOL, and the case against it is questionable. Whether HOL exhibits an unusual synergy in this regard is an interesting question, and our conclusions, even if correct for HOL, do not necessarily extend to other systems. Reflection offers many interesting ideas and challenges, but it isn't yet ready to push back the boundaries of what is feasible in theorem proving. Attempts to present it as a practical necessity and panacea for theorem proving in real world applications seem naive.

Acknowledgements

I am grateful to John Herbert and Roger Hale of SRI International for asking me to write this paper, and for valuable discussions. I have also profited from conversations with and advice from Richard Boulton, Thomas Forster, Mike Gordon, Doug Howe, Ken Kunen, Joseph Melia, Andy Pitts and especially Konrad Slind. The views expressed are my own responsibility, and should not be identified with any of the people named above, or with SRI. Apart from those already mentioned, comments on a draft version from Paul Curzon, Tim Leonard, Tom Melham, John Staples and especially Bob Boyer, have been very helpful and I hope have led to some improvements.

Appendix: Other kinds of reflection

In their interesting survey paper Giunchiglia and Smaill (1989) propose a distinction between a 'reflection principle' (strengthening the logic) and the process of 'reflection' (merely making the deductive process more efficient). This corresponds to our distinction between 'logical reflection' and 'computational reflection'. However there are at least two other uses of the term 'reflection' in the literature, and here we attempt to clarify them. We should add that there are a few instances which do not fit easily into the categorization we have chosen. An example is reflection in logic programming, as proposed by Bowen and Kowalski (1982) which is perhaps a blend of logical and procedural reflection. Perlis (1985) and Perlis (1988) discuss self-reference in first order logic, which bears some relation to logical reflection. A more mathematical treatment is given by Smoryński (1985), who also gives a readable account of Gödel's theorems and logical reflection schemas.

Set theoretic reflection

Zermelo-Fraenkel set theory and most other modern variants present the set theoretic universe as a cumulative hierarchy of sets[17]. This is built up from the empty set, and possibly a given collection of 'urelements', by iterating the powerset construction. The successive levels are usually written V_α where α runs through the ordinal numbers. The recursive definition splits into two cases, for successor and limit ordinals:

$$V_{\alpha+1} = \wp(V_\alpha)$$

$$V_\lambda = \bigcup_{\alpha<\lambda} V_\alpha$$

The complete 'universe' is usually written V; we can write:

$$V = \bigcup_\alpha V_\alpha$$

but we should be aware that V is not a set, and the above is really a figure of speech for the formal assertion:

$$\forall x.\ \exists \alpha.\ \mathit{Ordinal}(\alpha) \wedge x \in V_\alpha$$

All sets contained in V_ω, and hence in V_n for some finite ordinal n, are hereditarily finite (i.e. they are finite and all their members are in turn hereditarily finite). V_ω itself is the first infinite set. All Zermelo's axioms are satisfied if the 'universe' is the set $V_{\omega+\omega}$, i.e. all sets arise from applying the powerset operation finitely often to the empty set or the first infinite set.

Using the Axiom of Replacement (available in ZF but not in Zermelo's original system), we can show that there is actually a set $V_{\omega+\omega}$, as follows: by recursion and the Axiom of Replacement, we can construct a function whose range is $V_{\omega+\omega}$, and the Axiom of Union then allows us to collect the range in a new set. The iteration may similarly be continued to any transfinite ordinal. Consequently the hierarchy extends much further than it need do without Replacement.

A set-theoretic reflection principle asserts, roughly speaking, that some restricted initial portion of the ZF hierarchy (which will of course be a set occurring in the next level up) 'reflects' the structure of the whole universe. Crudely speaking, *any property true in V is also true in some V_α*. This is a vague statement and cannot be taken too literally. For example, 'every set is a member of V' is clearly false if V is replaced by any V_α. However a more consistent relativization, 'every set in V_α is a member of V_α', is true. Under this kind of interpretation, reasonable statements are obtained.

In particular, if we restrict ourselves to 'properties' expressible in an orthodox first order axiomatization of ZF set theory, the natural formalization of the reflection principle turns out to be provable. This was first shown by Montague (1966) and Lévy (1960); the latter coined the term 'reflection principle' and initiated a thorough study of such principles. Let us write ϕ^R for the 'relativization' of some formula ϕ to the set (or, with obvious change, class) R. This means restricting all the quantifiers in ϕ as follows: $\forall x.\ \psi[x]$ becomes $\forall x.\ x \in R \Rightarrow \psi[x]$ and $\exists x.\ \psi[x]$ becomes $\exists x.\ x \in R \wedge \psi[x]$. Now the reflection schema states that for any formula ϕ with free variables $x_1 \ldots x_n$:

$$\forall \alpha.\ \exists \beta.\ \beta > \alpha \wedge \forall x_1 \ldots x_n \in V_\beta.\ \phi \equiv \phi^{V_\beta}$$

[17]This now ubiquitous picture was only arrived at by Zermelo 20 years after his original axiomatization.

