A Sceptic's Approach to Combining HOL and Maple

J. Harrison and L. Théry
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street, Cambridge CB2 3QG, England

INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93, 06902 Sophia Antipolis Cedex, France

19th August 1997

Abstract. We contrast theorem provers and computer algebra systems, pointing out the advantages and disadvantages of each, and suggest a simple way to achieve a synthesis of some of the best features of both. Our method is based on the systematic separation of search for a solution and checking the solution, using a physical connection between systems. We describe the separation of proof search and checking in some detail, relating it to proof planning and to the complexity class NP, and discuss different ways of exploiting a physical link between systems. Finally, the method is illustrated by some concrete examples of computer algebra results proved formally in the HOL theorem prover with the aid of Maple.

Key words: Proof checking, automated theorem proving, computer algebra, complexity theory

JEL codes: ?

1. Theorem provers vs. computer algebra systems

Computer algebra systems (CASs) seem superficially similar to computer theorem provers: both are computer programs for helping people with formal symbolic manipulations. However in practice there is surprisingly little common ground between them, either as regards the internal workings of the systems themselves or their respective communities of implementors and users. CASs are used by (mostly applied) mathematicians, scientists and engineers, typically to perform multiprecision arithmetic, operations on polynomials (usually over $\mathbb{R}$) and classical 'continuous' mathematics such as differentiation, integration and series expansion. By contrast, theorem provers are mainly used by computer scientists interested in systems verification, or by logically-inclined mathematicians interested in formalization of mathematics or experimenting with new logics.

CASs are much more popular than theorem provers. The most obvious reason is that there are more people in their natural user communities as described above; in general, formal logic is a more specialized interest than differentiation. However there are other reasons too for the greater popularity of CASs. They tend to be easier to use, so much so that they are increasingly applied in education (though this is controversial). They also usually work faster since they are optimized for dealing efficiently with high-level mathematical problems. However theorem provers offer the following advantages: they are more expressive, they are more precise, and they are more reliable.

As remarked by Corless and Jeffrey [14], the typical computer algebra system supports a rather limited style of interaction. The user types in an expression E; the CAS cogitates, usually not for very long, before returning another expression E'. (If E and E' are identical, that usually means that the CAS was unable to do anything useful. Unfortunately, as we shall see, the converse does not always hold!) The implication is that we should accept the theorem $\vdash E = E'$. Occasionally some slightly more sophisticated data may be returned, e.g. a condition on the validity of the equation, or even a set of possible expressions $E'_1, \ldots, E'_n$ with corresponding conditions on validity, e.g.

$$\sqrt{x^2} = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x \leq 0 \end{cases}$$

However, the simple equational style of interaction is by far the most usual. Certainly, CASs are almost never capable of expressing really sophisticated logical dependencies as theorem provers are.

When we say that CASs are imprecise, we mean that their mathematical semantics is not always clear. For example, the polynomial expression $x^2 + 2x + 1$ can be read in several ways: as a member of the polynomial ring $\mathbb{R}[x]$ (not to mention $\mathbb{Z}[x]$ or $\mathbb{C}[x]$ ...), as the associated function $\mathbb{R} \to \mathbb{R}$, or as the value of that expression for some particular $x \in \mathbb{R}$. Such ambiguities are particularly insidious since in many situations it doesn't matter which interpretation is chosen. We have $x^2 + 2x + 1 = (x + 1)^2$ for any of them. However if we want to talk about irreducibility of a polynomial, we must be dealing with the first interpretation, while an equation:

$$(x^2 - 1)/(x - 1) = x + 1$$

is only strictly valid if we are considering the polynomials as elements of the field of rational functions $\mathbb{R}(x)$; if they are interpreted as the associated function or value this is false for $x = 1$.

At least, it is false if we interpret $s = t$ as 'both s and t are defined and equal' or 'either both s and t are undefined or both are defined and equal', but it is true on an interpretation 'wherever s and t are both defined, they are equal'. These ways of interpreting the equality sign all seem to appear in certain places if one reads the mathematical literature critically. Actually, Freyd and Scedrov [17] use a special asymmetric 'Venturi tube' equality meaning 'if the left is defined then so is the right and they are equal'. So there is yet another ambiguity here.

