Formal verification

John Harrison (johnh@ichips.intel.com)
Intel Corporation RA2-451,
2501 NW 229th Avenue, Hillsboro, OR 97124, USA

Abstract. These lectures are intended to give a broad overview of the most important formal verification techniques. These methods are applicable to hardware, software, protocols etc. and even to checking proofs in pure mathematics, though I will emphasize the applications to verification. Since other lecturers will cover some of the topics in more detail (in particular model checking), my lectures will cover mainly the material on deductive theorem proving, which in any case is my special area of interest. The arrangement of material in these notes is roughly in order of logical complexity, starting with methods for propositional logic and leading up to general theorem proving, then finishing with a description of my own theorem proving program 'HOL Light' and an extended case study on the verification of a floating-point square root algorithm used by Intel.

1. Propositional logic
2. Symbolic simulation
3. Model checking
4. Automated theorem proving
5. Arithmetical theories
6. Interactive theorem proving
7. HOL Light
8. Case study of floating-point verification

The treatment of the various topics is quite superficial, with the aim being overall perspective rather than in-depth understanding. The last lecture is somewhat more detailed and should give a good feel for the realities of formal verification in this field.

My book on theorem proving (Harrison 2009) covers many of these topics in more depth, together with simple-minded example code (written in Objective Caml) implementing the various techniques.

`http://www.cl.cam.ac.uk/~jrh13/atp/index.html`

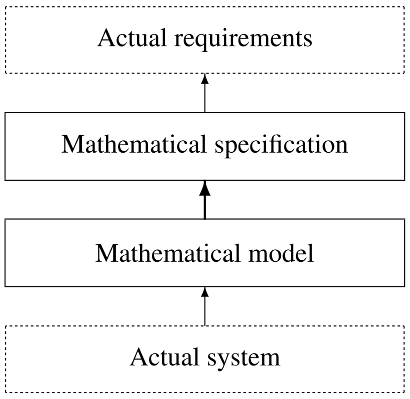
Other useful references for the material here are (Bradley and Manna 2007; Clarke, Grumberg, and Peled 1999; Kroening and Strichman 2008; Kropf 1999; MacKenzie 2001; Peled 2001).

0. Introduction

As most programmers know to their cost, writing programs that function correctly in all circumstances — or even saying what that means — is difficult. Most large programs contain 'bugs'. In the past, hardware has been substantially simpler than software, but this difference is eroding, and current leading-edge microprocessors are also extremely complex and usually contain errors. It has often been noted that mere testing, even on clever sets of test cases, is usually inadequate to guarantee correctness on *all* inputs, since the number of possible inputs and internal states, while finite, is usually astronomically large. For example (Dijkstra 1976):

> As I have now said many times and written in many places: program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

The main alternative to testing is formal verification, where it is rigorously *proved* that the system functions correctly on all possible inputs. This involves forming mathematical models of the system and its intended behaviour and linking the two:



Hardware versus software

The facts that (i) getting a mathematical proof right is also difficult (DeMillo, Lipton, and Perlis 1979), and (ii) correctness of formal models does not necessarily imply correctness of the actual system (Fetzer 1988) caused much pointless controversy in the 70s. It is now widely accepted that formal verification, with the proof itself checked by machine, gives a much greater degree of confidence than traditional techniques.

The main impediment to greater use of formal verification is not these generalist philosophical objections, but just the fact that it's rather difficult. Only in a few isolated safety-critical niches of the software industry is any kind of formal verification widespread, e.g. in avionics. But in the hardware industry, formal verification is widely practised, and increasingly seen as necessary. We can identify at least three reasons:

- Hardware is designed in a more modular way than most software. Constraints of interconnect layering and timing means that one cannot really design ‘spaghetti hardware’.
- More proofs in the hardware domain can be largely automated, reducing the need for intensive interaction by a human expert with the mechanical theorem-proving system.
- The potential consequences of a hardware error are greater, since such errors often cannot be patched or worked around, and may *in extremis* necessitate a hardware replacement.

To emphasize the last point, an error in the FDIV (floating-point division) instruction of some early Intel Intel® Pentium® processors resulted in 1994/5 in a charge to Intel of approximately $500M. Given this salutary lesson, and the size and diversity of its market, it’s therefore understandable that Intel is particularly interested in formal verification.

The spectrum of formal verification techniques

There are a number of different formal verification techniques used in industry. Roughly speaking, these use different logics to achieve a particular balance between generality and automation. At one end of the scale, propositional logic has limited expressiveness, but (amazingly) efficient decision methods are available. At the other end, higher-order logic is very expressive but no efficient decision methods are available (in fact, higher-order validity is not even recursively enumerable) and therefore proofs must be constructed interactively with extensive user guidance. The lectures that follow are organized on the same basis, with propositional logic first, followed by symbolic simulation and temporal logic model checking, and finally general theorem proving at the end.

The particular method that is most appropriate for a given task may depend on details of that application. Needless to say, the effective combination of all methods is attractive, and there has been much interest in this idea (Seger and Joyce 1991; Joyce and Seger 1993; Rajan, Shankar, and Srivas 1995). Another ‘hybrid’ application of formal verification is to prove correctness of some of the CAD tools used to produce hardware designs (Aagaard and Leeser 1994) or even of the abstraction and reduction algorithms used to model-check large or infinite-state systems (Chou and Peled 1999).

There has been a recent trend has towards *partial* verification, not necessarily seeking a complete proof of correctness, or even formulating what that means, but still proving important properties using techniques from formal verification. A common approach is to replace the actual system with an abstraction that may lose many details while still exposing some key aspect(s) that are then more tractable to reason about. For example, Microsoft’s Static Driver Verifier forms an abstraction of code for device drivers and checks compliance with various properties like the proper release of allocated resources, using symbolic representations and arithmetical theorem proving. The Airbus A380 flight control software has been subjected to abstract interpretation to prove rigorously that, subject to natural environmental constraints, no floating-point overflow exceptions can occur.

A widespread perception is that formal verification is an impractical pipedream with no connection to traditional methods. I prefer to see complete formal verification as the extreme point on a spectrum that takes in various intermediate possibilities. Some static checking of programs is already standard, if only via typechecking of the programming language or tools like `lint` and its derivatives. Examples like SDV and the A380 verification show that the intermediate space is quite rich in possibilities. Thus, even if one neglects the numerous successes of complete functional verification, verification technology should be considered a natural component of programming practice.

1. Propositional logic

I assume that everyone knows what propositional logic is. The following is mainly intended to establish notation. Propositional logic deals with basic assertions that may be either true or false and various ways of combining them into composite propositions, without analyzing their internal structure. It is very similar to Boole's original algebra of logic, and for this reason the field is sometimes called *Boolean algebra* instead. In fact, the Boolean notation for propositional operators is still widely used by circuit designers.

English	Standard	Boolean	Other
false	$\bot$	0	F
true	$\top$	1	T
not p	$\neg p$	$\overline{p}$	$-p$, $\sim p$
p and q	$p \wedge q$	pq	$p\&q$, $p \cdot q$
p or q	$p \vee q$	$p+q$	$p \mid q$, $p\ or\ q$
p implies q	$p \Rightarrow q$	$p \leqslant q$	$p \rightarrow q$, $p \supset q$
p iff q	$p \Leftrightarrow q$	$p = q$	$p \equiv q$, $p \sim q$

For example, $p \wedge q \Rightarrow p \vee q$ means 'if p and q are true, then p or q is true'. We assume that 'or' is interpreted inclusively, i.e. $p \vee q$ means 'p or q or both'. This formula is therefore always true, or a *tautology*.

Logic and circuits

At a particular time-step, we can regard each internal or external wire in a (binary) digital computer as having a Boolean value, 'false' for 0 and 'true' for 1, and think of each circuit element as a Boolean function, operating on the values on its input wire(s) to produce a value at its output wire. The most basic building-blocks of computers used by digital designers, so-called *logic gates*, correspond closely to the usual logical connectives. For example an 'AND gate' is a circuit element with two inputs and one output whose output wire will be high (true) precisely if both the input wires are high, and so it corresponds exactly in behaviour to the 'and' ('$\wedge$') connective. Similarly a 'NOT gate' (or *inverter*) has one input wire and one output wire, which is high when the input is low and low when the input is high; hence it corresponds to the 'not' connective ('$\neg$'). Thus, there is a close correspondence between digital circuits and formulas which can be sloganized as follows:

Digital design	Propositional Logic
circuit	formula
logic gate	propositional connective
input wire	atom
internal wire	subexpression
voltage level	truth value

An important issue in circuit design is proving that two circuits have the same function, i.e. give identical results on all inputs. This arises, for instance, if a designer makes some special optimizations to a circuit and wants to check that they are “safe”. Using the above correspondences, we can translate such problems into checking that a number of propositional formulas $P_n \Leftrightarrow P'_n$ are tautologies. Slightly more elaborate problems in circuit design (e.g. ignoring certain ‘don’t care’ possibilities) can also be translated to tautology-checking. Thus, efficient methods for tautology checking directly yield useful tools for hardware verification.

Tautology checking

Tautology checking is apparently a difficult problem in general: it is co-NP complete, the dual problem ‘SAT’ of propositional satisfiability being the original NP-complete problem (Cook 1971). However, there are a variety of algorithms that often work well on the kinds of problems arising in circuit design. And not just circuit design. The whole point of NP completeness is that many other apparently difficult combinatorial problems can be reduced to tautology/satisfiability checking. Recently it’s become increasingly clear that this is useful not just as a theoretical reduction but as a practical approach. Surprisingly, many combinatorial problems are solved better by translating to SAT than by customized algorithms! This probably reflects the enormous engineering effort that has gone into SAT solvers.

The simplest method of tautology checking is, given a formula with n primitive propositional variables, to try all 2^n possible combinations and see if the formula always comes out true. Obviously, this may work for small n but hardly for the cases of practical interest. More sophisticated SAT checkers, while still having runtimes exponential in n in the worst case, do very well on practical problems that may even involve *millions* of variables. These still usually involve true/false case-splitting of variables, but in conjunction with more intelligent simplification.

The Davis-Putnam method

Most high-performance SAT checkers are based on the venerable Davis-Putnam algorithm (Davis and Putnam 1960), or more accurately on the ‘DPLL’ algorithm, a later improvement (Davis, Logemann, and Loveland 1962).

The starting-point is to put the formula to be tested for satisfiability in ‘conjunctive normal form’. A formula is said to be in conjunctive normal form (CNF) when it is an ‘and of ors’, i.e. of the form:

$$C_1 \land C_2 \land \cdots \land C_n$$

with each C_i in turn of the form:

$$l_{i1} \lor l_{i2} \lor \cdots \lor l_{im_i}$$

and all the l_{ij}'s *literals*, i.e. primitive propositions or their negations. The individual conjuncts C_i of a CNF form are often called *clauses*. We usually consider these as sets, since both conjunction and disjunction are associative, commutative and idempotent, so it makes sense to talk of $\bot$ as the empty clause. If C_i consists of one literal, it is called a *unit clause*. Dually, disjunctive normal form (DNF) reverses the role of the 'and's and 'or's. These special forms are analogous to 'fully factorized' and 'fully expanded' in ordinary algebra — think of $(x+1)(x+2)(x+3)$ as CNF and $x^3+6x^2+11x+6$ as DNF. Again by analogy with algebra, we can always translate a formula into CNF by repeatedly rewriting with equivalences like:

$$\begin{aligned}
\neg(\neg p) &\Leftrightarrow p \\
\neg(p \land q) &\Leftrightarrow \neg p \lor \neg q \\
\neg(p \lor q) &\Leftrightarrow \neg p \land \neg q \\
p \lor (q \land r) &\Leftrightarrow (p \lor q) \land (p \lor r) \\
(p \land q) \lor r &\Leftrightarrow (p \lor r) \land (q \lor r)
\end{aligned}$$

However, this in itself can cause the formula to blow up exponentially before we even get to the main algorithm, which is hardly a good start. One can do better by introducing new variables to denote subformulas, and putting the resulting list of equivalences into CNF — so-called *definitional CNF*. It's not hard to see that this preserves satisfiability. For example, we start with the formula:

$$(p \lor (q \land \neg r)) \land s$$

introduce new variables for subformulas:

$$\begin{array}{l}
(p_1 \Leftrightarrow q \land \neg r)\land \\
(p_2 \Leftrightarrow p \lor p_1)\land \\
(p_3 \Leftrightarrow p_2 \land s)\land \\
p_3
\end{array}$$

then transform to CNF:

$$\begin{array}{l}
(\neg p_1 \lor q) \land (\neg p_1 \lor \neg r) \land (p_1 \lor \neg q \lor r)\land \\
(\neg p_2 \lor p \lor p_1) \land (p_2 \lor \neg p) \land (p_2 \lor \neg p_1)\land \\
(\neg p_3 \lor p_2) \land (\neg p_3 \lor s) \land (p_3 \lor \neg p_2 \lor \neg s)\land \\
p_3
\end{array}$$

The DPLL algorithm is based on the following satisfiability-preserving transformations:

I The 1-literal rule: if a unit clause p appears, remove $\neg p$ from other clauses and remove all clauses including p.
II The affirmative-negative rule: if p occurs *only* negated, or *only* unnegated, delete all clauses involving p.
III Case-splitting: consider the two separate problems by adding p and $\neg p$ as new unit clauses.

If you get the empty set of clauses, the formula is satisfiable; if you get an empty *clause*, it is unsatisfiable. Since the first two rules make the problem simpler, one only applies the case-splitting rule when no other progress is possible. In the worst case, many case-splits are necessary and we get exponential behaviour. But in practice it works quite well.

Industrial-strength SAT checkers

The above simple-minded sketch of the DPLL algorithm leaves plenty of room for improvement. The choice of case-splitting variable is often critical, the formulas can be represented in a way that allows for efficient implementation, and the kind of backtracking that arises from case splits can be made more efficient via 'intelligent backjumping' and 'conflict clauses'. A few recent highly efficient DPLL-based theorem provers are Chaff (Moskewicz, Madigan, Zhao, Zhang, and Malik 2001), BerkMin (Goldberg and Novikov 2002) and and MiniSat (Eén and Sörensson 2003).

Another interesting technique that is used in the tools from Prover Technology (`www.prover.com`), as well as the experimental system Heerhugo (Groote 2000), is Stålmarck's dilemma rule (Stålmarck and Säflund 1990). This involves using case-splits in a non-nested fashion, accumulating common information from both sides of a case split and feeding it back:

$$\begin{tikzcd}
& \Delta \arrow[dl] \arrow[dr] & \\
\Delta \cup \{\neg p\} \arrow[d, "R"'] & & \Delta \cup \{p\} \arrow[d, "R"] \\
\Delta \cup \Delta_0 \arrow[dr] & & \Delta \cup \Delta_1 \arrow[dl] \\
& \Delta \cup (\Delta_0 \cap \Delta_1) &
\end{tikzcd}$$

In some cases, this works out much better than the usual DPLL algorithm. For a nice introduction, see (Sheeran and Stålmarck 2000). Note that this method is covered by patents (Stålmarck 1994).

2. Symbolic simulation

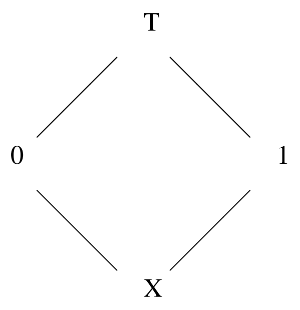
As noted, testing is also widely used in hardware design, and until a decade or so ago was the exclusive way of trying to establish correctness. Usually what is tested is not the circuit itself but some formal model — the whole idea is to flush out errors *before* the expensive step of building the hardware. The usual term for this kind of testing is *simulation*. We'll now consider two refinements of basic simulation, then show how they can profitably be combined.