For a proof, see for example Kunen (1980) or Krivine (1971). In the special case where ϕ is a sentence then we see that there exist arbitrarily large ordinals β such that ϕ is interdeducible with its relativization to V_β. Among the most simple consequences we see that ZF cannot be finitely axiomatizable in first order logic. Indeed if a finite set of axioms S sufficed to axiomatize ZF, then we could form their conjunction ϕ. But now the reflection schema yields a model (more precisely, proves in ZF the existence of a model) for ϕ, and hence S, inside ZF. This amounts to proving ZF's consistency inside ZF, which we know to be impossible by Gödel's second theorem. (A more direct proof not relying on Gödel's theorem is also possible from the reflection principle.)

Sharpening these observations demonstrates the interesting connection with logical reflection principles[18]. If $Pr_N(\ulcorner\phi\urcorner)$ means 'ϕ is provable from the first N axioms of ZF', then it is not hard to see that the following partial reflection schema:

$$\vdash_{ZF} Pr_N(\ulcorner\phi\urcorner) \Rightarrow \phi$$

Indeed, we know that $\vdash_{ZF} \phi \vee \neg\phi$, and so $\vdash_{ZF} \phi \vee (ZF_1 \wedge \ldots \wedge ZF_N \wedge \neg\phi)$, where the ZF_i are the first N axioms of ZF. Abbreviating the second disjunct by ψ, we know by the reflection theorem that $\vdash_{ZF} \psi \equiv \psi^{V_\alpha}$ for some ordinal α. However, replicating the routine proof of first order logic's soundness inside ZF, $\vdash_{ZF} Pr_N(\ulcorner\phi\urcorner) \Rightarrow \neg\psi^{V_\alpha}$, and consequently $\vdash_{ZF} Pr_N(\ulcorner\phi\urcorner) \Rightarrow \phi$.

Another noteworthy consequence of the reflection principle, noted by Kreisel (1965), is that the introduction of universes, popular with category theorists interested in providing a set-theoretic foundation for their work, is probably unnecessary. Such axioms amount to asserting the reflection principle for *all* axioms of ZF together. However only finitely many of those axioms will be used in any given proof, and by the reflection theorem these already hold in some set V_α.

Apart from such applications, the deductive strength of this and related principles has been studied by Lévy and others. In the presence of the other ZF axioms, the first order reflection schema given above turns out to be equivalent to the axioms of Infinity and Replacement taken together. This raises the possibility that one might axiomatize set theory using reflection principles.

Extended reflection principles, for example the higher order versions introduced by Bernays (1966), turn out to be equivalent to rather recherché additions to the ZF axioms, such as inaccessible cardinals, regular fixed points for normal functions, and Mahlo cardinals. Hence they provide, for some, a persuasive way of motivating such additions. One can even eschew any kind of formalization and simply regard reflection principles as quasi-philosophical assertions about the endless, indefinitely extensible nature of the ZF hierarchy.

The proposal of Kreisel (1967) that the notions of semantic validity for second order logic when interpreted in a formal, cumulative set theory like ZF and in 'informal set theory' (permitting for example proper classes), might be coextensive, has similar consequences. Some formalizations of Kreisel's principle turn out to be equivalent to higher order reflection principles. See for example Shapiro (1991).

Procedural reflection

Computer programs are ultimately run as machine code which exists ephemerally in the memory of the machine. However for many purposes it is useful to be able to step back from this simple picture, contemplating (and perhaps changing) the relationship between the running program and the original source. Some obvious examples, starting with the routine and ending in the exotic, are:

[18] This connection goes right back to Montague's work; nevertheless the term 'reflection' seems to be based on different idioms in the two cases.

- Debugging — here it is desirable to relate the execution of the program to its original (source code) syntax, for the benefit of the user, and allow the user to step through the program, inspecting and altering the state of the machine at various points.
- Profiling — here we want to associate runtimes with function bodies in the original source (which may have no simple relationship with portions of the eventual machine code), in order to identify 'hot spots' in the code.
- Self-modification — it is commonplace in Artificial Intelligence to have programs modify themselves in the light of the interaction with their environment. For example, a chess program may alter its play on the basis of past experience.