Even when a CAS can be relied upon to give a result that admits a precise mathematical interpretation, that doesn't mean that its answers are always right. For example, a recent version of Maple evaluates:

$$\int_{-1}^{1} \sqrt{x^2}\, dx = 0$$

What seems to happen is that the simplification $\sqrt{x^2} = x$ is applied, regardless of the sign of x. In general, CASs tend to perform simplifications fairly aggressively, even if they aren't strictly correct in all circumstances. The policy is to try always to do *something*, even if it isn't absolutely sound. After all, it often happens that ignoring a few singularities in the middle of a calculation does make no difference to the result. This policy may also be a consequence of the limited equational style of interaction that we have already drawn attention to. If the CAS has only two alternatives, to do nothing or to return an equation which is true only with a few provisos, it might be felt that it's better to do the latter. The two concerns of imprecision and incorrectness are in any case not easy to separate. For example, if a CAS claims that $\frac{d}{dx}(1/x) = -1/x^2$, is it assuming an interpretation of equality 'either both sides are undefined or both are defined and equal'? Or is it simply making a mistake and forgetting the condition $x \neq 0$? Very often this is not clear.

While computer algebra systems concentrate on efficient computation, theorem provers stress validity. We shall see below that this is particularly true of HOL and other members of the LCF family. Designers of such theorem provers try hard to ensure that they do not make wrong inferences, even if this leads to its being hard to make their systems do anything useful at all.

Interaction with theorem provers may vary. In automatic theorem proving, the user proposes challenges that the prover *automatically* tries to prove. In interactive theorem proving, the user not only proposes the challenge but gives some (if not all) reasoning steps that lead to the proof. Recently both communities seem to be converging to a middle ground. For example, a *proof planning* component has been added to automatic theorem provers in order to make user and prover cooperate in the discovery of the proof, while more and more automation and decision procedures are being added to mechanized theorem provers. Generally, interaction with theorem provers is more intricate than interaction with computation algebra systems. Rather than lines of computation the user often has to deal with trees where each leaf contains a formula to prove in some specific context.

Theorem provers are good at expressing and manipulating properties precisely. For example the property

$$\sqrt{x^2} = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x \leq 0 \end{cases}$$

can be represented by the theorem:

$$\forall x \in \mathbb{R}.\ (x \geq 0 \Rightarrow \sqrt{x^2} = x) \wedge (x \leq 0 \Rightarrow \sqrt{x^2} = -x)$$

It is then ensured that any attempt to rewrite an expression $\sqrt{y^2}$ for some y with this theorem leads to the discharge of one of the two conditions.

2. LCF and HOL

Theorem provers descended from Edinburgh LCF [20] insist on reducing all reasoning to formal proofs in something like a standard natural deduction system. However, the user is able to write arbitrary programs in the meta-language ML to automate patterns of inferences and hence reduce the tedium involved. The original LCF system implemented Scott's Logic of Computable Functions (hence the name LCF), but as emphasized by Gordon [18], the basic LCF approach is applicable to any logic, and now there are descendants implementing a variety of higher order logics, set theories and constructive type theories.

HOL [19] implements a version of simply typed λ-calculus with logical operations defined on top, more or less following Church [10]. It takes the LCF approach a step further in that all theory developments are pursued 'definitionally': new mathematical structures may be defined only by exhibiting a model for them in the existing theories, and new constants may only be introduced by definitional extension (roughly speaking, merely being a shorthand for an expression in the existing theory). This fits naturally with the LCF style, since it ensures that all extensions, whether of the deductive system or the mathematical theories, are consistent per construction. The former holds because theorems may *only* be created by composing the basic logical rules. This is enforced by encoding theorems as an ML abstract type whose only constructors are these primitive rules. HOL has about a dozen such rules, and they are all very simple.

The LCF methodology thus gives an outstanding combination of programmability and reliability. Assuming that the primitive rules are correct,[1] then none of the derived rules built on top can ever give a false 'theorem', and none of the derived theories can be inconsistent. And there is no shortage of such extensions: HOL includes derived rules for rewriting, associative-commutative rearrangement, linear arithmetic, tautology checking, inductive definitions and free recursive type definitions, among others, while the mathematical theories include numbers of various kinds, sets, lists and others. And should a user want another rule or theory, they can write it themselves using the same methodology.