Ternary simulation

One important optimization of straightforward simulation is *ternary simulation* (Bryant 1991a), which uses values drawn from 3-element set: as well as '1' (true) and '0' (false) we have an 'X', denoting an 'unknown' or 'undefined' value. This is *not* meant to imply that a wire in the circuit itself is capable of attaining some third truth-value, but merely reflects the fact that we don't know whether it is 0 or 1. The point of ternary simulation is that many values in the circuit may be irrelevant to some current concern, and we can abstract them away by setting them to 'X' during simulation. (For example, if we are analyzing a small part of a large circuit, e.g. an adder within a complete microprocessor, then only a few bits outside the present module are likely to have any effect on its behaviour and we can set others to X without altering the results of simulation.) We then extend the basic logical operations realized by logic gates to ternary values with this interpretation in mind. For example, we want $0 \wedge X = 0$, since whatever the value on one input, the output will be low if the other input is. Formally, we can present the use of X as an abstraction mapping, for which purpose it's also convenient to add an 'overconstrained' value T, with the intended interpretation:

$$\begin{aligned}
T &= \{\} \\
0 &= \{0\} \\
1 &= \{1\} \\
X &= \{0,1\}
\end{aligned}$$

It's useful to impose an information ordering to give this quaternary lattice:



As expected, the truth tables are monotonic with respect to this ordering:

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
X	X	X	X	X	X
X	0	0	X	X	X
X	1	X	1	1	X
0	X	0	X	1	X
0	0	0	0	1	1
0	1	0	1	1	0
1	X	X	1	X	X
1	0	0	1	0	0
1	1	1	1	1	1

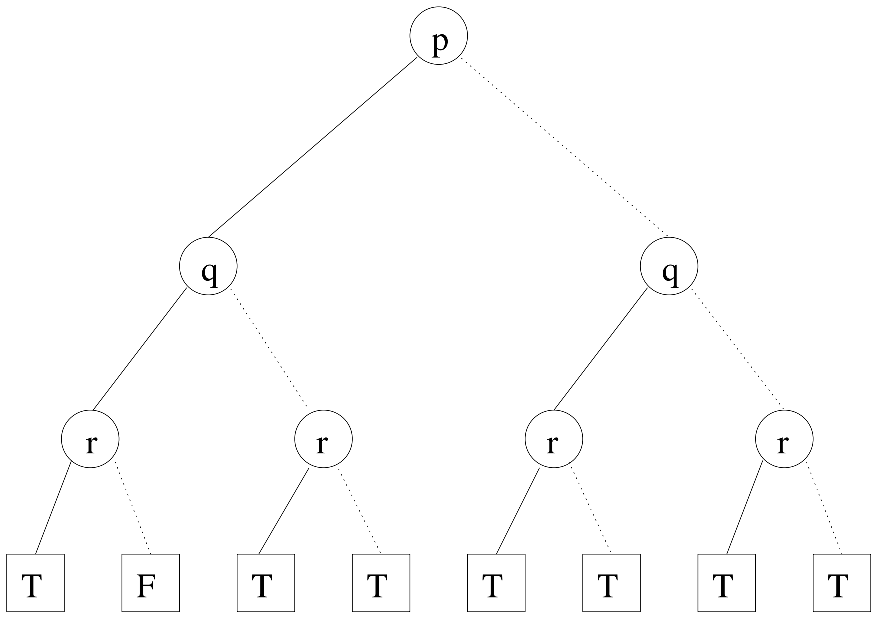
Symbolic simulation

Since we are working in an abstract model, we can easily generalize conventional simulation, which tests on particular combinations of 1s and 0s, to *symbolic simulation* (Carter, Joyner, and Brand 1979), where variables are used for some or all of the inputs, and the outputs are correspondingly symbolic expressions rather than simple truth-values. Testing equivalence then becomes testing of the Boolean expressions. Thus, if we use variables for all inputs, we essentially get back to combinational equivalence checking as we considered before. The main differences are:

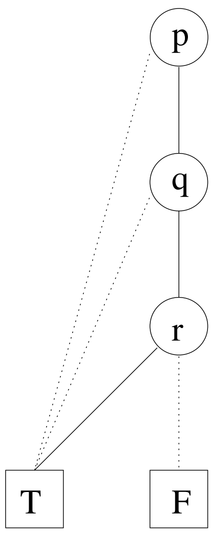
- We may use a more refined circuit model, rather than a simple Boolean switch model
- We explicitly compute the expressions that are the "values" at the internal wires and at the outputs as functions of the input variables, rather than just asserting relations between them.

Instead of using general propositional formulas to represent the signals, we can use a canonical representation, where equivalent formulas are represented by the same data structure. Equivalence checking then becomes cheap. The most important such representation is BDDs (binary decision diagrams). Symbolic simulation often does work quite well with BDDs and allows many circuits to be verified exhaustively (Bryant 1985).

BDDs are essentially a representation of Boolean functions as decision trees. We can think of them as laying out a truth table but sharing common subcomponents. Consider the truth table for a propositional formula involving primitive propositions $p_1, \ldots, p_n$. Rather than a 2^n-row truth table, we can consider a binary tree indicating how truth assignments for the successive atoms yield a truth value for the whole expression. For example, the function $p \wedge q \Rightarrow q \wedge r$ is represented as follows, where dashed lines indicate the 'false' assignment of a variable, and solid lines the 'true'.



However, there are many common subtrees here, and we can imagine sharing them as much as possible to obtain a directed acyclic graph. Although this degree of saving may be atypical, there is always some potential for sharing, at least in the lowest level of the tree. There are after all "only" 2^{2^k} possible subtrees involving k variables, in particular only two possible leaf nodes 'true' and 'false'.



In general, a representation of a Boolean formula as a binary tree with branch nodes labelled with the primitive atoms and leaf nodes either 'false' or 'true' is called a *binary decision tree*. If all common subtrees are maximally shared to give a directed acyclic graph, it is called a *binary decision diagram* (Lee 1959; Akers 1978). As part of maximal sharing, we assume that there are no redundant branches, i.e. none where the 'true' and 'false' descendents of a given node are the same subgraph — in such a configuration that subgraph could be replaced by one of its children. If the variable ordering is the same along all branches, we have *reduced ordered binary decision diagrams* (ROBDDs) introduced by Bryant (Bryant 1986). Nowadays, when people say 'BDD' they normally mean ROBDD, though there have been experiments with non-canonical variants.

Surprisingly many interesting functions have quite a compact BDD representation. However, there are exceptions such as multipliers (Bryant 1986) and the 'hidden weighted bit' function (Bryant 1991b). In these cases it is difficult to apply BDD-based methods directly.

Symbolic trajectory evaluation

A still more refined (and at first sight surprising) approach is to *combine* ternary and symbolic simulation, using Boolean variables to parametrize ternary values. Consider the rather artificial example of verifying that a piece of combinational circuitry with 7 inputs and one output implements a 7-input AND gate. In conventional simulation we would need to check all $2^7 = 128$ input values. In symbolic simulation we would only need to check one case, but that case is a symbolic expression that may be quite large. In ternary simulation, we could verify the circuit only from correct results in 8 explicit cases:

0	X	X	X	X	X	X
X	0	X	X	X	X	X
X	X	0	X	X	X	X
X	X	X	0	X	X	X
X	X	X	X	0	X	X
X	X	X	X	X	0	X
X	X	X	X	X	X	0
1	1	1	1	1	1	1

In a combined approach, we can parametrize these 8 test cases using just 3 Boolean variables (since $2^3 \geqslant 8$), say p, q and r. For example, we can represent the input wires as

$$a_0 = \begin{cases} 1 \text{ if } p \wedge q \wedge r \\ 0 \text{ if } \neg p \wedge \neg q \wedge \neg r \\ X \text{ otherwise} \end{cases}$$

and

$$a_1 = \begin{cases} 1 \text{ if } p \wedge q \wedge r \\ 0 \text{ if } \neg p \wedge \neg q \wedge r \\ X \text{ otherwise} \end{cases}$$

and so on until:

$$a_6 = \begin{cases} 1 \text{ if } p \wedge q \wedge r \\ 0 \text{ if } p \wedge q \wedge \neg r \\ X \text{ otherwise} \end{cases}$$

Symbolic trajectory evaluation (STE), introduced in (Seger and Bryant 1995), uses this kind of parametrization of circuit states in terms of Boolean variables, and a special logic for making correctness assertions, similar to but more restricted than the *temporal logics* we consider in the next lecture. Recently a more general form of STE known as *generalized symbolic trajectory evaluation* (GSTE) has been developed (Yang 2000). This can use STE-like methods to verify so-called ω-regular properties, optionally subject to fairness constraints, and so represents a substantial extension of its scope.

STE turns out to be very useful in partially automating the formal verification of some circuits. Part of the appeal is that the use of the ternary model gives a relatively easy way of abstracting out irrelevant detail. STE has been extensively used in formal verification at Intel. For one example, see (O'Leary, Zhao, Gerth, and Seger 1999). For a good introduction to the theory behind STE, see (Melham and Darbari 2002).

3. Reachability and model checking

We've mainly been concerned so far with analyzing combinational circuits, although symbolic simulation and STE can be used to track signals over any fixed finite number of clocks. However, it is sometimes of interest to ask questions like 'can a circuit *ever* get into a state where ...' or 'if line r (request) goes high at some point, must line a (acknowledge) eventually do so too?'. Dealing with queries over an unbounded time period like this requires the use of some new techniques.

We can abstract and generalize from the particular features of synchronous sequential digital circuits by considering them as particular cases of a *finite state transition system* or *finite state machine* (FSM). Such a system consists of a finite 'state space' S together with a binary relation $R \subseteq S \times S$ describing the possible transitions, where $R(a,b)$ is true iff it is possible for the system to make a transition from state a to state b in one time-step.[1]

For example, consider a circuit with three latches v_0, v_1 and v_2 that is supposed to implement a modulo-5 counter. The state of the system is described by the values of these latches, so we can choose $S = \{0,1\} \times \{0,1\} \times \{0,1\}$. The corresponding transition relation can be enumerated as follows:

$$\begin{aligned}
&(0,0,0) \to (0,0,1)\\
&(0,0,1) \to (0,1,0)\\
&(0,1,0) \to (0,1,1)\\
&(0,1,1) \to (1,0,0)\\
&(1,0,0) \to (0,0,0)
\end{aligned}$$

The transition systems arising from modelling circuits in this way have the special feature that the transition relation is deterministic, i.e. for each state a there is at most one state b such that $R(a,b)$. However, it's useful to consider state transition systems in general without this restriction, since the same methods can then be applied to a variety of other practically interesting situations, such as the analysis of parallel or nondeterministic hardware, programs or protocols. For example, it is often helpful in analyzing synchronization and mutual exclusion in systems with concurrent and/or interleaved components.

Forward and backward reachability

Many interesting questions are then of the form: if we start in a state $\sigma_0 \in S_0$, can we ever reach a state $\sigma_1 \in S_1$? For example, if we start the counter in state $(0,0,0)$, will it eventually return to the same state? (Fairly obviously, the answer is yes.) In general, we call questions like this *reachability* questions. In order to answer these questions, we can simply construct the reflexive-transitive closure R^* of the transition relation R, and see if

[1] Sometimes, one also considers a set of possible 'initial' and 'final' states to be part of the state transition system, but we prefer to consider the question of the starting and finishing states separately.

it links any pairs of states of interest. In principle, there is no difficulty, since the state space is finite.

Suppose that the set of starting states is S_0; we will consider this the first in an infinite sequence (S_i) of sets of states where S_i is the set of states reachable from S_0 in $\leqslant i$ transitions. Clearly we can compute S_{i+1} from S_i by the following recursion:

$$S_{i+1} = S_0 \cup \{b \mid \exists a \in S_i.\, R(a,b)\}$$

for a state is reachable in $\leqslant i+1$ steps if either it is in S_0, i.e. reachable in 0 steps, or it is a possible successor state b to a state a that is reachable in i steps. It is immediate from the intuitive interpretation that $S_i \subseteq S_{i+1}$ for all $i \geqslant 0$. Now, since each $S_i \subseteq S$, where S is finite, the sets S_i cannot properly increase forever, so we must eventually reach a stage where $S_{i+1} = S_i$ and hence $S_k = S_i$ for all $k \geqslant i$. It is easy to see that this S_i, which we will write S^*, is then precisely the set of states reachable from S_0 in any finite number of steps.

Thus, we have an algorithm, *forward reachability*, for deciding whether any states in some set $P \subseteq S$ are reachable from S_0. We simply compute S^* and then decide whether $S^* \cap P \neq \emptyset$. In the dual approach, *backward reachability*, we similarly compute the set of states P^* *from which* a state in P is reachable, and then ask whether $S_0 \cap P^* \neq \emptyset$. Once again we can compute the set P^* by iterating an operation until a fixpoint is reached. We can start with $A_0 = P$ and iterate:

$$A_{i+1} = P \cup \{a \mid \exists b \in A_i.\, R(a,b)\}$$

We can simply characterize A_i as the set of states from which a state in P is reachable in $\leqslant i$ steps. As before, we have $A_i \subseteq A_{i+1}$ and we must eventually reach a fixpoint that is the required set P^*. A formally appealing variant is to use $A_0 = \emptyset$ instead, and the first iteration will give $A_1 = P$ and thereafter we will obtain the same sequence offset by 1 and the same eventual fixpoint P^*. This helps to emphasize that we can consider this a case of finding the least fixpoint of the monotone mapping on sets of states $A \mapsto P \cup \{a \mid \exists b \in A.\, R(a,b)\}$ à la Knaster-Tarski (Knaster 1927; Tarski 1955).

Is there any reason to prefer forward or backward reachability over the other? Though they will always give the same answer if successfully carried out, one or the other may be much more efficient, depending on the particular system and properties concerned (Iwashita, Nakata, and Hirose 1996). We will follow the most common approach and focus on backward reachability. This is not because of efficiency concerns, but because, as we will see shortly, it generalizes nicely to analyzing more complicated 'future modalities' than just reachability.

Symbolic state representation

In practice, the kind of fixed-point computation sketched in the previous section can be impractical, because the enumeration of cases in the state set or transition relation is too large. Instead, we can use the techniques we have already established. Instead of representing sets of states and state transitions by explicit enumeration of cases, we can do it *symbolically*, using formulas. Suppose we encode states using Boolean variables

$v_1, \ldots, v_n$. Any $T \subseteq S$ can be identified with a Boolean formula $t[v_1, \ldots, v_n]$ that holds precisely when $(v_1, \ldots, v_n) \in T$. Moreover, by introducing n additional 'next state' variables $v'_1, \ldots, v'_n$ we can represent a binary relation A (such as the transition relation R) by a formula $a[v_1, \ldots, v_n, v'_1, \ldots, v'_n]$ that is true iff $A((v_1, \ldots, v_n), (v'_1, \ldots, v'_n))$. For example, we can represent the transition relation for the modulo-5 counter using (among other possibilities) the formula:

$$\begin{array}{l}(v'_0 \Leftrightarrow \neg v_0 \wedge \neg v_2) \wedge \\ (v'_1 \Leftrightarrow \neg(v_0 \Leftrightarrow v_1)) \wedge \\ (v'_2 \Leftrightarrow v_0 \wedge v_1)\end{array}$$

Here the parametrization in terms of Boolean variables just uses one variable for each latch, but this choice, while the most obvious, is not obligatory.

Bounded model checking

One immediate advantage of a symbolic state representation is that we can ask questions about k-step transition sequences without risking a huge blowup in the representation, as might happen in a canonical BDD representation used in symbolic simulation. We can simply duplicate the original formula k times with k sets of the n variables, and can ask any question about the relation between the initial and final states expressed as a Boolean formula $p[v_1^1, \ldots, v_n^1, v_1^k, \ldots, v_n^k]$:

$$\begin{array}{l}r[v_1^1, \ldots, v_n^1, v_1^2, \ldots, v_n^2] \wedge \cdots r[v_1^{k-1}, \ldots, v_n^{k-1}, v_1^k, \ldots, v_n^k] \\ \Rightarrow p[v_1^1, \ldots, v_n^1, v_1^k, \ldots, v_n^k\end{array}$$

This reduces the problem to tautology-checking. Of course, there is no guarantee that this is feasible in a reasonable amount of time, but experience on many practical problems with leading-edge tautology checkers has often showed impressive results for this so-called *bounded model checking* (Biere, Cimatti, Clarke, and Zhu 1999).