Implementing such facilities in an ad hoc way is sometimes quite involved, and it is not easy for ordinary users to add their own related facilities. It was argued by Smith (1984) that a general reflective programming language offers a uniform and flexible way of doing such things. He compares reflection with recursion. At first it seems a complicated, arcane and inefficient way of programming, in danger of infinite regress. But with experience it may come to be seen as natural, and hence tend to be implemented more efficiently. It promises to provide a highly flexible facility which may then be used to implement otherwise inexpressible programming constructs (such as adding exception-handling to the language). A more developed treatment can be found in des Rivières and Smith (1984).

Smith's approach is to start with a dialect of LISP, called 2-LISP, which is LISP shorn of the use-mention confusions which Smith detects in the mainstream version. Smith argues that LISP 'crosses semantic levels', confusing the notionally separate processes of passing from syntax to denotation and evaluating the denotation. For example, `(+ 1 '2)` is acceptable in all LISP and SCHEME implementations.

Programs are run via an interpreter whose code and data is made concrete in explicit datastructures. This results in 3-LISP, which has the ability to run code at different levels of interpretation. Instead of running a program 'at level n' the concrete representation can be run by an interpreter 'at level $n + 1$'. This interpreter may itself be run by another interpreter 'at level $n + 2$' and so on ad infinitum. The crucial points are that first, all properties of the program and its interpreter are made concrete, and secondly, that those concrete versions may be modified, affecting the program's behaviour.

In LISP implementations, this already happens to a limited extent when procedure calls and `EVAL` are supported uniformly for compiled code and interpreted S-expressions. Even running a BASIC interpreter on a microcoded CPU exhibits a multiplicity of levels. The reflective approach is distinguished by its unlimited scope and the homogeneity of the successive levels. It actually gives rise to a potential infinity of levels, the so-called 'reflective tower'. Smith's idea was that a program should find the lowest level available (since layers of interpretation are inefficient), only rising to higher levels when necessary. As far as practical implementations go, experience has been limited, but it may be that a less comprehensive form of reflection than that proposed by Smith, where the concrete representations are separate from the code executed and perhaps only partial (this is sometimes called 'declarative reflection'), is acceptably efficient.

Smith did not give a rigorous discussion of the intended semantics of the reflective tower. This was later undertaken by Wand and Friedman (1986), who gave a denotational description, using an additional 'metacontinuation' parameter to store the state of the interpreters above the one currently being considered. They also use a slightly refined terminology: 'reification' is the process by which an interpreter makes its state available, and 'reflection' is when the program changes that state and hence installs new data.

Since it was originally proposed, procedural reflection has attracted considerable attention in the object oriented programming community, initiated explicitly by Maes (1987). The approach fits nicely with the object philosophy, where the programmer's whole 'world' is supposed to be open to redefinition, yet some facilities, e.g. communication between processes in sophisticated ways, may be hard to implement in existing systems. A survey is given by Maes and Nardi (1988). Nevertheless it is not clear that reflection's practical utility has yet been convincingly demonstrated. For work on procedural reflection in a high-level functional programming language, see the paper by Zhu (1994).

References

Aagaard, M. and Leeser, M. (1994) Verifying a logic synthesis tool in Nuprl: A case study in software verification. In v. Bochmann, G. and Probst, D. K. (eds.), *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, Number 663 in Lecture Notes in Computer Science, Montreal, Canada, pp. 69–81. Springer Verlag.

Aiello, L., Cecchi, C., and Sartini, D. (1986) Representation and use of metaknowledge. *Proceedings of the IEEE*, **74**, 1304–1321.

Aiello, L. and Weyhrauch, R. W. (1980) Using meta-theoretic reasoning to do algebra. In Bibel, W. and Kowalski, R. (eds.), *5th Conference on Automated Deduction*, Volume 87 of *Lecture Notes in Computer Science*, Les Arcs, France, pp. 1–13. Springer-Verlag.

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

Ankeny, N. C. (1952) The least quadratic non-residue. *Annals of Mathematics (2)*, **55**, 65–72.

Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J. (eds.) (1991) *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, University of California at Davis, Davis CA, USA. IEEE Computer Society Press.

Armando, A., Cimatti, A., and Viganò, L. (1993) Building and executing proof strategies in a formal metatheory. In Torasso, P. (ed.), *Advances in Artifical Intelligence: Proceedings of the Third Congress of the Italian Association for Artificial Intelligence, IA*AI'93*, Volume 728 of *Lecture Notes in Computer Science*, Torino, Italy, pp. 11–22. Springer-Verlag.