[1] And of course that the implementation environment works correctly, but that's always a necessary assumption or we descend into infinite regress.

While HOL was originally intended for hardware verification applications, it has been used for many other things too. In particular there is a definitional construction of the real numbers and associated theories of elementary real analysis, including sequences, series, limits, continuity, differentiation and integration, as well as the properties of common transcendental functions such as sin and log [21]. The integration theory we formalized, the gauge integral [28, 24], obeys the Fundamental Theorem of Calculus without additional differentiability assumptions; it also has the same attractive limit behaviour as the Lebesgue integral, which it properly includes. This analysis development has helped to eliminate the bias towards 'discrete' mathematics typical in theorem proving systems, and so it seems a good platform from which to explore the relationship with computer algebra. We haven't developed complex analysis, algebraic numbers, multivariate calculus, matrices, and many other topics. But we have at least a foundation for further efforts, and because of the LCF approach, we can feel more confident that if we prove computer algebra results in HOL, they are both more precise and more reliable.

The LCF approach would seem at first sight to be hopelessly restrictive for programming really advanced derived rules such as those found in computer algebra. When one thinks of the wide variety of special purpose decision procedures available, it seems implausible that they can all be modified to perform a breakdown to standard logical primitives, let alone efficiently. Perhaps surprisingly, it often is practical — for a detailed analysis and a comparison with other techniques such as 'reflection', see [22]. One reason is that inference patterns may be represented as general (object) theorems, which need only be proved once, and can thereafter be used with just a little instantiation. Algorithms can often be implemented in more or less the standard style, with each step justified by a theorem. Another reason is that the processes of proof discovery and proof checking can be quite different, which brings us back to the topic of this paper.

Even under the LCF strictures, a result can be arrived at it any way whatsoever provided that it is *checked* by a rigorous reduction to primitive inferences. This idea has already been used for example in work on first order automation [27] and solving linear inequalities [7] in HOL. Now many computer algebra systems include complicated algorithms and heuristics for various tasks. Implementing these directly in the LCF style would be a major undertaking, and it seems unlikely that the results would be anything like as efficient. However one often sees, at least when one is especially looking, that many of the answers found may be *checked* relatively easily. The CAS can arrive at the result in its usual way, and need not be hampered by the need to produce any kind of formal proof. The eventual checking may then be done rigorously à la LCF, with proportionately little extra difficulty. To integrate finding and checking, we can physically link the prover and the CAS. In what follows, these themes appear several times, so we begin by discussing at length the issues that arise.

3. Finding and checking

From the perspective of computer theorem proving, there are interesting differences between proof finding and proof checking. Often the checking process is 'easier' in two particular senses, both of which are illustrated by the factorization of numbers. For example, contrast the task of multiplying:

$$3490529510847650949147849619903898133417764638493387843990820577$$

and

$$32769132993266709549961988190834461413177642967992942539798288533$$

with the task of going from the product (say 'r') back to the above factors. The latter seems more difficult, first from the point of view of computational complexity,[2] and second from the point of view of programming difficulty. Multiplying n-digit numbers even in the most naive way requires only n^2 operations. But factorizing an n-digit number is not even known to be polynomial in n, and certainly seems likely to be worse than quadratic.[3] Moreover, writing an adequate multiprecision arithmetic routine is not a very challenging programming exercise, but present-day factorization methods are rather complex — it's a hot research topic — and often rely for their justification on fairly highbrow mathematics.

In general, to achieve a separation of proof search and proof checking, we want the search stage to produce some kind of 'certificate' that allows the result to be arrived at by proof with acceptable efficiently. In the case of factorization, the certificate was simply the factors. Often the certificate can be construed simply as the 'answer' and the checking process a confirmation of it. But as we shall see, other certificates are possible. Bundy's 'proof plans', for example, essentially use a complete proof as the certificate; obviously a uniquely convenient one for checking by inference. The earlier example of

[2] The security of certain cryptographic schemes depends on that, though perhaps not *only* on that. In fact the above factorization was set as a challenge ('RSA129'), and was eventually achieved by a cooperative effort of around a thousand users lavishing spare CPU cycles on the task.