As presented above, bounded model checking permits one to answer only *bounded* reachability questions such as 'if we start in a state $\sigma_0 \in S_0$, can we reach a state $\sigma_1 \in S_1$ in k (or $\leqslant k$) cycles?'. Of course, every finite state transition system, precisely because the number of states is finite, has some characteristic 'diameter' d such that any sequence of d transition must contain a cycle. So it is possible in principle to decide unbounded reachability questions by testing bounded reachability for all $k \leqslant d$. In reality, d is invariably too large for this to be practical. But recently it has been shown (McMillan 2003) how to use interpolants generated from the propositional unsatisfiability proofs of successively longer reachability queries to find much sharper bounds, so that unbounded reachability is accessible by this technique. This promises to be a highly significant discovery, but we now turn to the more traditional technique for dealing with unbounded reachability queries.

BDD-based reachability

In the case of unbounded queries, the fixpoint computation can easily be performed directly on the symbolic representation. Traditionally, a canonical format such as BDDs has been used (Coudert, Berthet, and Madre 1989; Burch, Clarke, McMillan, Dill, and Hwang 1992; Pixley 1990). This can immediately cause problems if the BDD representation is infeasibly large, and indeed there have been recent explorations using non-canonical representations (Bjesse 1999; Abdulla, Bjesse, and Eén 2000). On the other hand, the fact that BDDs are a canonical representation means that if we repeat the iterative steps leading through a sequence S_i, we can immediately recognize when we have reached a fixed point simply by seeing if the BDD for S_{i+1} is identical to that for S_i, whereas in a non-canonical representation, we would need to perform a tautology check at each stage.

Most of the operations used to find the fixpoints are, in the symbolic context, simply logical operations. For example, union ('$\cup$') and intersection ('$\cap$') are simply implemented by conjunction ('$\wedge$') and disjunction ('$\vee$'). However, we also need the operation that maps a set P and a transition relation R to the set $Pre_R(P)$ (or just $Pre(P)$ when R is understood) of states from which there is a 1-step transition into a state in P:

$$Pre(P) = \{a \mid \exists b \in P. R(a,b)\}$$

In the symbolic representation, we can characterize this as follows:

$$Pre(p) = \exists v_1', \ldots, v_n'. r[v_1, \ldots, v_n, v_1', \ldots, v_n'] \wedge p[v_1', \ldots, v_n']$$

It is not difficult to implement this procedure on the BDD representation, though it is often a significant efficiency bottleneck.

Temporal logic model checking

There are still many interesting questions about transition systems that we can't answer using just reachability, the request-acknowledge example above being one simple case. But the material so far admits of generalization.

The backward reachability method answered a question 'starting in a state satisfying s, can we reach a state satisfying p?' by computing the set of all states p_* from which p is reachable, and then considering whether $p_* \wedge s = \bot$. If we introduce the notation:

$$EF(p)$$

to mean 'the state p is reachable', or 'there is some state on some path through the relation in which p holds', we can consider p_* as the set of states satisfying the formula $EF(p)$. We can systematically extend propositional logic in this way with further 'temporal operators' to yield a form of so-called *temporal logic*. The most popular logic in practice is *Computation Tree Logic* (CTL), introduced and popularized by Clarke and Emerson (Clarke and Emerson 1981).

One formal presentation of the semantics of CTL, using separate classes of 'path formulas' and 'state formulas' is the following. (We also need some way of assigning truth-values to primitive propositions based on a state σ, and we will write $[\![\sigma]\!]$ for this mapping applied to σ. In many cases, of course, the properties we are interested in will be stated precisely in terms of the variables used to encode the states, so a formula is actually its own denotation in the symbolic representation, modulo the transformation from CTL to pure propositional logic.) The valuation functions for an arbitrary formula are then defined as follows for state formulas:

$$
\begin{aligned}
sval(R,\sigma)(\bot) &= false \\
sval(R,\sigma)(\top) &= true \\
sval(R,\sigma)(\neg p) &= not(sval(R,\sigma)p) \\
sval(R,\sigma)p &= [\![\sigma]\!](p) \\
&\cdots \\
sval(R,\sigma)(p \Leftrightarrow q) &= (sval(R,\sigma)p = sval(R,\sigma)q) \\
sval(R,\sigma)(Ap) &= \forall\pi.\, Path(R,\sigma)\pi \Rightarrow pval(R,\pi)p \\
sval(R,\sigma)(Ep) &= \exists\pi.\, Path(R,\sigma)\pi \wedge pval(R,\pi)p
\end{aligned}
$$

and path formulas:

$$
\begin{aligned}
pval(R,\pi)(Fp) &= \exists t.\, sval(R,\pi(t))p \\
pval(R,\pi)(Gp) &= \forall t.\, sval(R,\pi(t))p \\
pval(R,\pi)(p \mathbin{\text{U}} q) &= \exists t.\, (\forall t'.\, t' < t \Rightarrow sval(R,\pi(t'))p) \wedge sval(R,\pi(t))q \\
pval(R,\pi)(Xp) &= sval(R,(t \mapsto \pi(t+1)))p
\end{aligned}
$$

Although it was convenient to use path formulas above, the appeal of CTL is exactly that we only really need to consider state formulas, with path formulas just an intermediate concept. We can eliminate the separate class of path formulas by just putting together the path quantifiers and temporal operators in combinations like '*EF*' and '*AG*'. A typical formula in this logic is the request-acknowledge property:

$$AG(a \Rightarrow EG(r))$$

The process of testing whether a transition system (defining a *model*) satisfies a temporal formula is called *model checking* (Clarke and Emerson 1981; Queille and Sifakis 1982). The process can be implemented by a very straightforward generalization of the fixpoint methods used in backwards reachability. The marriage of Clarke and Emerson's original explicit-state model-checking algorithms with a BDD-based symbolic representation, due to McMillan (Burch, Clarke, McMillan, Dill, and Hwang 1992), gives *sym-*

bolic model checking, and which has led to substantially wider practical applicability. Although at Intel, STE is often more useful, there are situations, e.g. the analysis of bus and cache protocols, where the greater generality of temporal logic model checking seems crucial.

For a detailed discussion of temporal logic model checking, see (Clarke, Grumberg, and Peled 1999). The topic is also covered in some books on logic in computer science like (Huth and Ryan 1999) and formal verification texts like (Kropf 1999). CTL is just one example of a temporal logic, and there are innumerable other varieties such as LTL (Pnueli 1977), CTL^* (Emerson and Halpern 1986) and the propositional μ-calculus (Kozen 1983). LTL stands for *linear temporal logic*, indicating that it takes a different point of view in its semantics, considering all paths and not permitting explicit 'A' and 'E' operators of CTL, but allowing temporal properties to be nested directly without intervening path quantifiers.

4. Automated theorem proving

It was noted that one of the reasons for the greater success of formal verification in the hardware domain is the fact that more of the verification task can be automated. However, for some more elaborate systems, we cannot rely on tools like equivalence checkers, symbolic simulators or model checkers. One potential obstacle — almost invariably encountered on large industrial designs — is that the system is too large to be verified in a feasible time using such methods. A more fundamental limitation is that some more sophisticated properties cannot be expressed at all in the simple Boolean world. For example, a floating-point *sin* function can hardly have its intended behaviour spelled out in terms of Boolean formulas on the component bits. Instead, we need a more expressive logical system where more or less any mathematical concepts, including real numbers and infinite sets, can be analyzed. This is usually formulated in first-order or higher-order logic, which, as well as allowing propositions to have internal structure, introduces the universal and existential quantifiers:

- The formula $\forall x.\ p$, where x is an (object) variables and p any formula, means intuitively 'p is true for *all* values of x'.
- The analogous formula $\exists x.\ p$ means intuitively 'p is true for *some* value(s) of x', or in other words 'there exists an x such that p'.

I will assume that the basic syntax and definitions of first order logic are known.

Automated theorem proving based on Herbrand's theorem

In contrast to propositional logic, many interesting questions about first order and higher order logic are undecidable even in principle, let alone in practice. Church (Church 1936) and Turing (Turing 1936) showed that even pure logical validity in first order logic is undecidable, introducing in the process many of the basic ideas of computability theory. On the other hand, it is not too hard to see that logical validity is *semidecidable* — this is certainly a direct consequence of completeness theorems for proof systems in first order logic (Gödel 1930), and was arguably implicit in work by Skolem (Skolem 1922). This means that we can at least program a computer to enumerate all valid first order formulas.

One simple approach is based on the following logical principle, due to Skolem and Gödel but usually mis-named "Herbrand's theorem":

> Let $\forall x_1, \ldots, x_n.\, P[x_1, \ldots, x_n]$ be a first order formula with only the indicated universal quantifiers (i.e. the body $P[x_1, \ldots, x_n]$ is quantifier-free). Then the formula is satisfiable iff the infinite set of 'ground instances' $p[t_1^i, \ldots, t_n^i]$ that arise by replacing the variables by arbitrary variable-free terms made up from functions and constants in the original formula is *propositionally* satisfiable.

We can get the original formula into the special form required by some simple normal form transformations, introducing Skolem functions to replace existentially quantified variables. And by compactness for propositional logic, we know that if the infinite set of instances is unsatisfiable, then so will be some finite subset. In principle we can enumerate *all* possible sets, one by one, until we find one that is not propositionally satisfiable. (If the formula is satisfiable, we will never discover it by this means. By undecidability, we know this is unavoidable.) A precise description of this procedure is tedious, but a simple example may help. Suppose we want to prove that the following is valid. This is often referred to as the 'drinker's principle', because you can think of it as asserting that there is some person x such that if x drinks, so does everyone.

$$\exists x.\, \forall y.\, D(x) \Rightarrow D(y)$$

We start by negating the formula. To prove that the original is valid, we need to prove that this is unsatisfiable:

$$\neg(\exists x.\, \forall y.\, D(x) \Rightarrow D(y))$$

We then make some transformations to a logical equivalent so that it is in 'prenex form' with all quantifiers at the front.

$$\forall x.\, \exists y.\, D(x) \land \neg D(y)$$

We then introduce a Skolem function f for the existentially quantified variable y:

$$\forall x.\, D(x) \land \neg D(f(x))$$

We now consider the *Herbrand* universe, the set of all terms built up from constants and functions in the original formula. Since here we have no nullary constants, we need to add one c to get started (this effectively builds in the assumption that all models have a non-empty domain). The Herbrand universe then becomes $\{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$. By Herbrand's theorem, we need to test all sets of ground instances for propositional satisfiability. Let us enumerate them in increasing size. The first one is:

$$D(c) \wedge \neg D(f(c))$$

This is not propositionally unsatisfiable, so we consider the next:

$$(D(c) \wedge \neg D(f(c))) \wedge (D(f(c)) \wedge \neg D(f(f(c)))$$

Now this is propositionally unsatisfiable, so we terminate with success.

Unification-based methods

The above idea (Robinson 1957) led directly some early computer implementations, e.g. by Gilmore (Gilmore 1960). Gilmore tested for propositional satisfiability by transforming the successively larger sets to disjunctive normal form. A more efficient approach is to use the Davis-Putnam algorithm — it was in this context that it was originally introduced (Davis and Putnam 1960). However, as Davis (Davis 1983) admits in retrospect:

> . . . effectively eliminating the truth-functional satisfiability obstacle only uncovered the deeper problem of the combinatorial explosion inherent in unstructured search through the Herbrand universe . . .

The next major step forward in theorem proving was a more intelligent means of choosing substitution instances, to pick out the small set of relevant instances instead of blindly trying all possibilities. The first hint of this idea appears in (Prawitz, Prawitz, and Voghera 1960), and it was systematically developed by Robinson (Robinson 1965), who gave an effective syntactic procedure called *unification* for deciding on appropriate instantiations to make terms match up correctly.

There are many unification-based theorem proving algorithms. Probably the best-known is *resolution*, in which context Robinson (Robinson 1965) introduced full unification to automated theorem proving. Another important method quite close to resolution and developed independently at about the same time is the inverse method (Maslov 1964; Lifschitz 1986). Other popular algorithms include tableaux (Prawitz, Prawitz, and Voghera 1960), model elimination (Loveland 1968; Loveland 1978) and the connection method (Kowalski 1975; Bibel and Schreiber 1975; Andrews 1976). Crudely speaking:

- Tableaux = Gilmore procedure + unification
- Resolution = Davis-Putnam procedure + unification

Tableaux and resolution can be considered as classic representatives of 'top-down' and 'bottom-up' methods respectively. Roughly speaking, in top-down methods one starts from a goal and works backwards, while in bottom-up methods one starts from the assumptions and works forwards. This has significant implications for the very nature of unifiable variables, since in bottom-up methods they are local (implicitly universally quantified) whereas in top-down methods they are global, correlated in different portions of the proof tree. This is probably the most useful way of classifying the various first-order search procedures and has a significant impact on the problems where they perform well.

4.1. Decidable problems

Although first order validity is undecidable in general, there are special classes of formulas for which it is decidable, e.g.

- AE formulas, which involve no function symbols and when placed in prenex form have all the universal quantifiers before the existential ones.
- Monadic formulas, involving no function symbols and only monadic (unary) relation symbols.
- Purely universal formulas

The decidability of AE formulas is quite easy to see, because no function symbols are there to start with, and because of the special quantifier nesting, none are introduced in Skolemization. Therefore the Herbrand universe is finite and the enumeration of ground instances cannot go on forever. The decidability of the monadic class can be proved in various ways, e.g. by transforming into AE form by pushing quantifiers inwards ('miniscoping'). Although neither of these classes is particularly useful in practice, it's worth noting that the monadic formulas subsume traditional Aristotelian syllogisms, at least on a straightforward interpretation of what they are supposed to mean. For example

If all M are P, and all S are M, then all S are P

can be expressed using monadic relations as follows:

$$(\forall x.\, M(x) \Rightarrow P(x)) \land (\forall x.\, S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\, S(x) \Rightarrow P(x))$$

For purely universal formulas, we can use *congruence closure* (Nelson and Oppen 1980; Shostak 1978; Downey, Sethi, and Tarjan 1980). This allows us to prove that one equation follows from others, e.g. that

$$\forall x.\, f(f(f(x)) = x \land f(f(f(f(f(x))))) = x \Rightarrow f(x) = x$$

As the name implies, congruence closure involves deducing all equalities between subterms that follow from the asserted ones by using equivalence and congruence properties of equality. In our case, for example, the first equation $f(f(f(x)) = x$ implies $f(f(f(f(x))) = f(x)$ and hence $f(f(f(f(f(x))))) = f(f(x))$, and then symmetry and transitivity with the second equation imply $f(f(x)) = x$, and so on. It straightforwardly extends to deciding the entire universal theory by refutation followed by DNF transformation and congruence closure on the equations in the disjuncts, seeing whether any negated equations in the same disjunct are implied.

An alternative approach is to reduce the problem to SAT by introducing a propositional variable for each equation between subterms, adding constraints on these variables to reflect congruence properties. This has the possibly significant advantage that no potentially explosive DNF transformation need be done, and exploits the power of SAT solvers. For example if $E_{m,n}$ (for $0 \leqslant m,n \leqslant 5$) represents the equation $f^m(x) = f^n(x)$, we want to deduce $E_{3,0} \land E_{5,0} \Rightarrow E_{1,0}$ assuming the equality properties $\bigwedge_n E_{n,n}$ (reflexivity), $\bigwedge_{m,n} E_{m,n} \Rightarrow E_{n,m}$ (symmetry), $\bigwedge_{m,n,p} E_{m,n} \land E_{n,p} \Rightarrow E_{m,p}$ (transitivity) and $\bigwedge_{m,n} E_{m,n} \Rightarrow E_{m+1,n+1}$ (congruence).

4.2. *Theories*

Also interesting in practice are situations where, rather than absolute logical validity, we want to know whether statements follow from some well-accepted set of mathematical axioms, or are true in some particular interpretation like the real numbers $\mathbb{R}$. We generalize the notion of logical validity, and say that p is a logical consequence of axioms A, written $A \models p$, if for any valuation, every interpretation that makes all the formulas in A true also makes p true. (This looks straightforward but unfortunately there is some inconsistency in standard texts. For example the above definition is found in (Enderton 1972), whereas in (Mendelson 1987) the definition has the quantification over valuations per formula: every interpretation in which each formula of A is true in all valuations makes p true in all valuations. Fortunately, in most cases all the formulas in A are sentences, formulas with no free variables, and then the two definitions coincide.) Ordinary validity of p is the special cases $\emptyset \models p$, usually written just $\models p$.