Arthan, R. D. (1994) Issues in implementing a high integrity proof tool. See Basin, Giunchiglia, and Kaufmann (1994), pp. 34–36.

Bach, E. (1990) Explicit bounds for primality testing and related problems. *Mathematics of Computation*, **55**, 355–380.

Baker, A. (1975) *Transcendental Number Theory*. Cambridge University Press.

Bar-Hillel, Y., Poznanski, E. I. J., Rabin, M. O., and Robinson, A. (eds.) (1966) *Essays on the Foundations of Mathematics: dedicated to A. A. Fraenkel on his seventieth anniversary*. The Magnes Press, the Hebrew University, Jerusalem. First edition (1966) published by the Jerusalem Academic Press Ltd.

Barwise, J. and Keisler, H. (eds.) (1991) *Handbook of mathematical logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Basin, D., Giunchiglia, F., and Kaufmann, M. (eds.) (1994) *12th International Conference on Automated Deduction, Workshop 1A: Correctness and metatheoretic extensibility of automated reasoning systems*, INRIA Lorraine.

Basin, D., Giunchiglia, F., and Traverso, P. (1991) Automating meta-theory creation and system extension. In *Trends in Artificial Intelligence: Proceedings of the 2nd Congress of the Italian Association for Artificial Intelligence, IA*AI*, Number 549 in Lecture Notes in Computer Science, pp. 48–57. Springer-Verlag.

Basin, D. A. (1994) Constructive metatheoretic extensibility. See Basin, Giunchiglia, and Kaufmann (1994), pp. 23–24.

Basin, D. A. and Constable, R. (1991) Metalogical frameworks. See Huet, Plotkin, and Jones (1991), pp. 47–72. Reprinted in Huet and Plotkin (1993), pp. 1–29.

Bernays, P. (1966) Zur Frage der Unendlichkeitsschemata in der axiomatischen Mengenlehre. See Bar-Hillel, Poznanski, Rabin, and Robinson (1966), pp. 3–49. English translation 'On the Problem of Schemata of Infinity in Axiomatic Set Theory' in Müller (1976), pp. 121–172.

Bjørner, D., Ershov, A. P., and Jones, N. D. (eds.) (1988) *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, Gammel Avernæs, Denmark. North-Holland.

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

Bowen, K. A. and Kowalski, R. A. (1982) Amalgamating language and metalanguage in logic programming. In Clark, K. L. and Tärnlund, S. A. (eds.), *Logic Programming*, Number 16 in APIC Studies in Data Processing, pp. 153–172. Academic Press.

Boyer, R. S. and Moore, J S. (1979) *A Computational Logic*. ACM Monograph Series. Academic Press.

Boyer, R. S. and Moore, J S. (1981) Metafunctions: proving them correct and using them efficiently as new proof procedures. In Boyer, R. S. and Moore, J S. (eds.), *The Correctness Problem in Computer Science*, pp. 103–184. Academic Press.

Boyer, R. S. and Yu, Y. (1992) Automating correctness proofs of machine code programs for a commercial microprocessor. See Kapur (1992), pp. 416–430.

de Bruijn, N. G. (1980) A survey of the project AUTOMATH. In Seldin, J. P. and Hindley, J. R. (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pp. 589–606. Academic Press.

Bryant, R. E. (1992) Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, **24**, 293–318.

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

Buss, S. R. (1994) On Gödel's theorems on lengths of proofs I: Number of lines and speedup for arithmetics. *Journal of Symbolic Logic*, **59**, 737–756.

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

Cherlin, G. L. (1976) Model theoretic algebra. *Journal of Symbolic Logic*, **41**, 537–545.

Chou, S.-C. (1988) An introduction to Wu's method for mechanical theorem proving in geometry. *Journal of Automated Reasoning*, **4**, 237–267.

Claesen, L. J. M. and Gordon, M. J. C. (eds.) (1992) *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Volume A-20 of *IFIP Transactions A: Computer Science and Technology*, IMEC, Leuven, Belgium. North-Holland.

Clote, P. and Krajíček, J. (eds.) (1993) *Arithmetic, proof theory, and computational complexity*. Clarendon Press.

Constable, R. (1986) *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall.

Constable, R. L. and Murthy, C. (1990) Extracting computational content from classical proofs. In Huet, G. and Plotkin, G. (eds.), *Proceedings of the First Workshop on Logical Frameworks*, pp. 141–156. Reprinted in Huet and Plotkin (1991), pp. 341–362.