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

first order automation in HOL is similar, though there the proof search is computationally intensive, whereas in proof planning, the main difference is that the proof search involves sophisticated AI techniques. Indeed according to [8], checking proof plans seems *slower* than finding them, though it is much easier to implement. In fact they report that 'it is an order of magnitude less expensive to find a plan that to execute it', though this may be due to a badly implemented inference engine and the relatively limited problem domain for the planner (inductive proofs).

3.1. Relationship to NP problems

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

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

The complementary problems to the NP ones are the co-NP ones. Here it is *negative* answers that can be accompanied by a certificate allowing easy checking. A good example of a co-NP problem is tautology checking, the dual of the NP-complete problem of Boolean satisfiability. Boolean satisfiability admits an easily checked certificate, viz, a satisfying valuation, but no such certificate exists for tautologies, unless $P = NP$. We may expect, therefore, that our analogs of co-NP complete problems will be harder to support with an efficient checking process. From a theoretical point of view this is almost true by definition, but the practical situation is not so clear. It may be that algorithms that are used with reasonable success to perform search in practice could produce a certificate that allows easy checking.

For example, a problem that looks intuitively complementary to the problem of factorization is primality testing. However as shown by Pratt [32], short certificates are possible and so the problem is not only in co-NP but also in NP.[4] An especially strong form of this result is due to Pomerance [31], who shows that every prime has an $O(log\ p)$ certificate, or more precisely that 'for every prime p there is a proof that it is prime which requires for its verification $(\frac{5}{2}+o(1))log_2 p$ multiplications mod p'. Of course whether useful primality testing algorithms can naturally produce such certificates is still an open question, but this very idea has been explored in practice by Elbers [15].

3.2. What must be internalized?

The separation of proof search and proof checking offers an easy way of incorporating sophisticated algorithms, computationally intensive search techniques and elaborate heuristics, without compromising either the efficiency of search or the security of proofs eventually found. It is interesting to enquire which algorithms can, in theory and in practice, provide the appropriate certificates. If formal checkability is considered important, it may lead to a shift in emphasis in the development and selection of algorithms for mathematical computations.

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

For example, consider the use of Knuth-Bendix completion to derive consequences from a set of algebraic laws. Slind [34] has implemented this procedure in HOL, where at each stage the new equations are derived by inference. However the fact that any particular rewrite system resulting is canonical is *not* proved inside the logic. This could be done either in an ad hoc way for the particular system concerned, or could be done by internalizing the algorithm and proving its correctness. Either would require much more work, and cost much more in efficiency. And if all we want is to prove *positive* consequences of the rewrites, this offers no benefits. On the other hand to prove *negative* results, e.g. that a group exists that does not satisfy a certain equation, then this kind of internalization would be necessary. Such a theme often appears in HOL, where the steps in an algorithm may all be justified by inference, but the overall reasoning justifying its usefulness, completeness, efficiency or whatever are completely external (one might say informal). We shall give an example below where the correctness of a procedure is easy

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

to see, but its completeness requires slightly more substantial mathematics. Specifically, the presence of a certain factor suffices for the proof, but the knowledge that such a factor will always be found, which justifies completeness, is completely external.

4. Combining systems

The general issue of combining theorem provers and other symbolic computation systems has recently been attracting more attention. As well as our own experiments with a computer algebra system, detailed below, HOL has for example been linked to other theorem provers [2] and to model checkers [33]. Methodologies for cooperation between systems are classified by Calmet et al. [9] according to several different properties. For example, if more than two systems are involved, the network topology is significant: are there links between each pair or is all communication mediated by some central system? A related issue is which, if any, systems in the network act as master or slave. In our case, we use a system with just two components, HOL and Maple, and HOL is clearly the master.