By a *theory*, we mean a set of formulas closed under first-order validity, or in other words, the set of logical consequences of a set of axioms. The smallest theory is the set of consequences of the empty set of axioms, i.e. the set of logically valid formulas. Note also that for any particular interpretation, the set of formulas true in that interpretation is also a theory. Some particularly important characteristics of a theory are:

- Whether it is *consistent*, meaning that we never have both $T \models p$ and $T \models \neg p$. (Equivalently, that we do not have $T \models \bot$, or that *some* formula does not follow from T.)
- Whether it is *complete*, meaning that for any sentence p, either $T \models p$ or $T \models \neg p$.
- Whether it is *decidable*, meaning that there is an algorithm that takes as input a formula p and decides whether $T \models p$.

Note that since we have a semidecision procedure for first-order validity, any complete theory based on a finite (or even semicomputable, with a slightly more careful analysis) set of axioms is automatically decidable: just search in parallel for proofs of $A \Rightarrow p$ and $A \Rightarrow \neg p$.

5. Arithmetical theories

First order formulas built up from equations and inequalities and interpreted over common number systems are often decidable. A common way of proving this is *quantifier elimination*.

5.1. *Quantifier elimination*

We say that a theory T in a first-order language L *admits quantifier elimination* if for each formula p of L, there is a quantifier-free formula q such that $T \models p \Leftrightarrow q$. (We assume that the equivalent formula contains no new free variables.) For example, the well-known criterion for a quadratic equation to have a (real) root can be considered as an example of quantifier elimination in a suitable theory T of reals:

$$T \models (\exists x.\ ax^2 + bx + c = 0) \Leftrightarrow a \neq 0 \wedge b^2 \geqslant 4ac \vee a = 0 \wedge (b \neq 0 \vee c = 0)$$

If a theory admits quantifier elimination, then in particular any closed formula (one with no free variables, such as $\forall x.\, \exists y.\, x < y$) has a T-equivalent that is ground, i.e. contains no variables at all. In many cases of interest, we can quite trivially decide whether a ground formula is true or false, since it just amounts to evaluating a Boolean combination of arithmetic operations applied to constants, e.g. $2 < 3 \Rightarrow 4^2 + 5 < 23$. (One interesting exception is the theory of algebraically closed fields of unspecified characteristic, where quantifiers can be eliminated but the ground formulas cannot in general be evaluated without knowledge about the characteristic.) Consequently quantifier elimination in such cases yields a decision procedure, and also shows that such a theory T is complete, i.e. every closed formula can be proved or refuted from T. For a good discussion of quantifier elimination and many explicit examples, see (Kreisel and Krivine 1971). One of the simplest examples is the theory of 'dense linear (total) orders without end points' is based on a language containing the binary relation '$<$' as well as equality, but no function symbols. It can be axiomatized by the following set of sentences:

$$
\begin{array}{l}
\forall x\, y.\, x = y \lor x < y \lor y < x \\
\forall x\, y\, z.\, x < y \land y < z \Rightarrow x < z \\
\forall x.\, x \not< x \\
\forall x\, y.\, x < y \Rightarrow \exists z.\, x < z \land z < y \\
\forall x.\, \exists y.\, x < y \\
\forall x.\, \exists y.\, y < x
\end{array}
$$

5.2. Presburger arithmetic

One of the earliest theories shown to have quantifier elimination is linear arithmetic over the natural numbers or integers, as first shown by Presburger (Presburger 1930). Linear arithmetic means that we are allows the usual equality and inequality relations, constants and addition, but no multiplication, except by constants. In fact, to get quantifier elimination we need to add infinitely many divisibility relations D_k for all integers $k \geqslant 2$. This doesn't affect decidability because those new relations are decidable for particular numbers.

Presburger's original algorithm is fairly straightforward, and follows the classic quantifier elimination pattern of dealing with the special case of an existentially quantified conjunction of literals. (We can always put the formula into disjunctive normal form and distribute existential quantifiers over the disjuncts, then start from the innermost quantifier.) For an in-depth discussion of Presburger's original procedure, the reader can consult (Enderton 1972) and (Smoryński 1980), or indeed the original article, which is quite readable — (Stansifer 1984) gives an annotated English translation. A somewhat more efficient algorithm more suited to computer implementation is given by Cooper (Cooper 1972).

5.3. Complex numbers

Over the complex numbers, we can also allow multiplication and still retain quantifier elimination. Here is a sketch of a naive algorithm for complex quantifier elimination. By the usual quantifier elimination pre-canonization, it suffices to be able to eliminate a single existential quantifier from a conjunction of positive and negative equations:

$$\exists x.\, p_1(x) = 0 \wedge \cdots \wedge p_n(x) = 0 \wedge q_1(x) \neq 0 \wedge \cdots q_m(x) \neq 0$$

We'll sketch now how this can be reduced to the $m \leqslant 1$ and $n \leqslant 1$ case. To reduce n we can use one equation to reduce the powers of variables in the others by elimination, e.g.

$$\begin{aligned} 2x^2 + 5x + 3 = 0 \wedge x^2 - 1 = 0 &\Leftrightarrow 5x + 5 = 0 \wedge x^2 - 1 = 0 \\ &\Leftrightarrow 5x + 5 = 0 \wedge 0 = 0 \\ &\Leftrightarrow 5x + 5 = 0 \end{aligned}$$

To reduce m, we may simply multiply all the $q_i(x)$ together since $q_i(x) \neq 0 \wedge q_{i+1}(x) \neq 0 \Leftrightarrow q_i(x) \cdot q_{i+1}(x) \neq 0$. Now, the problem that remains is:

$$\exists x.\, p(x) = 0 \wedge q(x) \neq 0$$

or equivalently $\neg(\forall x.\, p(x) = 0 \Rightarrow q(x) = 0)$. Consider the core formula:

$$\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$$

Assume that neither $p(x)$ nor $q(x)$ is the zero polynomial. Since we are working in an algebraically closed field, we know that the polynomials $p(x)$ and $q(x)$ split into linear factors whatever they may be:

$$\begin{aligned} p(x) &= (x - a_1) \cdot (x - a_2) \cdot \cdots \cdot (x - a_n) \\ q(x) &= (x - b_1) \cdot (x - b_2) \cdot \cdots \cdot (x - b_m) \end{aligned}$$

Now $p(x) = 0$ is equivalent to $\bigvee_{1 \leqslant i \leqslant n} x = a_i$ and $q(x) = 0$ is equivalent to $\bigvee_{1 \leqslant j \leqslant m} x = b_j$. Thus, the formula $\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$ says precisely that

$$\forall x. \bigvee_{1 \leqslant i \leqslant n} x = a_i \Rightarrow \bigvee_{1 \leqslant j \leqslant m} x = b_j$$

or in other words, all the a_i appear among the b_j. However, since there are just n linear factors in the antecedent, a given factor $(x - a_i)$ cannot occur more than n times and thus the polynomial divisibility relation $p(x)|q(x)^n$ holds. Conversely, if this divisibility relation holds for $n \neq 0$, then clearly $\forall x.\, p(x) = 0 \Rightarrow q(x) = 0$ holds. Thus, the key quantified formula can be reduced to a polynomial divisibility relation, and it's not difficult to express this as a quantifier-free formula in the coefficients, thus eliminating the quantification over x.

5.4. Real algebra

In the case of the real numbers, one can again use addition and multiplication arbitrarily and it is decidable whether the formula holds in $\mathbb{R}$. A simple (valid) example is a case of the Cauchy-Schwartz inequality:

$$\forall x_1\, x_2\, y_1\, y_2.\, (x_1 \cdot y_1 + x_2 \cdot y_2)^2 \leqslant (x_1^2 + x_2^2) \cdot (y_1^2 + y_2^2)$$

This decidability result is originally due to Tarski (Tarski 1951), though Tarski's method has non-elementary complexity and has apparently never been implemented. Perhaps the most efficient general algorithm currently known, and the first actually to be implemented on a computer, is the Cylindrical Algebraic Decomposition (CAD) method introduced by Collins (Collins 1976). (For related work around the same time see (Łojasiewicz 1964) and the method, briefly described in (Rabin 1991), developed in Monk's Berkeley PhD thesis.) A simpler method, based on ideas by Cohen, was developed by Hörmander (Hörmander 1983) — see also (Bochnak, Coste, and Roy 1998) and (Gårding 1997) — and it is this algorithm that we will sketch.

Consider first the problem of eliminating the quantifier from $\exists x.\, P[x]$, where $P[x]$ is a Boolean combination of univariate polynomials (in x). Suppose the polynomials involved in the body are $p_1(x), \ldots, p_n(x)$. The key to the algorithm is to obtain a *sign matrix* for the set of polynomials. This is a division of the real line into a (possibly empty) ordered sequence of m points $x_1 < x_2 < \cdots < x_m$ representing precisely the zeros of the polynomials, with the rows of the matrix representing, in alternating fashion, the points themselves and the intervals between adjacent pairs and the two intervals at the ends:

$$(-\infty, x_1), x_1, (x_1, x_2), x_2, \ldots, x_{m-1}, (x_{m-1}, x_m), x_m, (x_m, +\infty)$$

and columns representing the polynomials $p_1(x), \ldots, p_n(x)$, with the matrix entries giving the signs, either positive $(+)$, negative $(-)$ or zero (0), of each polynomial p_i at the points and on the intervals. For example, for the collection of polynomials:

$$p_1(x) = x^2 - 3x + 2$$
$$p_2(x) = 2x - 3$$

the sign matrix looks like this:

Point/Interval	p_1	p_2
$(-\infty, x_1)$	$+$	$-$
x_1	0	$-$
(x_1, x_2)	$-$	$-$
x_2	$-$	0
(x_2, x_3)	$-$	$+$
x_3	0	$+$
$(x_3, +\infty)$	$+$	$+$

Note that x_1 and x_3 represent the roots 1 and 2 of $p_1(x)$ while x_2 represents 1.5, the root of $p_2(x)$. However the sign matrix contains no numerical information about the location of the points x_i, merely specifying the order of the roots of the various polynomials and what signs they take there and on the intervening intervals. It is easy to see that the sign matrix for a set of univariate polynomials $p_1(x), \ldots, p_n(x)$ is sufficient to answer any question of the form $\exists x.\, P[x]$ where the body $P[x]$ is quantifier-free and all atoms are of the form $p_i(x) \bowtie_i 0$ for any of the relations $=$, $<$, $>$, $\leqslant$, $\geqslant$ or their negations.

We simply need to check each row of the matrix (point or interval) and see if one of them makes each atomic subformula true or false; the formula as a whole can then simply be "evaluated" by recursion.

In order to perform general quantifier elimination, we simply apply this basic operation to all the innermost quantified subformulas first (we can consider a universally quantified formula $\forall x. P[x]$ as $\neg(\exists x. \neg P[x])$ and eliminate from $\exists x. \neg P[x]$). This can then be iterated until all quantifiers are eliminated. The only difficulty is that the coefficients of a polynomial may now contain other variables as parameters. But we can quite easily handle these by performing case-splits over the signs of coefficients and using pseudo-division of polynomials instead of division as we present below.

So the key step is finding the sign matrix, and for this the following simple observation is key. To find the sign matrix for

$$p, p_1, \ldots, p_n$$

it suffices to find one for the set of polynomials

$$p', p_1, \ldots, p_n, q_0, q_1, \ldots, q_n$$

where p', which we will sometimes write p_0 for regularity's sake, is the derivative of p, and q_i is the remainder on dividing p by p_i. For suppose we have a sign matrix for the second set of polynomials. We can proceed as follows.

First, we split the sign matrix into two equally-sized parts, one for the $p', p_1, \ldots, p_n$ and one for the $q_0, q_1, \ldots, q_n$, but for now keeping all the points in each matrix, even if the corresponding set of polynomials has no zeros there. We can now infer the sign of $p(x_i)$ for each point x_i that is a zero of one of the polynomials $p', p_1, \ldots, p_n$, as follows. Since q_k is the remainder of p after division by p_k, $p(x) = s_k(x)p_k(x) + q_k(x)$ for some $s_k(x)$. Therefore, since $p_k(x_i) = 0$ we have $p(x_i) = q_k(x_i)$ and so we can derive the sign of p at x_i from that of the corresponding q_k.

Now we can throw away the second sign matrix, giving signs for the $q_0, \ldots, q_n$, and retain the (partial) matrix for $p, p', p_1, \ldots, p_n$. We next 'condense' this matrix to remove points that are not zeros of one of the $p', p_1, \ldots, p_n$, but only of one of the q_i. The signs of the $p', p_1, \ldots, p_n$ in an interval from which some other points have been removed can be read off from any of the subintervals in the original subdivision — they cannot change because there are no zeros for the relevant polynomials there.

Now we have a sign matrix with correct signs at all the points, but undetermined signs for p on the intervals, and the possibility that there may be additional zeros of p inside these intervals. But note that since there are certainly no zeros of p' inside the intervals, there can be at most one additional root of p in each interval. Whether there is one can be inferred, for an internal interval (x_i, x_{i+1}), by seeing whether the signs of $p(x_i)$ and $p(x_{i+1})$, determined in the previous step, are both nonzero and are different. If not, we can take the sign on the interval from whichever sign of $p(x_i)$ and $p(x_{i+1})$ is nonzero (we cannot have them both zero, since then there would have to be a zero of p' in between). Otherwise we insert a new point y between x_i and x_{i+1} which is a zero (only) of p, and infer the signs on the new subintervals (x_i, y) and (y, x_{i+1}) from the signs at the endpoints. Other polynomials have the same signs on (x_i, y), y and (y, x_{i+1}) that had been inferred for the original interval (x_i, x_{i+1}). For external intervals, we can use the same reasoning if we temporarily introduce new points $-\infty$ and $+\infty$ and infer the sign of $p(-\infty)$ by flipping the sign of p' on the lowest interval $(-\infty, x_1)$ and the sign of $p(+\infty)$ by copying the sign of p' on the highest interval $(x_n, +\infty)$.

5.5. *Word problems*

Suppose K is a class of structures, e.g. all groups. The *word problem* for K asks whether a set E of equations between constants implies another such equation $s = t$ in all algebraic structures of class K. More precisely, we may wish to distinguish:

- The uniform word problem for K: deciding given any E and $s = t$ whether $E \models_M s = t$ for all interpretations M in K.
- The word problem for K, E: with E fixed, deciding given any $s = t$ whether $E \models_M s = t$ for all interpretations M in K.
- The free word problem for K: deciding given any $s = t$ whether $\models_M s = t$ for all interpretations M in K.

As a consequence of general algebraic results (e.g. every integral domain has a field of fractions, every field has an algebraic closure), there is a close relationship between word problems and results for particular interpretations. For example, for any universal formula in the language of rings, such as a word problem implication $\bigwedge_i s_i = t_i \Rightarrow s = t$, the following are equivalent, and hence we can solve it using complex quantifier elimination:

- It holds in all integral domains of characteristic 0
- It holds in all fields of characteristic 0
- It holds in all algebraically closed fields of characteristic 0
- It holds in any given algebraically closed field of characteristic 0
- It holds in $\mathbb{C}$

There is also a close relationship between word problems and ideal membership questions (Scarpellini 1969; Simmons 1970), sketched in a later section. These ideal membership questions can be solved using efficient methods like Gröbner bases (Buchberger 1965; Cox, Little, and O'Shea 1992).

5.6. *Practical decision procedures*

In many 'practical' applications of decision procedures, a straightforward implementation of one of the above quantifier elimination procedures may be a poor fit, in both a positive and negative sense:

- Fully general quantifier elimination is not needed
- The decidable theory must be combined with others

For example, since most program verification involves reasoning about integer inequalities (array indices etc.), one might think that an implementation of Presburger arithmetic is appropriate. But in practice, most of the queries (a) are entirely universally quantified, and (b) do not rely on subtle divisibility properties. A much faster algorithm can be entirely adequate. In some cases the problems are even more radically limited. For example, (Ball, Cook, Lahriri, and Rajamani 2004) report that the majority of their integer inequality problems fall into a very class with at most two variables per inequality and coefficients of only ± 1, for which a much more efficient decision procedure is available (Harvey and Stuckey 1997). Of course there are exceptions, with some applications such as indexed predicate abstraction (Lahiri and Bryant 2004) demanding more general quantifier prefixes.