Danvy, O. (1988) Across the bridge between reflection and partial evaluation. See Bjørner, Ershov, and Jones (1988), pp. 83–116.

Davis, M. (ed.) (1965) *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Raven Press, NY.

Davis, M. and Schwartz, J. T. (1979) Metatheoretic extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, **5**, 217–230.

Dudley, R. M. (1988) *Real analysis and probability*. The Wadsworth & Brooks/Cole mathematics series. Wadsworth & Brooks/Cole.

Edwards, H. M. (1989) Kronecker's views on the foundations of mathematics. In Rowe, D. E. and McCleary, J. (eds.), *The History of Modern Mathematics; Volume 1: Ideas and Their Reception*, pp. 67–77. Academic Press.

Eklof, P. (1973) Lefschetz' principle and local functors. *Proceedings of the AMS*, **37**, 333–339.

Feferman, S. (1960) Arithmetization of metamathematics in a general setting. *Fundamenta Mathematicae*, **49**, 35–92.

Feferman, S. (1962) Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, **27**, 259–316.

Feferman, S. (1989) Finitary inductively presented logics. In Ferro, R. et al. (eds.), *Logic Colloquium 88*, Studies in Logic and the Foundations of Mathematics, Padova, Italy, pp. 191–220. North-Holland.

Feferman, S. (1991) Reflecting on incompleteness. *Journal of Symbolic Logic*, **56**, 1–49.

Felty, A. and Miller, D. (1988) Specifying theorem provers in a higher-order logic programming language. See Lusk and Overbeek (1988), pp. 61–80.

Gentzen, G. (1935) Über das logische Schliessen. *Mathematische Zeitschrift*, **39**, 176–210. This was Gentzen's Inaugural Dissertation at Göttingen. English translation, 'Investigations into Logical Deduction', in Szabo (1969), p. 68–131.

Giunchiglia, F., Armando, A., Cimatti, A., and Traverso, P. (1994) First steps towards provably correct system synthesis of system code. See Basin, Giunchiglia, and Kaufmann (1994), pp. 28–30.

Giunchiglia, F. and Smaill, A. (1989) Reflection in constructive and non-constructive automated reasoning. In Abramson, H. and Rogers, M. H. (eds.), *Meta-Programming in Logic Programming*, pp. 123–140. MIT Press.

Gödel, K. (1931) Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, **38**, 173–198. English translation, 'On Formally Undecidable Propositions of Principia Mathematica and Related Systems, I', in van Heijenoort (1967), pp. 592–618 or Davis (1965), pp. 4–38.

Gödel, K. (1936) Über die Länge von Beweisen. *Ergebnisse eines mathematischen Kolloquiums*, **7**, 23–24. English translation, 'On The Length of Proofs', in Davis (1965), pp. 82–83.

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

Gordon, M. J. C. (1982) Representing a logic in the LCF metalanguage. In Neel, D. (ed.), *Tools and notions for program construction: an advanced course*, pp. 163–185. Cambridge University Press.

Gordon, M. J. C., Hale, R., Herbert, J., von Wright, J., and Wong, W. (1994) Proof checking for the HOL system. See Basin, Giunchiglia, and Kaufmann (1994), pp. 49–50.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanized Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag.

Harper, R., Honsell, F., and Plotkin, G. (1987) A framework for defining logics. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, Ithaca, NY, pp. 194–204. IEEE Computer Society Press.

Harrison, J. (1993) A HOL decision procedure for elementary real algebra. See Joyce and Seger (1993a), pp. 426–436.

Harrison, J. (1995) Binary decision diagrams as a HOL derived rule. *The Computer Journal*, **38**. To appear.

Harrison, J. and Théry, L. (1993) Extending the HOL theorem prover with a computer algebra system to reason about the reals. See Joyce and Seger (1993a), pp. 174–184.

van Heijenoort, J. (ed.) (1967) *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931.* Harvard University Press.

Hoffmann, C. M. and O'Donnell, M. J. (1982) Pattern matching in trees. *Journal of the ACM*, **29**, 68–95.

Hoover, D. N. and McCullough, D. (1992) Verifying launch interceptor routines with the asymptotic method. ORA internal report.

Howe, D. J. (1988) Computational metatheory in Nuprl. See Lusk and Overbeek (1988), pp. 238–257.

Howe, D. J. (1992) Reflecting the semantics of reflected proof. In Aczel, P., Simmons, H., and Wainer, S. (eds.), *Proof Theory*, pp. 229–250. Cambridge University Press.