4.1. TRUST

One of the most interesting categorizations of such arrangements is according to *degree of trust*. For example, our work does not involve trusting Maple at all, since all its results are rigorously checked. However one might, at the other end of the scale, trust all Maple's results completely: if when given an expression E, Maple returns E', then the theorem $\vdash E = E'$ is accepted. Such an arrangement, exploiting a link between Isabelle and Maple, is described by Ballarin et al. [3]. This runs the risk of importing into the theorem prover all the defects in correctness of the CAS, which we have already discussed. However it may, if used only for problems in a limited domain, be quite acceptable. For example, despite our best efforts, arithmetic by inference in HOL is inevitably rather slow, and the results of CASs are generally pretty reliable. So one might use such a scheme, restricted to the evaluation of ground terms, perhaps making explicit in the HOL theorem, in the case of irrational numbers, the implied accuracy of the CAS's result. For example if '`evalf(Pi,20)`' in Maple returns `3.1415926535897932385`, we may assert the theorem:

$$\vdash |\pi - 3.1415926535897932385| < 10^{-18}$$

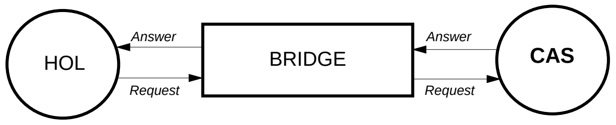
An interesting way of providing an intermediate level of trust was proposed by Mike Gordon.[5] This is to tag each theorem $\vdash \psi$ derived by trusting an external tool with an additional assumption logically equivalent to falsity. We can define a new constant symbol for this purpose, bearing the name of the external tool concerned; `MAPLE` in our case. The theorem `MAPLE` $\vdash \psi$ is, from a logical point of view, trivially true. But pragmatically it is quite appealing. First, it has a natural reading 'assuming Maple is sound, then ψ'. Moreover, any derived theorems that use this fact will automatically inherit the assumption `MAPLE` and any others like it, so indicating clearly the dependence of the theorem on the correctness of external tools. A slightly different version of this idea is implemented as the 'oracles' mechanism in the latest version of Isabelle [30].

Finally, another possible method is to perform checking yet *defer* it until later, e.g. batching the checks and running them all overnight. This fits naturally into the framework of lazy theorems proposed by Boulton [6]. Of course, there is the defect that if one of the checks fails overnight, then the day's work might be invalidated, but one hopes that the external tool will generally give correct answers.

Our finding-checking approach may seem unnecessarily pedantic and difficult, but it gives a reliable way of conforming to a definite logical calculus. In any case, from a research point of view, it seems to raise the most interesting questions.

4.2. Implementation Issues

In order to connect a computer algebra and a prover, we need first to define how data are exchanged and how control is passed between the two systems. In our experiment, the implementation has been directly inspired by CAS/PI [26]. In this work, the author presents an interactive mathematical environment where different computer algebra systems can be used to perform computations. The connection between CAS/PI and the different computer algebras use the idea of *software bus* developed in [13]. Applying this to the problem of connecting a theorem prover to a computer algebra, we obtain a software bus with three different processes: HOL, Maple, and a bridge:



Data integration has been obtained by adding the formalism of HOL terms to the ones already defined in [26] for Maple and Mathematica terms. Con-

[5] Message to info-hol mailing list on 13 Jan 93, available on the Web as `ftp://ftp.cl.cam.ac.uk/hvg/info-hol-archive/09xx/0972`.

trol integration follows a simple question/answer model. A request is sent by HOL which is received and translated by the bridge and then resent to the computer algebra system. The answer from the CAS is then transferred back to the prover through the bridge.

From the toplevel of the prover one can access the CAS by the function `call_CAS` which takes a HOL term and a method to be applied on this term, and returns the resulting term. In the current implementation, only two methods are available. `SIMPLIFY` gives the answer as if the term had been typed at the top level of the CAS. `FACTORIZE` tries to factorize the term. Here are some examples of how this function could be used:

```
# call_CAS "(((FACT 5)EXP2)-1)MOD(3EXP2)" ‘SIMPLIFY‘;;
"8" : term
Run time: 0.1s

# call_CAS "(x*x)+(7*x)+12" ‘FACTORIZE‘;;
"(x + 3) * (x + 4)" : term
Run time: 0.1s
```