It's quite common in program verification to need a combination of a decidable theory, or indeed more than one, with uninterpreted function symbols. For example, in typical correctness theorems for loops, one can end up with problems like the following (the antecedent comes from a combination of the loop invariant and the loop termination condition):

$$x - 1 < n \land \neg(x < n) \Rightarrow a[x] = a[n]$$

In pure Presburger arithmetic one can deduce $x - 1 < n \land \neg(x < n) \Rightarrow x = n$, but one needs a methodology for making the further trivial deduction $x = n \Rightarrow a[x] = a[n]$. For non-trivial quantifier prefixes, this problem rapidly becomes undecidable, but for purely universal formulas like the above, there are well-established methods. The Nelson-Oppen (Nelson and Oppen 1979) approach is the most general. It exploits the fact that the only communication between component procedures need be equations and negated equations ($s = t$ and $s \neq t$), by virtue of a result in logic known as the *Craig interpolation theorem*. An alternative, which can be viewed as an optimization of Nelson-Oppen for some common cases, is due to Shostak (Shostak 1984). It has taken a remarkably long time to reach a rigorous understanding of Shostak's method; indeed Reuß and Shankar (Reuß and Shankar 2001) showed that Shostak's original algorithm and all the then known later refinements were in fact incomplete and potentially nonterminating!

At the time of writing, there is intense interest in decision procedures for combinations of (mainly, but not entirely quantifier-free) theories. The topic has become widely known as *satisfiability modulo theories* (SMT), emphasizing the perspective that it is a generalization of the standard propositional SAT problem. Indeed, most of the latest SMT systems use methods strongly influenced by the leading SAT solvers, and are usually organized around a SAT-solving core. The idea of basing other decision procedures around SAT appeared in several places and in several slightly different contexts, going back at least to (Armando, Castellini, and Giunchiglia 1999). The simplest approach is to use the SAT checker as a 'black box' subcomponent. Given a formula to be tested for satisfiability, just treat each atomic formula as a propositional atom and feed the formula to the SAT checker. If the formula is propositionally unsatisfiable, then it is trivially unsatisfiable as a first-order formula and we are finished. If on the other hand the SAT solver returns a satisfying assignment for the propositional formula, test whether the implicit conjunction of literals is also satisfiable within our theory or theories. If it is satisfiable, then we can conclude that so is the whole formula and terminate. However, if the putative satisfying valuation is *not* satisfiable in our theories, we conjoin its negation with the input formula and repeat the procedure.

6. Interactive theorem proving

Even though first order validity is semi-decidable, it is seldom practical to solve interesting problems using unification-based approaches to pure logic. Nor is it the case that practical problems often fit conveniently into one of the standard decidable subsets. The best we can hope for in most cases is that the human will have to guide the proof process, but the machine may be able to relieve the tedium by filling in gaps, while always ensuring that no mistakes are made. This kind of application was already envisaged by Wang (Wang 1960)

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous than *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

The first notable interactive provers were the SAM (semi-automated mathematics) series. In 1966, the fifth in the series, SAM V, was used to construct a proof of a hitherto unproven conjecture in lattice theory (Bumcrot 1965). This was indubitably a success for the semi-automated approach because the computer automatically proved a result now called "SAM's Lemma" and the mathematician recognized that it easily yielded a proof of Bumcrot's conjecture.

Not long after the SAM project, the AUTOMATH (de Bruijn 1970; de Bruijn 1980) and Mizar (Trybulec 1978; Trybulec and Blair 1985) proof checking systems appeared, and each of them in its way has been profoundly influential. Although we will refer to these systems as 'interactive', we use this merely as an antonym of 'automatic'. In fact, both AUTOMATH and Mizar were oriented around batch usage. However, the files that they process consist of a *proof*, or a proof sketch, which they *check* the correctness of, rather than a statement for which they attempt to find a proof automatically.

Mizar has been used to proof-check a very large body of mathematics, spanning pure set theory, algebra, analysis, topology, category theory and various unusual applications like mathematical puzzles and computer models. The body of mathematics formally built up in Mizar, known as the 'Mizar Mathematical Library' (MML), seems unrivalled in any other theorem proving system. The 'articles' (proof texts) submitted to the MML are automatically abstracted into human-readable form and published in the *Journal of Formalized Mathematics*, which is devoted entirely to Mizar formalizations.[2]

LCF — a programmable proof checker

The ideal proof checker should be *programmable*, i.e. users should be able to extend the built-in automation as much as desired. There's no particular difficulty in allowing this. Provided the basic mechanisms of the theorem prover are straightforward and well-documented and the source code is made available, there's no reason why a user shouldn't extend or modify it. However, the difficulty comes if we want to restrict the user to extensions that are logically sound — as presumably we might well wish to, since unsoundness renders questionable the whole idea of machine-checking of supposedly more fallible human proofs. Even fairly simple automated theorem proving programs are often subtler than they appear, and the difficulties of integrating a large body of special proof methods into a powerful interactive system without compromising soundness is not trivial.

[2]Available on the Web via `http://www.mizar.org/JFM`.

One influential solution to this difficulty was introduced in the Edinburgh LCF project led by Robin Milner (Gordon, Milner, and Wadsworth 1979). Although this was for an obscure 'logic of computable functions' (hence the name LCF), the key idea, as Gordon (Gordon 1982) emphasizes, is equally applicable to more orthodox logics supporting conventional mathematics, and subsequently many programmable proof checkers were designed using the same principles, such as Coq,[3] HOL (Gordon and Melham 1993), Isabelle (Paulson 1994) and Nuprl (Constable 1986).

The key LCF idea is to use a special type (say `thm`) of proven theorems in the implementation language, so that anything of type `thm` must by construction have been *proved* rather than simply asserted. (In practice, the implementation language is usually a version of ML, which was specially designed for this purpose in the LCF project.) This is enforced by making `thm` an *abstract type* whose only constructors correspond to approved inference rules. But the user is given full access to the implementation language and can put the primitive rules together in more complicated ways using arbitrary programming. Because of the abstract type, any result of type `thm`, however it was arrived at, must ultimately have been produced by correct application of the primitive rules. Yet the means for arriving at it may be complex. Most obviously, we can use ML as a kind of 'macro language' to automate common patterns of inference. But much more sophisticated derived rules can be written that, for example, prove formulas of Presburger arithmetic while automatically decomposing to logical primitives. In many theorem-proving tasks, more 'ad hoc' manipulation code can be replaced by code performing inference without significant structural change. In other cases we can use an automated proof procedure, or even an external system like a computer algebra system (Harrison and Théry 1998), as an oracle to find a proof that is later checked inside the system. Thus, LCF gives a combination of programmability and logical security that would probably be difficult to assure by other means.

An LCF kernel for first-order logic

To explain the LCF idea in more concrete terms, we will show a complete LCF-style kernel for first order logic with equality, implemented in Objective CAML.[4] We start by defining the syntax of first order logic, as found in any standard text like (Enderton 1972; Mendelson 1987). First we have an OCaml type of first order terms, where a term is either a named variable or a named function applied to a list of arguments (constants like 1 are regarded as nullary functions):

```
type term = Var of string | Fn of string * term list;;
```

Using this, we can now define a type of first order formulas:

```
type formula = False
             | True
             | Atom of string * term list
             | Not of formula
             | And of formula * formula
             | Or of formula * formula
             | Imp of formula * formula
```

[3]See the Coq Web page `http://pauillac.inria.fr/coq`.

[4]See `http://www.ocaml.org`.

```
          | Iff of formula * formula
          | Forall of string * formula
          | Exists of string * formula;;
```

Before proceeding, we define some OCaml functions for useful syntax operations: constructing an equation, checking whether a term occurs in another, and checking whether a term occurs free in a formula:

```
let mk_eq s t = Atom("=",[s;t]);;

let rec occurs_in s t =
  s = t or
  match t with
    Var y -> false
  | Fn(f,args) -> exists (occurs_in s) args;;

let rec free_in t fm =
  match fm with
    False -> false
  | True -> false
  | Atom(p,args) -> exists (occurs_in t) args
  | Not(p) -> free_in t p
  | And(p,q) -> free_in t p or free_in t q
  | Or(p,q) -> free_in t p or free_in t q
  | Imp(p,q) -> free_in t p or free_in t q
  | Iff(p,q) -> free_in t p or free_in t q
  | Forall(y,p) -> not (occurs_in (Var y) t) & free_in t p
  | Exists(y,p) -> not (occurs_in (Var y) t) & free_in t p;;
```

There are many complete proof systems for first order logic. We will adopt a Hilbert-style proof system close to one first suggested by Tarski (Tarski 1965), and subsequently presented in a textbook (Monk 1976). The idea is to avoid defining relatively tricky syntactic operations like substitution. We first define the signature for the OCaml abstract datatype of theorems:

```
module type Proofsystem =
   sig type thm
       val axiom_addimp : formula -> formula -> thm
       val axiom_distribimp :
            formula -> formula -> formula -> thm
       val axiom_doubleneg : formula -> thm
       val axiom_allimp : string -> formula -> formula -> thm
       val axiom_impall : string -> formula -> thm
       val axiom_existseq : string -> term -> thm
       val axiom_eqrefl : term -> thm
       val axiom_funcong : string -> term list -> term list -> thm
       val axiom_predcong : string -> term list -> term list -> thm
       val axiom_iffimp1 : formula -> formula -> thm
       val axiom_iffimp2 : formula -> formula -> thm
       val axiom_impiff : formula -> formula -> thm
       val axiom_true : thm
       val axiom_not : formula -> thm
       val axiom_or : formula -> formula -> thm
       val axiom_and : formula -> formula -> thm
       val axiom_exists : string -> formula -> thm
```

```
      val modusponens : thm -> thm -> thm
      val gen : string -> thm -> thm
      val concl : thm -> formula
   end;;
```

and then the actual implementation of the primitive inference rules. For example, `modusponens` is the traditional *modus ponens* inference rule allowing us to pass from two theorems of the form $\vdash p \Rightarrow q$ and $\vdash p$ to another one $\vdash q$:

$$\frac{\vdash p = q \quad \vdash p}{\vdash q}\ \texttt{EQ\_MP}$$

In the usual LCF style, this becomes a function taking two arguments of type `thm` and producing another. In fact, most of these inference rules have no theorems as input, and can thus be considered as axiom schemes. For example, `axiom_addimp` creates theorems of the form $\vdash p \Rightarrow (q \Rightarrow p)$ and `axiom_existseq` creates those of the form $\exists x.\, x = t$ provided x does not appear in the term t:

```
module Proven : Proofsystem =
  struct type thm = formula
         let axiom_addimp p q = Imp(p,Imp(q,p))
         let axiom_distribimp p q r = Imp(Imp(p,Imp(q,r)),Imp(Imp(p,q),Imp(p,r)))
         let axiom_doubleneg p = Imp(Imp(Imp(p,False),False),p)
         let axiom_allimp x p q = Imp(Forall(x,Imp(p,q)),Imp(Forall(x,p),Forall(x,q)))
         let axiom_impall x p =
           if not (free_in (Var x) p) then Imp(p,Forall(x,p))
           else failwith "axiom_impall"
         let axiom_existseq x t =
           if not (occurs_in (Var x) t) then Exists(x,mk_eq (Var x) t)
           else failwith "axiom_existseq"
         let axiom_eqrefl t = mk_eq t t
         let axiom_funcong f lefts rights =
            fold_right2 (fun s t p -> Imp(mk_eq s t,p))
                        lefts rights (mk_eq (Fn(f,lefts)) (Fn(f,rights)))
         let axiom_predcong p lefts rights =
            fold_right2 (fun s t p -> Imp(mk_eq s t,p))
                        lefts rights (Imp(Atom(p,lefts),Atom(p,rights)))
         let axiom_iffimp1 p q = Imp(Iff(p,q),Imp(p,q))
         let axiom_iffimp2 p q = Imp(Iff(p,q),Imp(q,p))
         let axiom_impiff p q = Imp(Imp(p,q),Imp(Imp(q,p),Iff(p,q)))
         let axiom_true = Iff(True,Imp(False,False))
         let axiom_not p = Iff(Not p,Imp(p,False))
         let axiom_or p q = Iff(Or(p,q),Not(And(Not(p),Not(q))))
         let axiom_and p q = Iff(And(p,q),Imp(Imp(p,Imp(q,False)),False))
         let axiom_exists x p = Iff(Exists(x,p),Not(Forall(x,Not p)))
         let modusponens pq p =
           match pq with Imp(p',q) when p = p' -> q
                       | _ -> failwith "modusponens"
         let gen x p = Forall(x,p)
         let concl c = c
  end;;
```

Although simple, these rules are in fact complete for first-order logic with equality. At first they are tedious to use, but using the LCF technique we can build up a set of derived rules. The following derives $p \Rightarrow p$:

```
let imp_refl p = modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
                                          (axiom_addimp p (Imp(p,p))))
                             (axiom_addimp p p);;
```

Before long, we can reach the stage of automatic derived rules that, for example, prove propositional tautologies automatically, perform Knuth-Bendix completion, and prove first order formulas by standard proof search and translation into primitive inferences.

Proof style

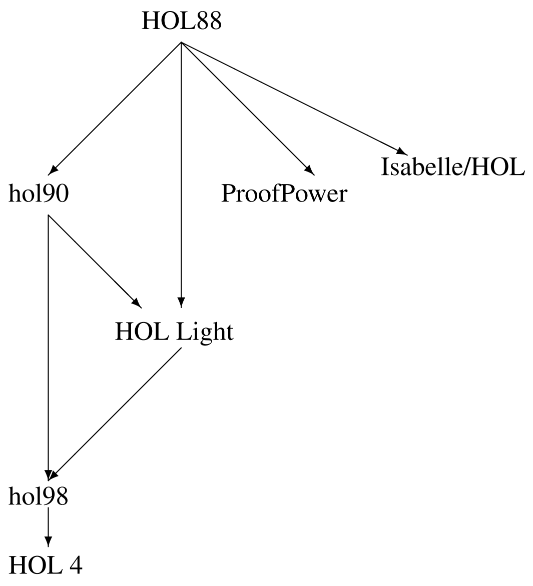
One feature of the LCF style is that proofs (being programs) tend to be highly *procedural*, in contrast to the more declarative proofs supported by Mizar — for more on the contrast see (Harrison 1996c). This can have important disadvantages in terms of readability and maintainability. In particular, it is difficult to understand the formal proof scripts in isolation; they need to be run in the theorem prover to understand what the intermediate states are. Nevertheless as pointed out in (Harrison 1996b) it is possible to implement more declarative styles of proof on top of LCF cores. For more recent experiments with Mizar-like declarative proof styles see (Syme 1997; Wenzel 1999; Zammit 1999; Wiedijk 2001).

Theorem proving in industry

Theorem provers that have been used in real industrial applications include ACL2 (Kaufmann, Manolios, and Moore 2000), HOL Light (Gordon and Melham 1993; Harrison 1996a) and PVS (Owre, Rushby, and Shankar 1992). We noted earlier that formal verification methods can be categorized according to their logical expressiveness and automation. The same kind of balance can be drawn within the general theorem proving section. Although these theorem provers all have undecidable decision problems, it is still possible to provide quite effective partial automation by using a more restricted logic. ACL2 follows this philosophy: it uses a quantifier-free logic analogous to PRA (Primitive Recursive Arithmetic) (Goodstein 1957). HOL and PVS use richer logics with higher-order quantification; PVS's type system is particularly expressive. Nevertheless they attempt to provide some useful automation, and HOL in particular uses the LCF approach to ensure soundness and programmability. This will be emphasized in the application considered below.