Huet, G. (1975) A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, **1**, 27–57.

Huet, G. and Plotkin, G. (eds.) (1991) *Logical Frameworks*. Cambridge University Press.

Huet, G. and Plotkin, G. (eds.) (1993) *Logical Environments*. Cambridge University Press.

Huet, G., Plotkin, G., and Jones, C. (eds.) (1991) *Proceedings of the Second Workshop on Logical Frameworks*. Available by FTP from `ftp.dcs.ed.ac.uk` as `export/bra/proc91.dvi.Z`.

Joyce, J. J. and Seger, C. (eds.) (1993a) *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. Springer-Verlag.

Joyce, J. J. and Seger, C. (1993b) The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. See Joyce and Seger (1993a), pp. 185–198.

Jutting, L. S. van Bentham (1977) *Checking Landau's "Grundlagen" in the AUTOMATH System*. Ph. D. thesis, Eindhoven University of Technology.

Kapur, D. (ed.) (1992) *11th International Conference on Automated Deduction*, Volume 607 of *Lecture Notes in Computer Science*, Saratoga, NY. Springer-Verlag.

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

Knuth, D. E. (1973) *The Art of Computer Programming; Volume 3: Sorting and Searching*. Addison-Wesley Series in Computer Science and Information processing. Addison-Wesley.

Kreisel, G. (1952) On the interpretation of non-finitist proofs — part II; interpretation of number theory; applications. *The Journal of Symbolic Logic*, **17**, 43–58.

Kreisel, G. (1956) Some uses of metamathematics. *British Journal for the Philosophy of Science*, **7**, 161–173.

Kreisel, G. (1965) Mathematical logic. In Saaty, T. L. (ed.), *Lectures on Modern Mathematics, vol. III*, pp. 95–195. Wiley.

Kreisel, G. (1967) Informal rigour and completeness proofs. In Lakatos, I. (ed.), *Problems in the Philosophy of Mathematics: Proceedings of the International Colloquium in the Philosophy of Science*, Bedford College, Regent's Park, London, pp. 138–171. North-Holland. See also the following discussion, pp. 172–186.

Kreisel, G. and Krivine, J.-L. (1971) *Elements of mathematical logic: model theory* (Revised second ed.). Studies in Logic and the Foundations of Mathematics. North-Holland. First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles'.

Kreisel, G. and Lévy, A. (1968) Reflection principles and their use for establishing the complexity of axiomatic systems. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, **14**, 97–142.

Krivine, J.-L. (1971) *Introduction to Axiomatic Set Theory.* Synthese Library. D. Reidel Publishing Company. Translation of the French 'Théorie Axiomatique des Ensembles', first published by Presses Universitaires de France, Paris. Translated by David Miller.

Kromodimoeljo, S. and Pase, W. (1994) Proof logging and proof checking in NEVER. See Basin, Giunchiglia, and Kaufmann (1994), pp. 41.

Kumar, R., Kropf, T., and Schneider, K. (1991) Integrating a first-order automatic prover in the HOL environment. See Archer, Joyce, Levitt, and Windley (1991), pp. 170–176.

Kunen, K. (1980) *Set Theory: An Introduction to Independence Proofs*, Volume 102 of *Studies in Logic and the Foundations of Mathematics.* North-Holland.

Landau, E. (1966) *Foundations of analysis: the arithmetic of whole, rational, irrational, and complex numbers. A supplement to textbooks on the differential and integral calculus* (Third ed.). Chelsea Publishing Company. Translated from German 'Grundlagen der Analysis' by F. Steinhardt.

Leśniewski, S. (1929) Grunzüge eines neuen Systems der Grundlagen der Mathematik. *Fundamenta Mathematicae*, **14**, 1–81. English translation, 'Fundamentals of a new system of the foundations of mathematics' in Surma, Srzednicki, Barnett, and Rickey (1992), vol. II, pp. 410–605.

Lévy, A. (1960) Principles of reflection in axiomatic set theory. *Fundamenta Mathematicae*, **49**, 1–10.

Löb, M. H. (1955) Solution of a problem of Leon Henkin. *Journal of Symbolic Logic*, **20**, 115–118.

Lusk, E. and Overbeek, R. (eds.) (1988) *9th International Conference on Automated Deduction*, Volume 310 of *Lecture Notes in Computer Science*, Argonne, Illinois, USA. Springer-Verlag.

Maes, P. (1987) Concepts and experiments in computational reflection. In Meyrowitz, N. (ed.), *Object-Oriented Programming Systems, Languages and Applications: Proceedings of OOPSLA'87*, Orlando, Florida, pp. 147–155. Association for Computing Machinery. Special issue of SIGPLAN Notices, vol. 22, number 12.