The organization we have adopted is very flexible. First it is straightforward to share one computer algebra between different HOL processes. For this, every new HOL session simply needs to be incorporated into the same software bus. A more interesting extension is the possibility to broadcast requests from the prover to several CASs. In that case, in addition to connecting the different CASs to the same software bus, an auction mechanism for prioritizing results from the CASs is required. For the moment, the benefit of such an extension is not obvious. The only other CAS we could easily integrate is Mathematica as its term syntax is known by our system. This problem of data integration is not new. As a matter of fact in the computer algebra community there have been a number of attempts to arrive at a standard for language and translation; recently many of these have been unified in an ambitious project called 'OpenMath'.[6] Time will tell whether this will be a success.

Independently of these extensions, two limitations of our implementation are worth noticing. First we use a stateless communication between HOL and Maple. While this has not been a problem so far in our experiments, we believe a more complex dialogue between the theorem prover and the CAS would make it necessary to relax this limitation. Second, all type information that is present in a term is automatically deleted when the term is shipped to Maple. This is a consequence of the computer algebra system we choose,

[6] See the OpenMath Web page,
`http://www.rrz.uni-koeln.de/themen/Computeralgebra/OpenMath/index.html`
.

where terms are type-free. Other computer algebra systems such as Axiom [25] have proved that real use can be made of type information. Connecting HOL with these systems would imply coupling the protocol to exchange data with a system that can interpret type values.

5. A running example

In some cases, verification is not trivial, but requires some quite powerful simplification mechanisms to verify the answer. For example, one can verify antiderivatives by differentiating, and closed-form summations by induction or differencing [12], but this still may leave a difficult task of simplifying the result to show it is as required. We will now give an extended example which illustrates this phenomenon, and show how it may be solved by a second pass, exploitation the same finding/checking separation. Thus, we illustrate how the combined system can be used in a 'cascaded' or nested fashion.

The procedure we present is a tactic written in HOL that automatically finds the integrals of trigonometric polynomials. We illustrate the steps of this procedure on the following simple trigonometric integral:

$$\int_0^x sin(u)^3 \; du$$

In Step 1, we ask Maple to simplify this integral and it tells us that the result is

$$-\frac{1}{3}sin(x)^2cos(x) - \frac{2}{3}cos(x) + \frac{2}{3}$$

which we will write as $f(x)$. Now if we can prove that $\frac{d}{dx}f(x) = sin(x)^3$, then by the Fundamental Theorem of Calculus, we can derive:

$$\int_0^x sin(u)^3 \; du = f(x) - f(0)$$

and since we also have $f(0) = 0$ (this amounts to checking that the right 'constant of integration' has been used) Maple's result would be confirmed. In Step 2 we differentiate $f(x)$ inside HOL, using `DIFF_CONV`, a derived rule which works strictly by inference and collects any necessary side-conditions on the derivative theorem. We get:

$$\vdash \frac{d}{dx}f(x) = -\frac{1}{3}(2sin(x)cos(x)cos(x) - sin(x)^3) + \frac{2}{3}sin(x)$$

Unfortunately, even after routine simplification, we have not derived $sin(x)^3$. Note that this is not because our simplification in HOL is not powerful enough; Maple itself gives more or less the same result when asked to perform the differentiation. However, it is not hard to give a simplification procedure that suffices for proving that a polynomial in $sin(x)$ and $cos(x)$ is identically zero. The idea is that every trigonometric polynomial that is identically zero must have $sin(x)^2 + cos(x)^2 - 1$ as a factor. So in Step 3 we ask Maple to factorize

$$-\frac{1}{3}(2uvv - u^3) + \frac{2}{3}u - u^3$$

and finally in Step 4 we check the result in HOL by a straightforward expansion, so we get a theorem:

$$\vdash -\frac{1}{3}(2uvv - u^3) + \frac{2}{3}u - u^3 = -\frac{2}{3}u(v^2 + u^2 - 1)$$

which contains the required factor. From this we get the result we wanted: Maple's original result is echoed in HOL. But this time the result is an absolutely formal statement based on a precisely defined and rigorously constructed integration theory, itself founded on a definitionally constructed theory of reals. Moreover, the HOL proof we have derived with Maple's help uses just the dozen or so primitive rules of higher order logic, even to perform the arithmetic on polynomial coefficients. We speculate that such a 'high level' mathematical result has never been proved in such a painstaking way before. But with our combined system, it and many others like it can be done automatically in a few seconds.