7. HOL Light

In the early 80s Mike Gordon (one of the original LCF team, now working in Cambridge), as well as supervising the further development of LCF, was interested in the formal verification of hardware. For this purpose, classical higher order logic seemed a natural vehicle, since it allows a rather direct rendering of notions like signals as functions from time to values. The case was first put by Hanna (Hanna and Daeche 1986) and, after a brief experiment with an ad hoc formalism 'LSM' based on Milner's Calculus of Communicating Systems, Gordon (Gordon 1985) also became a strong advocate. Gordon modified Cambridge LCF to support classical higher order logic, and so HOL (for Higher Order Logic) was born. The first major public release was HOL88 (Gordon and Melham 1993), and it spawned a large family of later versions, of which we will concentrate on our own HOL Light (Harrison 1996a),[5] which is designed to have a simple and clean logical foundation:



Following Church (Church 1940), the HOL logic is based on simply typed λ-calculus, so all terms are either variables, constants, applications or abstractions; there is no distinguished class of formulas, merely terms of boolean type. The main difference from Church's system is that parametric polymorphism is an object-level, rather than a meta-level, notion. For example, a theorem about type (α)`list` can be instantiated and used for specific instances like `(int)list` and `((bool)list)list`. Thus, the types in HOL are essentially like terms of first order logic:

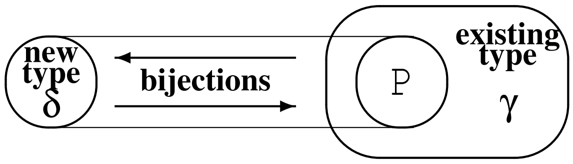
```
type hol_type = Tyvar of string
              | Tyapp of string *  hol_type list;;
```

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space):

```
let the_type_constants = ref ["bool",0; "fun",2];;
```

Later we add an infinite type `ind` (individuals). All other types are introduced by a rule of type definition, to be in bijection with any nonempty subset of an existing type.

[5]See `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`.



HOL terms are those of simply-typed lambda calculus. In the abstract syntax, only variables and constants are decorated with types.

```
type term = Var of string * hol_type
          | Const of string * hol_type
          | Comb of term * term
          | Abs of term * term;;
```

The usual notation for these categories are $v : ty$, $c : ty$, $f\ x$ and $\lambda x.\,t$. The abstract type interface ensures that only well-typed terms can be constructed. This stock of terms may appear spartan, but using defined constants and a layer of parser and prettyprinter support, many standard syntactic conventions are broken down to λ-calculus. For example, the universal quantifier, following Church, is simply a higher order function, but the conventional notation $\forall x.\,P[x]$ is supported, mapping down to $\forall(\lambda x.\,P[x])$. Similarly there is a constant LET, which is semantically the identity and is used only as a tag for the prettyprinter, and following Landin (Landin 1966), the construct 'let $x = t$ in s' is broken down to 'LET $(\lambda x.\,s)\ t$'.

The abstract type interface also ensures that constant terms can only be constructed for defined constants. The only primitive constant for the logic itself is equality = with polymorphic type $\alpha \to \alpha \to$ `bool`.

```
let the_term_constants =
   ref ["=",  mk_fun_ty aty (mk_fun_ty aty bool_ty)];;
```

Later we add the Hilbert $\varepsilon : (\alpha \to$ `bool`$) \to \alpha$ yielding the Axiom of Choice. All other constants are introduced using a rule of constant definition. Given a term t (closed, and with some restrictions on type variables) and an unused constant name c, we can define c and get the new theorem:

$$\vdash c = t$$

Both terms and type definitions give conservative extensions and so in particular preserve logical consistency. Thus, HOL is doubly ascetic:

- All proofs are done by primitive inferences
- All new types and constants are defined not postulated.

As noted, HOL has no separate syntactic notion of formula: we just use terms of Boolean type. HOL's theorems are single-conclusion sequents constructed from such formulas:

```
type thm = Sequent of (term list * term);;
```

In the usual LCF style, these are considered an abstract type and the inference rules become CAML functions operating on type `thm`. For example:

```
let ASSUME tm =
  if type_of tm = bool_ty then Sequent([tm],tm)
  else failwith "ASSUME: not a proposition";;
```

is the rule of assumption. The complete set of primitive rules, in usual logical notation, is as follows (some of these have side-conditions that are not shown here):

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\, s) = (\lambda x.\, t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\, t)x = t}\ \texttt{BETA}$$

$$\frac{}{\{p\} \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]}\ \texttt{INST}$$

$$\frac{\Gamma[\alpha_1,\ldots,\alpha_n] \vdash p[\alpha_1,\ldots,\alpha_n]}{\Gamma[\gamma_1,\ldots,\gamma_n] \vdash p[\gamma_1,\ldots,\gamma_n]} \; \texttt{INST\_TYPE}$$

We can create various simple derived rules in the usual LCF fashion, such as a one-sided congruence rule:

```
let AP_TERM tm th =
  try MK_COMB(REFL tm,th)
  with Failure _ -> failwith "AP_TERM";;
```

and a symmetry rule to reverse equations (`rator` and `rand` return the operator and operand of a function application $f\ x$):

```
let SYM th =
  let tm = concl th in
  let l,r = dest_eq tm in
  let lth = REFL l in
  EQ_MP (MK_COMB(AP_TERM (rator (rator tm)) th,lth)) lth;;
```

Even the logical connectives themselves are defined:

$$\top = (\lambda x.\, x) = (\lambda x.\, x)$$

$$\wedge = \lambda p.\, \lambda q.\, (\lambda f.\, f\ p\ q) = (\lambda f.\, f\ \top\ \top)$$

$$\Rightarrow = \lambda p.\, \lambda q.\, p \wedge q = p$$

$$\forall = \lambda P.\, P = \lambda x.\, \top$$

$$\exists = \lambda P.\, \forall Q.\, (\forall x.\, P(x) \Rightarrow Q) \Rightarrow Q$$

$$\vee = \lambda p.\, \lambda q.\, \forall r.\, (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r$$

$$\bot = \forall P.\, P$$

$$\neg = \lambda t.\, t \Rightarrow \bot$$

$$\exists! = \lambda P.\, \exists P \wedge \forall x.\, \forall y.\, P\ x \wedge P\ y \Rightarrow (x = y)$$

(These are *not* constructive type theory's Curry-Howard definitions.) We can now implement the usual natural deduction rules, such as conjunction introduction:

```
let CONJ =
  let f = `f:bool->bool->bool`
  and p = `p:bool` and q = `q:bool` in
  let pth =
    let pth = ASSUME p and qth = ASSUME q in
    let th1 = MK_COMB(AP_TERM f (EQT_INTRO pth),EQT_INTRO qth) in
    let th2 = ABS f th1 in
    let th3 = BETA_RULE (AP_THM (AP_THM AND_DEF p) q) in
    EQ_MP (SYM th3) th2 in
  fun th1 th2 ->
    let th = INST [concl th1,p; concl th2,q] pth in
    PROVE_HYP th2 (PROVE_HYP th1 th);;
```

Now we can automate monotone inductive definitions, using a Knaster-Tarski derivation. The implementation is quite hard work, but it's then easy to use. It can cope with infinitary definitions and user-defined monotone operators.

```
let TC_RULES,TC_INDUCT,TC_CASES = new_inductive_definition
   `(!x y. R x y ==> TC R x y) /\
    (!x y z. TC R x y /\ TC R y z ==> TC R x z)`;;
```

This just uses the basic logic, but at this point we introduce the usual axioms for classical logic and mathematics:

- Choice in the form of the Hilbert ε and its characterizing axiom $\vdash \forall x.\, P(x) \Rightarrow P(\varepsilon x.\, P(x))$
- Extensionality as an η-conversion axiom $\vdash (\lambda x.\, t\, x) = t$
- Infinity as a new type `ind` and an assertion that it's Dedekind-infinite.

Everything else is now purely definitional. The Law of the Excluded Middle is deduced from Choice (Diaconescu 1975) rather than being postulated separately. We then proceed with the development of standard mathematical infrastructure:

- Basic arithmetic of the natural numbers
- Theory of wellfounded relations
- General recursive data types
- Lists
- Elementary 'set' theory
- Axiom of Choice variants like Zorn's Lemma, wellordering principle etc.
- Construction of real and complex numbers
- Real analysis

Among the higher-level derived rules are:

- Simplifier for (conditional, contextual) rewriting.
- Tactic mechanism for mixed forward and backward proofs.
- Tautology checker.
- Automated theorem provers for pure logic, based on tableaux and model elimination.
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.
- Generic normalizers for rings and fields
- General quantifier elimination over $\mathbb{C}$
- Gröbner basis algorithm over fields

HOL Light is perhaps the purest example of the LCF methodology that is actually useful, in that the logical core is minimal and almost all mathematical concepts are defined or constructed rather than merely postulated. But thanks to the LCF methodology and the speed of modern computers, we can use it to tackle non-trivial mathematics and quite difficult applications.

The first sustained attempt to actually formalize a body of mathematics (concepts and proofs) was *Principia Mathematica* (Whitehead and Russell 1910). This successfully derived a body of fundamental mathematics from a small logical system. However, the task of doing so was extraordinarily painstaking, and indeed Russell (Russell 1968) remarked that his own intellect ‘never quite recovered from the strain of writing it’. But now in the computer age, we can defer much of this work to a computer program like HOL Light.

8. Floating-point verification

In the present section we describe some work applying HOL Light to some problems in industrial floating-point verification, namely correctness of square root algorithms for the Intel® Itanium® architecture.

Square root algorithms based on `fma`

The centrepiece of the Intel® Itanium® floating-point architecture is the `fma` (floating-point multiply-add or fused multiply-accumulate) family of instructions. Given three floating-point numbers x, y and z, these can compute $x \cdot y \pm z$ as an atomic operation, with the final result rounded as usual according to the IEEE Standard 754 for Binary Floating-Point Arithmetic (IEEE 1985), but without intermediate rounding of the product $x \cdot y$. Of course, one can always obtain the usual addition and multiplication operations as the special cases $x \cdot 1 + y$ and $x \cdot y + 0$.

The `fma` has many applications in typical floating-point codes, where it can often improve accuracy and/or performance. In particular (Markstein 1990) correctly rounded quotients and square roots can be computed by fairly short sequences of `fma`s, obviating the need for dedicated instructions. Besides enabling compilers and assembly language programmers to make special optimizations, deferring these operations to software often yields much higher throughput than with typical hardware implementations. Moreover, the floating-point unit becomes simpler and easier to optimize because minimal hardware need be dedicated to these relatively infrequent operations, and scheduling does not have to cope with their exceptionally high latency.

Itanium® architecture compilers for high-level languages will typically translate division or square root operations into appropriate sequences of machine instructions. Which sequence is used depends (i) on the required precision and (ii) whether one wishes to minimize latency or maximize throughput. For concreteness, we will focus on a particular algorithm for calculating square roots in double-extended precision (64-bit precision and 15-bit exponent field):

```
1.  $y_0 = \mathtt{frsqrta}(a)$
2.  $H_0 = \frac{1}{2}y_0$              $S_0 = ay_0$
3.  $d_0 = \frac{1}{2} - S_0H_0$
4.  $H_1 = H_0 + d_0H_0$        $S_1 = S_0 + d_0S_0$
5.  $d_1 = \frac{1}{2} - S_1H_1$
6.  $H_2 = H_1 + d_1H_1$        $S_2 = S_1 + d_1S_1$
7.  $d_2 = \frac{1}{2} - S_2H_2$        $e_2 = a - S_2S_2$
8.  $H_3 = H_2 + d_2H_2$        $S_3 = S_2 + e_2H_2$
9.  $e_3 = a - S_3S_3$
10. $S = S_3 + e_3H_3$
```

All operations but the last are done using the register floating-point format with rounding to nearest and with all exceptions disabled. (This format provides the same 64-bit precision as the target format but has a greater exponent range, allowing us to avoid intermediate overflow or underflow.) The final operation is done in double-extended precision using whatever rounding mode is currently selected by the user.

This algorithm is a non-trivial example in two senses. Since it is designed for the maximum precision supported in hardware (64 bits), greater precision cannot be exploited in intermediate calculations and so a very careful analysis is necessary to ensure correct rounding. Moreover, it is hardly feasible to test such an algorithm exhaustively, even if an accurate and fast reference were available, since there are about 2^{80} possible inputs. (By contrast, one could certainly verify single-precision and conceivably verify double precision by exhaustive or quasi-exhaustive methods.)

Algorithm verification

It's useful to divide the algorithm into three parts, and our discussion of the correctness proof will follow this separation:

1 Form[6] an initial approximation $y_0 = \frac{1}{\sqrt{a}}(1+\varepsilon)$ with $|\varepsilon|$ 6 $2^{-8.8}$.

2–8 Convert this to approximations $H_0 \approx \frac{1}{2\sqrt{a}}$ and $S_0 \approx \sqrt{a}$, then successively refine these to much better approximations H_3 and S_3 using Goldschmidt iteration (Goldschmidt 1964) (a Newton-Raphson variant).

9–10 Use these accurate approximations to produce the square root S correctly rounded according to the current rounding mode, setting IEEE flags or triggering exceptions as appropriate.

Initial approximation

The `frsrta` instruction makes a number of initial checks for special cases that are dealt with separately, and if necessary normalizes the input number. It then uses a simple table lookup to provide the approximation. The algorithm and table used are precisely specified in the Itanium® instruction set architecture. The formal verification is essentially some routine algebraic manipulations for exponent scaling, then a 256-way case split followed by numerical calculation. The following HOL theorem concerns the correctness of the core table lookup:

```
|- normal a ∧ &0 <= Val a
   ⇒ abs(Val(frsqrta a) / inv(sqrt(Val a)) - &1)
            < &303 / &138050
```

Refinement

Each `fma` operation will incur a rounding error, but we can easily find a mathematically convenient (though by no means optimally sharp) bound for the relative error induced by rounding. The key principle is the '$1+e$' property, which states that the rounded

[6]Using `frsqrta`, the only Itanium® instruction specially intended to support square root. In the present discussion we abstract somewhat from the actual machine instruction, and ignore exceptional cases like $a = 0$ where it takes special action.