Maes, P. and Nardi, D. (eds.) (1988) *Meta-Level Architectures and Reflection.* North-Holland.

Maharaj, S. and Gunter, E. (1994) Studying the ML module system in HOL. See Melham and Camilleri (1994), pp. 346–361.

Martin-Löf, P. (1985) Constructive mathematics and computer programming. In Hoare, C. A. R. and Shepherdson, J. C. (eds.), *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science, pp. 167–184. Prentice-Hall.

Mason, I. A. and Talcott, C. L. (1992) References, local variables and operational reasoning. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, CA, USA, pp. 186–197. IEEE Computer Society Press.

Matthews, S. (1994) A theory and its metatheory in FS_0. Publication unknown.

Matthews, S., Smaill, A., and Basin, D. (1991) Experience with FS_0 as a framework theory. See Huet, Plotkin, and Jones (1991), pp. 231–252. Reprinted in Huet and Plotkin (1993), pp. 61–82.

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

Melham, T. F. (1993) *Higher Order Logic and Hardware Verification*, Volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press. A revision of the author's PhD thesis.

Melham, T. F. and Camilleri, J. (eds.) (1994) *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, Volume 859 of *Lecture Notes in Computer Science*, Valletta, Malta. Springer-Verlag.

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

Milner, R. and Tofte, M. (1991) *Commentary on Standard ML*. The MIT Press.

Milner, R., Tofte, M., and Harper, R. (1990) *The Definition of Standard ML*. The MIT Press.

MOD, U. K. (1991) The procurement of safety critical software in defence equipment. Interim Defence Standard 00-55, UK Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, GLASGOW G2 8EX, UK.

Montague, R. (1966) Fraenkel's addition to the axioms of Zermelo. See Bar-Hillel, Poznanski, Rabin, and Robinson (1966), pp. 91–114. First edition (1966) published by the Jerusalem Academic Press Ltd.

Moore, J (1994) Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, **12**, 33–45.

Müller, G. H. (ed.) (1976) *Sets and Classes: on the work by Paul Bernays*, Number 84 in Studies in Logic and the Foundations of Mathematics. North-Holland.

Owre, S., Rushby, J. M., and Shankar, N. (1992) PVS: A prototype verification system. See Kapur (1992), pp. 748–752.

Paris, J. and Harrington, L. (1991) A mathematical incompleteness in Peano Arithmetic. See Barwise and Keisler (1991), pp. 1133–1142.

Paulson, L. C. (1987) *Logic and computation: interactive proof with Cambridge LCF*. Number 2 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Paulson, L. C. (1994) *Isabelle: a generic theorem prover*, Volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag. With contributions by Tobias Nipkow.

Perlis, D. (1985) Languages with self-reference I: Foundations. *Artificial Intelligence*, **25**, 301–322.

Perlis, D. (1988) Languages with self-reference II: Knowledge, belief, and modality. *Artificial Intelligence*, **34**, 179–212.

Pottinger, G. (1992) Completeness for the HOL logic: Preliminary report. Posted to `info-hol` mailing list on 28th Jan 1992. Available in the `info-hol` archive by anonymous FTP from `ftp.cl.cam.ac.uk` in directory `hvg/info-hol-archive`.

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

Reif, W. and Schönegge, A. (1994) A reflection mechanism in KIV using structured specifications. See Basin, Giunchiglia, and Kaufmann (1994), pp. 19–21.

Resnik, M. D. (1974) On the philosophical significance of consistency proofs. *Journal of Philosophical Logic*, **3**, 133–147. Reprinted in Shanker (1988), pp. 115–130.

des Rivières, J. and Smith, B. C. (1984) The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 331–347. Association for Computing Machinery.

Robinson, A. (1963) *Introduction to model theory and to the metamathematics of algebra*. Studies in Logic and the Foundations of Mathematics. North-Holland.

Rosser, J. B. (1936) Extensions of some theorems of Gödel and Church. *Journal of Symbolic Logic*, **1**, 87–91.

Rudnicki, P. (1992) An overview of the MIZAR project. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

Rushby, J. (1991) Design choices in specification languages and verification systems. See Archer, Joyce, Levitt, and Windley (1991), pp. 194–204.

Schneider, K., Kumar, R., and Kropf, T. (1992) Efficient representation and computation of tableaux proofs. See Claesen and Gordon (1992), pp. 39–57.