6. Other applications

There are several examples of computer algebra results which may be checked relatively easily:

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

In most cases the certificate is simply the answer. An exception is the GCD, where a slightly more elaborate certificate is better for our purposes. If we ask Maple to find the GCD of $x^2 - 1$ and $x^5 + 1$ using its `gcd` function, for example, it responds with $x + 1$. How can this result be checked? It's certainly straightforward to check that this is *a* common divisor. If we don't want to code polynomial division ourselves in HOL, we can call Maple's `divide` function, and then simply verify the quotient as above. But how can we prove that $x + 1$ is a *greatest* common divisor[7]? At first sight, there is no easy way, short of replicating something like the Euclidean algorithm inside the logic (though that isn't a really difficult prospect).

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

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

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

As for integration, the techniques we describe here are quite limited, in that in general they apply only to indefinite integrals. However there are some special circumstances where one can achieve similar checking by differentiating with respect to free variables in the body of the integral, leading to the popular trick of 'differentiating under the integral sign' [1].

We have contented ourselves with trying out elementary examples such as the one presented above, and have not yet made a systematic study of symbolic algorithms to assess which ones admit useful certificates. The certificates we have been using so far are still very simple and are composed in terms of the answers of the CAS. But it is clear from the above that our technique is in principle of reasonably wide applicability, and for genuinely useful problems at that. This is borne out by recent work on the verification of a floating point algorithm [23]. Here, Maple is used to find squarefree decompositions, Sturm sequences and sets of isolating intervals for the roots of polynomials, as well as performing polynomial division. These results are all checked rigorously in HOL in order to arrive at a theorem provably characterizing all the roots of a certain polynomial. (Admittedly, the version of HOL used was different and so the particular linkup described here was not exploited.)

7. Summary and related work

The most substantial attempt to create a sound and reliable computer algebra system is probably the work of Beeson [4] on Mathpert. This is a computer

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

algebra system designed mainly for educational use, and the intended use dictates two important features. First, it attempts to perform only logically sound steps. Secondly, it tries to justify its reasoning instead of producing *ex cathedra* pronouncements. Since it has not yet been used extensively, it's hard to judge how successful it is. By contrast, our effort is relatively modest, but gets quite a long way for relatively little implementation difficulty.

Conversely, the most convincing example of importing real logical expressiveness and theorem proving power into computer algebra is the work of Clarke and Zhao [12] on Analytica. Here a theorem prover is coded in the Mathematica system. It is capable of proving some remarkably complicated theorems, e.g. some expressions due to Ramanujan, completely automatically. However, it still relies on Mathematica's native simplifier, so it is does yet provide such a high level of rigour as our LCF approach. Other research on linking theorem provers and CASs which we have not already mentioned is by Farmer et al. [16], who describe how the list of assumptions in a sequent can be used to deal with contextual computation.

Our general theme of checkability has been stressed by several researchers, notably Blum [5], who also mentions the GCD example. He suggests that in many situations, checking results may be more practical and effective than verifying code. This argument is related to, in some sense a generalization of, arguments by Harrison [22] in favour of the LCF approach to theorem proving rather than so-called 'reflection'. Mehlhorn et al. [29] describe the addition of verification to routines in the LEDA library of C++ routines for computational geometry (e.g. finding convex hulls and Voronoi diagrams). Our interest is a little different in that it involves checking according to a formal deductive calculus. However it seems that many of the same issues arise. For instance, they remark that 'a convex hull program that delivers a triangulation of the hull is much easier to check than a program that only returns the hull polytope', which parallels our example of a certificate for the GCD consisting of more than just the answer.