result involves only a small relative perturbation to the exact result. In HOL the formal statement is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
           (round fmt rc x = x * (&1 + e))
```

The bound on e depends on the precision of the floating-point format and the rounding mode; for round-to-nearest mode, `mu rc` is $1/2$. The theorem has two side conditions, one being a nontriviality hypothesis, and the other an assertion that the value x does not *lose precision*. We will not show the formal definition (Harrison 1999) here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range (or is zero).

Actually applying this theorem, and then bounding the various error terms, would be quite tedious if done by hand. We have programmed some special derived rules in HOL to help us. First, these automatically bound absolute magnitudes of quantities, essentially by using the triangle rule $|x+y| 6 |x|+|y|$. This usually allows us to show that no overflow occurs. However, to apply the $1+e$ theorem, we also need to exclude underflow, and so must establish *minimum* (nonzero) absolute magnitudes. This is also largely done automatically by HOL, repeatedly using theorems for the minimum nonzero magnitude that can result from an individual operation. For example, if $2^e 6 |a|$, then either $a+b\cdot c$ is exactly zero or $2^{e-2p} 6 |a+b\cdot c|$ where p is the precision of the floating-point format containing a, b and c.

It's now quite easy with a combination of automatic error bounding and some manual algebraic rearrangement to obtain quite good relative error bounds for the main computed quantities. In fact, in the early iterations, the rounding errors incurred are insignificant in comparison with the approximation errors in the H_i and S_i. Thus, the relative errors in these quantities are roughly in step. If we write

$$H_i \approx \frac{1}{2\sqrt{a}}(1+\varepsilon_i) \qquad S_i \approx \sqrt{a}(1+\varepsilon_i)$$

then

$$d_i \approx \frac{1}{2} - S_i H_i = \frac{1}{2} - \frac{1}{2}(1+\varepsilon_i)^2 = -(\varepsilon_i + \varepsilon_i^2/2)$$

Consequently, correcting the current approximations in the manner indicated will approximately square the relative error, e.g.

$$S_{i+1} \approx S_i + d_i S_i = S_i(1+d_i) \approx \sqrt{a}(1+\varepsilon_i)(1-\varepsilon_i-\varepsilon_i^2/2) = \sqrt{a}(1-\frac{3}{2}\varepsilon_i^2)$$

Towards the end, the rounding errors in S_i and H_i become more significantly decoupled and for the penultimate iteration we use a slightly different refinement for S_3.

$$e_2 \approx a - S_2S_2 = a - (\sqrt{a}(1+\varepsilon_2)^2) \approx -2a\varepsilon_2$$

and so:

$$S_2 + e_2H_2 \approx \sqrt{a}(1+\varepsilon_2) - (2a\varepsilon_2)(\frac{1}{2\sqrt{a}}(1+\varepsilon_2')) \approx \sqrt{a}(1-\varepsilon_2\varepsilon_2')$$

Thus, $S_2 + e_2H_2$ will be quite an accurate square root approximation. In fact the HOL proof yields $S_2 + e_2H_2 = \sqrt{a}(1+\varepsilon)$ with $|\varepsilon|$ 6 $5579/2^{79} \approx 2^{-66.5}$.

The above sketch elides what in the HOL proofs is a detailed bound on the rounding error. However this only really becomes significant when S_3 is rounded; this may in itself contribute a relative error of order 2^{-64}, significantly more than the error before rounding. Nevertheless it is important to note that if $\sqrt{a}$ happens to be an exact floating-point number (e.g. $\sqrt{1.5625} = 1.25$), S_3 will be that number. This is a consequence of the fact that the error in $S_2 + e_2H_2$ is less than half the distance between surrounding floating-point numbers.

Correct rounding

The final two steps of the algorithm simply repeat the previous iteration for S_3 and the basic error analysis is the same. The difficulty is in passing from a relative error before rounding to correct rounding afterwards. Again we consider the final rounding separately, so S is the result of rounding the exact value $S^* = S_3 + e_3H_3$. The error analysis indicates that $S^* = \sqrt{a}(1+\varepsilon)$ for some $|\varepsilon|$ 6 $25219/2^{140} \approx 2^{-125.37}$. The final result S will, by the basic property of the `fma` operation, be the result of rounding S^* in whatever the chosen rounding mode may be. The *desired* result would be the result of rounding exactly $\sqrt{a}$ in the same way. How can we be sure these are the same?

First we can dispose of some special cases. We noted earlier that if $\sqrt{a}$ is already exactly a floating-point number, then S_3 will already be that number. In this case we will have $e_3 = 0$ and so $S^* = S_3$. Whatever the rounding mode, rounding a number already in the format concerned will give that number itself:

```
  |- a IN iformat fmt ⇒ (round fmt rc a = a)
```

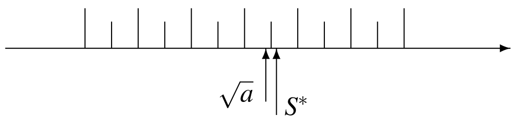
so the result will be correct. Moreover, the earlier observation extends to show that if $\sqrt{a}$ is fairly close (in a precise sense) to a floating-point number, then S_3 will be that number. It is then quite straightforward to see that the overall algorithm will be accurate without great subtlety: we just need the fact that e_3 has the right sign and roughly the correct magnitude, so S^* will never misround in directed rounding modes. Thus, we can also deal immediately with what would otherwise be difficult cases for the directed rounding modes, and concentrate our efforts on rounding to nearest.

On general grounds we note that $\sqrt{a}$ *cannot* be exactly the mid-point between two floating-point numbers. This is not hard to see, since the square root of a number in a given format cannot denormalize in that format, and a non-denormal midpoint has $p+1$ significant digits, so its square must have more than p.[7]

[7] An analogous result holds for quotients but here the denormal case must be dealt with specially. For example $2^{E_{min}} \times 0.111\cdots111/2$ is exactly a midpoint.

```
|- &0 <= a ∧ a IN iformat fmt ∧ b IN midpoints fmt
   ⇒ ¬(sqrt a = b)
```

This is a useful observation. We'll never be in the tricky case where there are two equally close floating-point numbers (resolved by the 'round to even' rule.) So in round-to-nearest, S^* and $\sqrt{a}$ could only round in different ways if there were a midpoint between them, for only then could the closest floating-point numbers to them differ. For example in the following diagram where large lines indicate floating-point numbers and smaller ones represent midpoints, $\sqrt{a}$ would round 'down' while S^* would round 'up':[8]



Although analyzing this condition combinatorially would be complicated, there is a much simpler sufficient condition. One can easily see that it would suffice to show that for any midpoint m:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* couldn't lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m IN midpoints fmt ⇒ abs(x - y) < abs(x - m))
   ⇒ (round fmt Nearest x = round fmt Nearest y)
```

One can arrive at an 'exclusion zone' theorem giving the minimum possible $|\sqrt{a} - m|$. However, this can be quite small, about $2^{-(2p+3)}$ relative to $\sqrt{a}$, where p is the precision. For example, in our context with $p = 64$, consider the square root of the next floating-point number below 1, whose mantissa consists entirely of 1s. Its square root is about 2^{-131} from a midpoint:

$$\sqrt{1 - 2^{-64}} \approx (1 - 2^{65}) - 2^{-131}$$

Therefore, our relative error in S^* of about $2^{-125.37}$ is far from adequate to justify perfect rounding based on the simple 'exclusion zone' theorem, for which we need something of order 2^{-131}. However, our relative error bounds are far from sharp, and it seems quite plausible that the algorithm does nevertheless work correctly. What can we do?

One solution is to utilize more refined theorems (Markstein 2000), but this is complicated and may still fail to justify several algorithms that are intuitively believed to work correctly. An ingenious alternative developed by Cornea (Cornea-Hasegan 1998) is to observe that there are relatively few cases like $0.111\cdots1111$ whose square roots come close enough to render the exclusion zone theorem inapplicable, and these can be isolated by fairly straightforward number-theoretic methods. We can therefore:

[8]Similarly, in the other rounding modes, misrounding could only occur if $\sqrt{a}$ and S^* are separated by a floating-point number. However as we have noted one can deal with those cases more directly.

- Isolate the special cases $a_1, \ldots, a_n$ that have square roots within the critical distance of a midpoint.
- Conclude from the simple exclusion zone theorem that the algorithm will give correct results except possibly for $a_1, \ldots, a_n$.
- Explicitly show that the algorithm is correct for the $a_1, \ldots, a_n$, (effectively by running it on those inputs).

This two-part approach is perhaps a little unusual, but not unknown even in pure mathematics.[9] For example, consider “Bertrand’s Conjecture” (first proved by Chebyshev), stating that for any positive integer n there is a prime p with $n \leqslant p \leqslant 2n$. The most popular proof (Erdös 1930), involves assuming $n > 4000$ for the main proof and separately checking the assertion for $n \leqslant 4000$.[10]

By some straightforward mathematics described in (Cornea-Hasegan 1998) and formalized in HOL without difficulty, one can show that the difficult cases for square roots have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for some integer $|d| \leqslant D$, where D is roughly the factor by which the guaranteed relative error is excessive:

$$2^{p+2}m = k^2 + d \qquad 2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen $|d| \leqslant D$. For example, we might be interested in whether $2^{p+1}m = k^2 - 7$ has a solution. If so, the possible value(s) of m are added to the set of difficult cases. It’s quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$\vdash (2^{p+1}m = k^2 + d) \Rightarrow (m = n_1) \vee \ldots \vee (m = n_i)$$

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called ‘Hensel lifting’). For example, if

$$2^{25}m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and deduce:

$$2^{24}m = 2k'^2 + 2k' - 3$$

By more even/odd reasoning, this has no solutions. In general, we recurse down to an equation that is trivially unsatisfiable, as here, or immediately solvable. One equation can split into two, but never more. For example, we have a formally proved HOL the-

[9] A more extreme case is the 4-color theorem, whose proof relies on extensive (computer-assisted) checking of special cases (Appel and Haken 1976).

[10] An ‘optimized’ way of checking, referred to in (Aigner and Ziegler 2001) as “Landau’s trick”, is to verify that 3, 5, 7, 13, 23, 43, 83, 163, 317, 631, 1259, 2503 and 4001 are all prime and each is less than twice its predecessor.

orem asserting that for any double-extended number a,[11] rounding $\sqrt{a}$ and $\sqrt{a}(1+\varepsilon)$ to double-extended precision using any of the four IEEE rounding modes will give the same results provided $|\varepsilon| < 31/2^{131}$, with the possible exceptions of $2^{2e}m$ for:

$$\begin{aligned} m \in \{ & 10074057467468575321, 10376293541461622781, \\ & 10376293541461622787, 11307741603771905196, \\ & 13812780109330227882, 14928119304823191698, \\ & 16640932189858196938, 18446744073709551611, \\ & 18446744073709551612, 18446744073709551613, \\ & 18446744073709551614, 18446744073709551615\} \end{aligned}$$

and $2^{2e+1}m$ for

$$\begin{aligned} m \in \{ & 9223372036854775809, 9223372036854775811, \\ & 11168682418930654643\} \end{aligned}$$

Note that while some of these numbers are obvious special cases like $2^{64}-1$, the "pattern" in others is only apparent from the kind of mathematical analysis we have undertaken here. They aren't likely to be exercised by random testing, or testing of plausible special cases.[12]

Checking formally that the algorithm works on the special cases can also be automated, by applying theorems on the uniqueness of rounding to the concrete numbers computed. (For a formal proof, it is not sufficient to separately test the implemented algorithm, since such a result has no formal status.) In order to avoid trying all possible even or odd exponents for the various significands, we exploit some results on invariance of the rounding and arithmetic involved in the algorithm under systematic scaling by 2^{2k}, doing a simple form of symbolic simulation by formal proof.

Flag settings

Correctness according to the IEEE Standard 754 not only requires the correctly rounded result, but the correct setting of flags or triggering of exceptions for conditions like overflow, underflow and inexactness. Actually, almost all these properties follow directly from the arguments leading to perfect rounding. For example, the mere fact that two real numbers round equivalently *in all rounding modes* implies that one is exact iff the other is:

```
|- ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⇒ ∀rc. (round fmt rc x = x) = (round fmt rc y = y)
```

The correctness of other flag settings follows in the same sort of way, with underflow only slightly more complicated (Harrison 1999).

[11]Note that there is more subtlety required when using such a result in a mixed-precision environment. For example, to obtain a single-precision result for a double-precision input, an algorithm that suffices for single-precision inputs may not be adequate even though the final precision is the same.

[12]On the other hand, we can well consider the mathematical analysis as a *source* of good test cases.

Conclusions and related work

What do we gain from developing these proofs formally in a theorem prover, compared with a detailed hand proof? We see two main benefits: reliability and re-usability.

Proofs of this nature, large parts of which involve intricate but routine error bounding and the exhaustive solution of Diophantine equations, are very tedious and error-prone to do by hand. In practice, one would do better to use *some* kind of machine assistance, such as *ad hoc* programs to solve the Diophantine equations and check the special cases so derived. Although this can be helpful, it can also create new dangers of incorrectly implemented helper programs and transcription errors when passing results between 'hand' and 'machine' portions of the proof. By contrast, we perform all steps of the proof in a painstakingly foundational system, and can be quite confident that no errors have been introduced. The proof proceeds according to strict logical deduction, all the way from the underlying pure mathematics up to the symbolic "execution" of the algorithm in special cases.

Although we have only discussed one particular example, many algorithms with a similar format have been developed for use in systems based on the Itanium® architecture. One of the benefits of implementing division and square root in software is that different algorithms can be substituted depending on the detailed accuracy and performance requirements of the application. Not only are different (faster) algorithms provided for IEEE single and double precision operations, but algorithms often have two versions, one optimized for minimum latency and one for maximal throughput. These algorithms are all quite similar in structure and large parts of the correctness proofs use the same ideas. By performing these proofs in a programmable theorem prover like HOL, we are able to achieve high re-use of results, just tweaking a few details each time. Often, we can produce a complete formal proof of a new algorithm in just a day. For a even more rigidly stereotyped class of algorithms, one could quite practically implement a totally automatic verification rule in HOL.

Underlying these advantages are three essential theorem prover features: soundness, programmability and executability. HOL scores highly on these points. It is implemented in a highly foundational style and does not rely on the correctness of very complex code. It is freely programmable, since it is embedded in a full programming language. In particular, one can program it to perform various kinds of computation and symbolic execution by proof. The main disadvantage is that proofs can sometimes take a long time to run, precisely because they *always* decompose to low-level primitives. This applies with particular force to some kinds of symbolic execution, where instead of simply accepting an equivalence like $2^{94}+3 = 13 \cdot 19 \cdot 681943 \cdot 7941336391 \cdot 14807473717$ based, say, on the results of a multiprecision arithmetic package, a detailed formal proof is constructed under the surface. To some extent, this sacrifice of efficiency is a conscious choice when we decide to adopt a highly foundational system, but it might be worth weakening this ideology at least to include concrete arithmetic as an efficient primitive operation.

This is by no means the only work in this area. On the contrary, floating-point verification is regarded as one of the success stories of formal verification. For related work on square root algorithms in commercial systems see (Russinoff 1998; O'Leary, Zhao, Gerth, and Seger 1999; Sawada and Gamboa 2002). For similar verifications of division algorithms and transcendental functions by the present author, see (Harrison 2000b; Harrison 2000a), while (Muller 2006) and (Markstein 2000) give a detailed discussion of some relevant floating-point algorithms. When verifying transcendental functions, one needs a fair amount of pure mathematics formalized just to get started. So, in a common intellectual pattern, it is entirely possible that Mizar-like formalizations of mathematics undertaken for purely intellectual reasons might in the end turn out to be useful in practical applications.

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*, Volume 663 of *Lecture Notes in Computer Science*, Montreal, Canada, pp. 69–81. Springer Verlag.

Abdulla, P. A., Bjesse, P., and Eén, N. (2000) Symbolic reachability analysis based on SAT-solvers. In Graf, S. and Schwartzbach, M. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, Volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag.

Aigner, M. and Ziegler, G. M. (2001) *Proofs from The Book* (2nd ed.). Springer-Verlag.

Akers, S. B. (1978) Binary decision diagrams. *ACM Transactions on Computers*, **C-27**, 509–516.

Alur, R. and Peled, D. A. (eds.) (2004) *Computer Aided Verification, 16th International Conference, CAV 2004*, Volume 3114 of *Lecture Notes in Computer Science*, Boston, MA. Springer-Verlag.

Andrews, P. B. (1976) Theorem proving by matings. *IEEE transactions on Computers*, **25**, 801–807.

Appel, K. and Haken, W. (1976) Every planar map is four colorable. *Bulletin of the American Mathematical Society*, **82**, 711–712.

Armando, A., Castellini, C., and Giunchiglia, E. (1999) SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European conference on Planning*, Lecture Notes in Computer Science, pp. 97–108. Springer-Verlag.

Ball, T., Cook, B., Lahriri, S. K., and Rajamani, S. K. (2004) Zapato: Automatic theorem proving for predicate abstraction refinement. See Alur and Peled (2004), pp. 457–461.

Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., and Théry, L. (eds.) (1999) *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Volume 1690 of *Lecture Notes in Computer Science*, Nice, France. Springer-Verlag.

Bibel, W. and Schreiber, J. (1975) Proof search in a Gentzen-like system of first order logic. In Gelenbe, E. and Potier, D. (eds.), *Proceedings of the International Computing Symposium*, pp. 205–212. North-Holland.

Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999) Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Volume 1579 of *Lecture Notes in Computer Science*, pp. 193–207. Springer-Verlag.

Bjesse, P. (1999) Symbolic model checking with sets of states represented as formulas. Technical Report SC-1999-100, Department of Computer Science, Chalmers University of Technology.

Bochnak, J., Coste, M., and Roy, M.-F. (1998) *Real Algebraic Geometry*, Volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag.

Bradley, A. R. and Manna, Z. (2007) *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag.

Bryant, R. E. (1985) Symbolic verification of MOS circuits. In Fuchs, H. (ed.), *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pp. 419–438. Computer Science Press.

Bryant, R. E. (1986) Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, **C-35**, 677–691.

Bryant, R. E. (1991a) A method for hardware verification based on logic simulation. *Journal of the ACM*, **38**, 299–328.

Bryant, R. E. (1991b) On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, **C-40**, 205–213.