Seidenberg, A. (1954) A new decision method for elementary algebra. *Annals of Mathematics*, **60**, 365–374.

Shankar, N. (1994) *Metamathematics, Machines and Gödel's Proof*, Volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Shanker, S. G. (ed.) (1988) *Gödel's Theorem in Focus*, Philosophers in Focus series. Croom Helm.

Shapiro, S. (1991) *Foundations without Foundationalism: a case for second-order logic*. Number 17 in Oxford Logic Guides. Clarendon Press.

Slind, K. (1991) 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. Author's Masters thesis.

Slind, K. (1992) Adding new rules to an LCF-style logic implementation. See Claesen and Gordon (1992), pp. 549–559.

Smith, B. C. (1984) Reflection and semantics in LISP. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages*, pp. 23–35. Association for Computing Machinery.

Smoryński, C. (1977) ω-consistency and reflection. In *Colloque International de Logique*, Volume 249 of *Colloques Internationaux*, Clermont-Ferrand, pp. 167–181. Éditions du Centre National de la Recherche Scientifique.

Smoryński, C. (1985) *Self-Reference and Modal Logic*. Springer-Verlag.

Smoryński, C. (1991) The incompleteness theorems. See Barwise and Keisler (1991), pp. 821–865.

Surma, S. J., Srzednicki, J. T., Barnett, D. I., and Rickey, V. F. (eds.) (1992) *Stanisław Leśniewski: Collected Works*. Kluwer Academic Publishers.

Syme, D. (1993) Reasoning with the formal definition of Standard ML in HOL. See Joyce and Seger (1993a), pp. 43–60.

Szabo, M. E. (ed.) (1969) *The collected papers of Gerhard Gentzen*, Studies in Logic and the Foundations of Mathematics. North-Holland.

Takeuti, G. (1978) *Two applications of logic to mathematics*. Number 13 in Publications of the Mathematical Society of Japan. Iwanami Shoten, Tokyo. Number 3 in Kano memorial lectures.

Talcott, C. and Weyhrauch, R. (1988) Partial evaluation, higher-order abstractions, and reflection principles as system building tools. See Bjørner, Ershov, and Jones (1988), pp. 507–529.

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

Tarski, A. (ed.) (1956) *Logic, Semantics and Metamathematics*. Clarendon Press.

Taylor, P. (1988) Using Constructions as a metalanguage. LFCS Report Series ECS-LFCS-88-70, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, UK.

Troelstra, A. S. (1973) *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Number 344 in Lecture Notes in Mathematics. Springer-Verlag. Second, corrected edition available, as ILCC Prepublication Series number X-93-05, from the University of Amsterdam.

Turing, A. M. (1939) Systems of logic based on ordinals. *Proceedings of the London Mathematical Society (2)*, **45**, 161–228. Reprinted in Davis (1965), pp. 154–222.

VanInwegen, M. and Gunter, E. (1993) HOL-ML. See Joyce and Seger (1993a), pp. 61–74.

Wand, M. and Friedman, D. P. (1986) The mystery of the tower revealed: A nonreflective description of the reflective tower. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pp. 298–307. Association for Computing Machinery.

Weil, A. (1946) *Foundations of algebraic geometry*, Volume 29 of *AMS Colloquium Publications*. American Mathematical Society. Revised edition 1962.

Welinder, M. (1994) Towards efficient conversions by the use of partial evaluation. Presented in poster session of 1994 HOL Users Meeting and only published in participants' supplementary proceedings. Available on the Web from `http://www.dcs.glasgow.ac.uk/~hug94/sproc.html`.

Weyhrauch, R. W. (1980) Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, **13**, 133–170.

Weyhrauch, R. W. (1982) An example of FOL using metatheory. In Loveland, D. W. (ed.), *Proceedings of the 6th Conference on Automated Deduction*, Number 138 in Lecture Notes in Computer Science, New York, pp. 151–158. Springer Verlag.

Weyhrauch, R. W. and Talcott, C. (1994) The logic of FOL systems: Formulated in set theory. In Jones, N. D., Hagiya, M., and Sato, M. (eds.), *Logic, Language and Computation: Festschrift in Honor of Satoru Takasu*, Number 792 in Lecture Notes in Computer Science, pp. 119–132. Springer Verlag.

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

von Wright, J. (1994) Representing higher-order logic proofs in HOL. See Melham and Camilleri (1994), pp. 456–470.

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

Zhu, M.-Y. (1994) Computational reflection in PowerEpsilon. *ACM SIGPLAN Notices*, **29**(1), 13–19.