References

1. G. Almkvist and D. Zeilberger. The method of differentiating under the integral sign. *Journal of Symbolic Computation*, 10:571–591, 1990.
2. M. Archer, G. Fink, and L. Yang. Linking other theorem provers to HOL using PM: Proof manager. In Claesen and Gordon [11], pages 539–548.
3. C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In A. H. M. Levelt, editor, *International Symposium on Symbolic and Algebraic Computation, ISSAC'95*, pages 150–157, Montreal, 1995. Association for Computing Machinery.
4. M. Beeson. Mathpert: Computer support for learning algebra, trig, and calculus. In A. Voronkov, editor, *Logic programming and automated reasoning: international conference LPAR '92*, volume 624 of *Lecture Notes in Computer Science*, pages 454–456, St. Petersburg, Russia, 1992. Springer-Verlag.

5. M. Blum. Program result checking: A new approach to making programs more reliable. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 1993. Springer-Verlag.
6. R. Boulton. A lazy approach to fully-expansive theorem proving. In Claesen and Gordon [11], pages 19–38.
7. R. J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993. Author's PhD thesis.
8. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–323, 1991.
9. J. Calmet and K. Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In F. Baader and K. U. Schulz, editors, *Proceedings of the First International Workshop 'Frontiers of Combining Systems' (FroCoS'96)*, Kluwer Series on Applied Logic, pages 133–146, Munich, 1996. Kluwer.
10. A. Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
11. L. J. M. Claesen and M. J. C. Gordon, editors. *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, 1992. North-Holland.
12. E. Clarke and X. Zhao. Analytica — a theorem prover for Mathematica. Technical report, School of Computer Science, Carnegie Mellon University, 1991.
13. D. Clément, F. Montagnac, and V. Prunet. Integrated software components: a paradigm for control integration. In A. Endres and H. Weber, editors, *Software development environments and CASE technology: European symposium*, volume 509 of *Lecture Notes in Computer Science*, pages 167–177, Königwinter, 1991. Springer-Verlag.
14. R. M. Corless and D. J. Jeffrey. Well... it isn't quite that simple. *SIGSAM Bulletin*, 26(3):2–6, August 1992.
15. H. Elbers. Construction of short formal proofs of primality. Preprint, 1996.
16. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Reasoning with contexts. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems: International Symposium, DISCO '93*, volume 722 of *Lecture Notes in Computer Science*, pages 216–228, Gmunden, Austria, 1993. Springer-Verlag.
17. P. J. Freyd and A. Scedrov. *Categories, allegories*. North-Holland, 1990.
18. M. J. C. Gordon. Representing a logic in the LCF metalanguage. In D. Néel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.
19. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
20. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
21. J. Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 5:35–59, 1994.
22. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. On Web: `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz`.
23. J. Harrison. Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.
24. R. Henstock. A Riemann-type integral of Lebesgue power. *Canadian Journal of Mathematics*, 20:79–87, 1968.

25. R. D. Jenks and R. S. Sutor. *AXIOM: the scientific computation system*. Springer-Verlag, 1992.
26. N. Kajler. CAS/Pi: A portable and extensible interface for computer algebra systems. In P. S. Wang, editor, *International Symposium on Symbolic and Algebraic Computation, ISSAC'92*, pages 376–386. Association for Computing Machinery, 1992.
27. R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, pages 170–176, University of California at Davis, Davis CA, USA, 1991. IEEE Computer Society Press.
28. J. Kurzweil. Generalized ordinary differential equations and continuous dependence on a parameter. *Czechoslovak Mathematics Journal*, 82:418–446, 1958.
29. K. Mehlhorn et al. Checking geometric programs or verification of geometric structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (FCRC'96)*, pages 159–165, Philadelphia, 1996. Association for Computing Machinery.
30. L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
31. C. Pomerance. Very short primality proofs. *Mathematics of Computation*, 48:315–322, 1987.
32. V. Pratt. Every prime has a succinct certificate. *SIAM Journal of Computing*, 4:214–220, 1975.
33. C. Seger and J. J. Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Department of Computer Science, University of British Columbia, 2366 Main Mall, University of British Columbia, Vancouver, B.C, Canada V6T 1Z4, 1991.
34. K. Slind. An implementation of higher order logic. Technical Report 91-419-03, University of Calgary Computer Science Department, 2500 University Drive N. W., Calgary, Alberta, Canada, TN2 1N4, 1991. Author's Masters thesis.