Buchberger, B. (1965) *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. Ph. D. thesis, Mathematisches Institut der Universität Innsbruck. English translation in Journal of Symbolic Computation vol. 41 (2006), pp. 475–511.

Bumcrot, R. (1965) On lattice complements. *Proceedings of the Glasgow Mathematical Association*, **7**, 22–23.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992) Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, **98**, 142–170.

Carter, W. C., Joyner, W. H., and Brand, D. (1979) Symbolic simulation for correct machine design. In *Proceedings of the 16th ACM/IEEE Design Automation Conference*, pp. 280–286. IEEE Computer Society Press.

Caviness, B. F. and Johnson, J. R. (eds.) (1998) *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and monographs in symbolic computation. Springer-Verlag.

Chou, C.-T. and Peled, D. (1999) Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, **23**, 265–298.

Church, A. (1936) An unsolvable problem of elementary number-theory. *American Journal of Mathematics*, **58**, 345–363.

Church, A. (1940) A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**, 56–68.

Clarke, E. M. and Emerson, E. A. (1981) Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen, D. (ed.), *Logics of Programs*, Volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, pp. 52–71. Springer-Verlag.

Clarke, E. M., Grumberg, O., and Peled, D. (1999) *Model Checking*. MIT Press.

Collins, G. E. (1976) Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In Brakhage, H. (ed.), *Second GI Conference on Automata Theory and Formal Languages*, Volume 33 of *Lecture Notes in Computer Science*, Kaiserslautern, pp. 134–183. Springer-Verlag.

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

Cook, S. A. (1971) The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pp. 151–158. ACM.

Cooper, D. C. (1972) Theorem proving in arithmetic without multiplication. In Melzer, B. and Michie, D. (eds.), *Machine Intelligence 7*, pp. 91–99. Elsevier.

Cornea-Hasegan, M. (1998) Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, **1998-Q2**, 1–11. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

Coudert, O., Berthet, C., and Madre, J.-C. (1989) Verification of synchronous sequential machines based on symbolic execution. In Sifakis, J. (ed.), *Automatic Verification Methods for Finite State Systems*, Volume 407 of *Lecture Notes in Computer Science*, pp. 365–373. Springer-Verlag.

Cox, D., Little, J., and O'Shea, D. (1992) *Ideals, Varieties, and Algorithms*. Springer-Verlag.

Davis, M. (1983) The prehistory and early history of automated deduction. See Siekmann and Wrightson (1983), pp. 1–28.

Davis, M., Logemann, G., and Loveland, D. (1962) A machine program for theorem proving. *Communications of the ACM*, **5**, 394–397.

Davis, M. and Putnam, H. (1960) A computing procedure for quantification theory. *Journal of the ACM*, **7**, 201–215.

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

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.

DeMillo, R., Lipton, R., and Perlis, A. (1979) Social processes and proofs of theorems and programs. *Communications of the ACM*, **22**, 271–280.

Diaconescu, R. (1975) Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, **51**, 176–178.

Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice-Hall.

Downey, P. J., Sethi, R., and Tarjan, R. (1980) Variations on the common subexpression problem. *Journal of the ACM*, **27**, 758–771.

Eén, N. and Sörensson, N. (2003) An extensible SAT-solver. In Giunchiglia, E. and Tacchella, A. (eds.), *Theory and Applications of Satisfiability Testing: 6th International Conference SAT 2003*, Volume 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer-Verlag.

Emerson, E. A. and Halpern, J. Y. (1986) "sometimes" and "not never" revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, **33**, 151–178.

Enderton, H. B. (1972) *A Mathematical Introduction to Logic*. Academic Press.

Erdös, P. (1930) Beweis eines Satzes von Tschebyshev. *Acta Scientiarum Mathematicarum (Szeged)*, **5**, 194–198.

Fetzer, J. H. (1988) Program verification: The very idea. *Communications of the ACM*, **31**, 1048–1063.

Gårding, L. (1997) *Some Points of Analysis and Their History*, Volume 11 of *University Lecture Series*. American Mathematical Society / Higher Education Press.

Gilmore, P. C. (1960) A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, **4**, 28–35.

Gödel, K. (1930) Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, **37**, 349–360. English translation 'The completeness of the axioms of the functional calculus of logic' in van Heijenoort (1967), pp. 582–591.

Goldberg, E. and Novikov, Y. (2002) BerkMin: a fast and robust Sat-solver. In Kloos, C. D. and Franca, J. D. (eds.), *Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, Paris, France, pp. 142–149. IEEE Computer Society Press.

Goldschmidt, R. E. (1964) Applications of division by convergence. Master's thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass.

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

Gordon, M. (1985) Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

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

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 Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Groote, J. F. (2000) The propositional formula checker Heerhugo. *Journal of Automated Reasoning*, **24**, 101–125.

Hanna, F. K. and Daeche, N. (1986) Specification and verification using higher-order logic: A case study. In Milne, G. and Subrahmanyam, P. A. (eds.), *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pp. 179–213.

Harrison, J. (1996a) HOL Light: A tutorial introduction. In Srivas, M. and Camilleri, A. (eds.), *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, Volume 1166 of *Lecture Notes in Computer Science*, pp. 265–269. Springer-Verlag.

Harrison, J. (1996b) A Mizar mode for HOL. In Wright, J. v., Grundy, J., and Harrison, J. (eds.), *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, pp. 203–220. Springer-Verlag.

Harrison, J. (1996c) Proof style. In Giménez, E. and Paulin-Mohring, C. (eds.), *Types for Proofs and Programs: International Workshop TYPES'96*, Volume 1512 of *Lecture Notes in Computer Science*, Aussois, France, pp. 154–172. Springer-Verlag.

Harrison, J. (1999) A machine-checked theory of floating point arithmetic. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 113–130.

Harrison, J. (2000a) Formal verification of floating point trigonometric functions. In Hunt, W. A. and Johnson, S. D. (eds.), *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, Volume 1954 of *Lecture Notes in Computer Science*, pp. 217–233. Springer-Verlag.

Harrison, J. (2000b) Formal verification of IA-64 division algorithms. In Aagaard, M. and Harrison, J. (eds.), *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, Volume 1869 of *Lecture Notes in Computer Science*, pp. 234–251. Springer-Verlag.

Harrison, J. (2009) *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

Harrison, J. and Théry, L. (1998) A sceptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, **21**, 279–294.

Harvey, W. and Stuckey, P. (1997) A unit two variable per inequality integer constraint solver for constraint logic programming. *Australian Computer Science Communications*, **19**, 102–111.

Hörmander, L. (1983) *The Analysis of Linear Partial Differential Operators II*, Volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag.

Huth, M. and Ryan, M. (1999) *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press.

IEEE (1985) Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

Iwashita, H., Nakata, T., and Hirose, F. (1996) CTL model checking based on forward state traversal. In *Proceedings of tte IEEE/ACM conference on Computer Aided Design (ICCAD '96)*, pp. 82–87. Association for Computing Machinery.

Joyce, J. J. and Seger, C. (1993) The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In Joyce, J. J. and Seger, C. (eds.), *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, Volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, pp. 185–198. Springer-Verlag.

Kaufmann, M., Manolios, P., and Moore, J. S. (2000) *Computer-Aided Reasoning: An Approach*. Kluwer.

Knaster, B. (1927) Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, **6**, 133–134. Volume published in 1928.

Kowalski, R. (1975) A proof procedure using connection graphs. *Journal of the ACM*, **22**, 572–595.

Kozen, D. (1983) Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**, 333–354.

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' published by Dunod, Paris in 1964.

Kroening, D. and Strichman, O. (2008) *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag.

Kropf, T. (1999) *Introduction to Formal Hardware Verification*. Springer-Verlag.

Lahiri, S. K. and Bryant, R. E. (2004) Indexed predicate discovery for unbounded system verification. See Alur and Peled (2004), pp. 135–147.

Landin, P. J. (1966) The next 700 programming languages. *Communications of the ACM*, **9**, 157–166.

Lee, C. Y. (1959) Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, **38**, 985–999.

Lifschitz, V. (1986) *Mechanical Theorem Proving in the USSR: the Leningrad School*. Monograph Series on Soviet Union. Delphic Associates, 7700 Leesburg Pike, #250, Falls Church, VA 22043. See also 'What is the inverse method?' in the Journal of Automated Reasoning, vol. 5, pp. 1–23, 1989.

Łojasiewicz, S. (1964) Triangulations of semi-analytic sets. *Annali della Scuola Normale Superiore di Pisa, ser. 3*, **18**, 449–474.

Loveland, D. W. (1968) Mechanical theorem-proving by model elimination. *Journal of the ACM*, **15**, 236–251.

Loveland, D. W. (1978) *Automated theorem proving: a logical basis*. North-Holland.

MacKenzie, D. (2001) *Mechanizing Proof: Computing, Risk and Trust*. MIT Press.

Markstein, P. (2000) *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall.

Markstein, P. W. (1990) Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, **34**, 111–119.

Maslov, S. J. (1964) An inverse method of establishing deducibility in classical predicate calculus. *Doklady Akademii Nauk*, **159**, 17–20.

McMillan, K. L. (2003) Interpolation and SAT-based model checking. In Hunt, W. A. and Somenzi, F. (eds.), *Computer Aided Verification, 15th International Conference, CAV 2003*, Volume 2725 of *Lecture Notes in Computer Science*, Boulder, CO, pp. 1–13. Springer-Verlag.

Melham, T. and Darbari, A. (2002) Symbolic trajectory evaluation in a nutshell. Unpublished, available from the authors.

Mendelson, E. (1987) *Introduction to Mathematical Logic* (Third ed.). Mathematics series. Wadsworth and Brooks Cole.

Monk, J. D. (1976) *Mathematical logic*, Volume 37 of *Graduate Texts in Mathematics*. Springer-Verlag.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001) Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 530–535. ACM Press.

Muller, J.-M. (2006) *Elementary functions: Algorithms and Implementation* (2nd ed.). Birkhäuser.

Nelson, G. and Oppen, D. (1979) Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, **1**, 245–257.

Nelson, G. and Oppen, D. (1980) Fast decision procedures based on congruence closure. *Journal of the ACM*, **27**, 356–364.

O'Leary, J., Zhao, X., Gerth, R., and Seger, C.-J. H. (1999) Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, **1999-Q1**, 1–14. Available on the Web as `http://download.intel.com/technology/itj/q11999/pdf/floating_point.pdf`.

Owre, S., Rushby, J. M., and Shankar, N. (1992) PVS: A prototype verification system. In Kapur, D. (ed.), *11th International Conference on Automated Deduction*, Volume 607 of *Lecture Notes in Computer Science*, Saratoga, NY, pp. 748–752. Springer-Verlag.

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

Peled, D. A. (2001) *Software Reliability Methods*. Springer-Verlag.

Pixley, C. (1990) A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the DIMACS workshop on Computer Aided Verification*, pp. 293–320. DIMACS (technical report 90-31).

Pnueli, A. (1977) The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–67.

Prawitz, D., Prawitz, H., and Voghera, N. (1960) A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, **7**, 102–128.

Presburger, M. (1930) Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pp. 92–101, 395. Warsaw. Annotated English version by Stansifer (1984).

Queille, J. P. and Sifakis, J. (1982) Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, Volume 137 of *Lecture Notes in Computer Science*, pp. 195–220. Springer-Verlag.

Rabin, M. O. (1991) Decidable theories. In Barwise, J. and Keisler, H. (eds.), *Handbook of mathematical logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 595–629. North-Holland.

Rajan, S., Shankar, N., and Srivas, M. K. (1995) An integration of model-checking with automated proof-checking. In Wolper, P. (ed.), *Computer-Aided Verification: CAV '95*, Volume 939 of *Lecture Notes in Computer Science*, Liege, Belgium, pp. 84–97. Springer-Verlag.

Reuß, H. and Shankar, N. (2001) Deconstructing Shostak. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 19–28. IEEE Computer Society Press.

Robinson, A. (1957) Proving a theorem (as done by man, logician, or machine). In *Summaries of Talks Presented at the Summer Institute for Symbolic Logic*. Second edition published by the Institute for Defense Analysis, 1960. Reprinted in Siekmann and Wrightson (1983), pp. 74–76.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**, 23–41.

Russell, B. (1968) *The autobiography of Bertrand Russell*. Allen & Unwin.

Russinoff, D. (1998) A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, **1**, 148–200. Available on the Web at `http://www.russinoff.com/papers/k7-div-sqrt.html`.

Sawada, J. and Gamboa, R. (2002) Mechanical verification of a square root algorithms using Taylor's theorem. In Aagaard, M. and O'Leary, J. (eds.), *Formal Methods in Computer-Aided Design: Fourth International Conference FMCAD 2002*, Volume 2517 of *Lecture Notes in Computer Science*, pp. 274–291. Springer-Verlag.

Scarpellini, B. (1969) On the metamathematics of rings and integral domains. *Transactions of the American Mathematical Society*, **138**, 71–96.

Seger, C. and Joyce, J. J. (1991) 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.

Seger, C.-J. H. and Bryant, R. E. (1995) Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, **6**, 147–189.

Sheeran, M. and Stålmarck, G. (2000) A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, **16**, 23–58.

Shostak, R. (1978) An algorithm for reasoning about equality. *Communications of the ACM*, **21**, 356–364.

Shostak, R. (1984) Deciding combinations of theories. *Journal of the ACM*, **31**, 1–12.

Siekmann, J. and Wrightson, G. (eds.) (1983) *Automation of Reasoning — Classical Papers on Computational Logic, Vol. I (1957-1966)*. Springer-Verlag.

Simmons, H. (1970) The solution of a decision problem for several classes of rings. *Pacific Journal of Mathematics*, **34**, 547–557.

Skolem, T. (1922) Einige Bemerkungen zur axiomatischen Begründung der Mengenlehre. In *Matematikerkongress i Helsingfors den 4–7 Juli 1922, Den femte skandinaviska matematikerkongressen, Redogörelse*. Akademiska Bokhandeln, Helsinki. English translation "Some remarks on axiomatized set theory" in van Heijenoort (1967), pp. 290–301.

Smoryński, C. (1980) *Logical Number Theory I: An Introduction*. Springer-Verlag.

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

Stålmarck, G. and Säflund, M. (1990) Modeling and verifying systems and software in propositional logic. In Daniels, B. K. (ed.), *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, Gatwick, UK, pp. 31–36. Pergamon Press.

Stansifer, R. (1984) Presburger's article on integer arithmetic: Remarks and translation. Technical Report CORNELLCS:TR84-639, Cornell University Computer Science Department.

Syme, D. (1997) DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Tarski, A. (1951) *A Decision Method for Elementary Algebra and Geometry*. University of California Press. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J. C. C. McKinsey. Reprinted in Caviness and Johnson (1998), pp. 24–84.

Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**, 285–309.

Tarski, A. (1965) A simplified formalization of predicate logic with identity. *Arkhiv für mathematische Logik und Grundlagenforschung*, **7**, 61–79.

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

Trybulec, A. and Blair, H. A. (1985) Computer aided reasoning. In Parikh, R. (ed.), *Logics of Programs*, Volume 193 of *Lecture Notes in Computer Science*, Brooklyn, pp. 406–412. Springer-Verlag.

Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, **42**, 230–265.

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

Wang, H. (1960) Toward mechanical mathematics. *IBM Journal of Research and Development*, **4**, 2–22.

Wenzel, M. (1999) Isar - a generic interpretive approach to readable formal proof documents. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 167–183.

Whitehead, A. N. and Russell, B. (1910) *Principia Mathematica (3 vols)*. Cambridge University Press.

Wiedijk, F. (2001) Mizar light for HOL Light. In Boulton, R. J. and Jackson, P. B. (eds.), *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, Volume 2152 of *Lecture Notes in Computer Science*, pp. 378–394. Springer-Verlag.

Yang, J. (2000) A theory for generalized symbolic trajectory evaluation. In *Proceedings of the 2000 Symposium on Symbolic Trajectory Evaluation*, Chicago. Available via `http://www.intel.com/research/scl/stesympsite.htm`.

Zammit, V. (1999) On the implementation of an extensible declarative proof language. See Bertot, Dowek, Hirschowitz, Paulin, and Théry (1999), pp. 185–202.