Formal Verification using HOL Light

John Harrison
Intel Corporation

Kestrel Institute

Palo Alto CA

March 22, 2005

Floating-point bugs

Even when not a matter of life and death, the financial consequences of a bug can be very serious:

- 1994 FDIV bug in the Intel®Pentium® processor: US $500 million.
- Today, new products are ramped much faster and a similar bug might be even more expensive.

So Intel is especially interested in all techniques to reduce errors.

Complexity of designs

At the same time, market pressures are leading to more and more complex designs where bugs are more likely.

- A 4-fold increase in bugs in Intel processor designs per generation.
- Approximately 8000 bugs introduced during design of the Pentium 4.

Fortunately, pre-silicon detection rates are now very close to 100%.

Just enough to tread water . . .

Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

- Slow — especially pre-silicon
- Too many possibilities to test them all

For example:

- 2^{160} possible pairs of floating point numbers (possible inputs to an adder).
- Vastly higher number of possible states of a complex microarchitecture.

Consequently, considerable interest in formal verification methods.

Formal verification in industry

Formal verification is increasingly becoming standard practice in the hardware industry.

- Hardware is designed in a more modular way than most software.
- There is more scope for complete automation
- The potential consequences of a hardware error are greater

But currently increasing interest in model checking and theorem proving in the software industry.

Formal verification methods

Many different methods are used in formal verification, mostly trading efficiency and automation against generality.

- Propositional tautology checking
- Symbolic simulation
- Symbolic trajectory evaluation
- Temporal logic model checking
- Decidable subsets of first order logic
- First order automated theorem proving
- Interactive theorem proving

Intel’s formal verification work

Intel uses formal verification quite extensively, e.g.

- Verification of Intel® Pentium® 4 floating-point unit with a mixture of STE and theorem proving
- Verification of bus protocols using pure temporal logic model checking
- Verification of microcode and software for many Intel® Itanium® floating-point operations, using pure theorem proving

FV found many high-quality bugs in P4 and verified “20%” of design

FV is now standard practice in the floating-point domain

Our work

Here we will focus on our work using pure theorem proving.

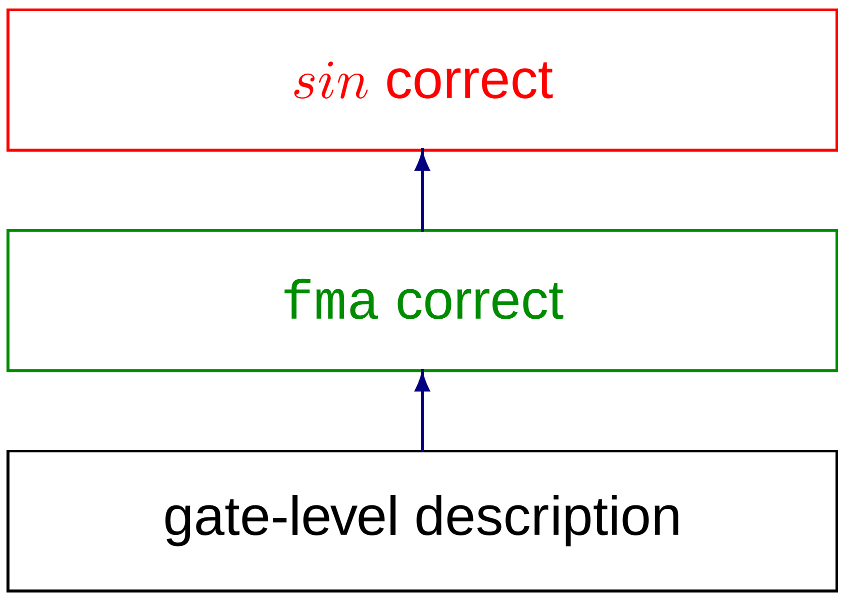
We have formally verified correctness of various floating-point algorithms designed for the Intel® Itanium® architecture.

- Division
- Square root
- Transcendental functions (log, sin etc.)

In some cases we prove exact rounding, in other cases a bound on the (relative or ulp) error.

Levels of verification

High-level algorithms assume correct behavior of some hardware primitives.



Proving my assumptions is someone else's job . . .

Characteristics of this work

The verification we're concerned with is somewhat atypical:

- Rather simple according to typical programming metrics, e.g. 5-150 lines of code, often no loops.
- Relies on non-trivial mathematics including number theory, analysis and special properties of floating-point rounding.

Tools that are often effective in other verification tasks, e.g. temporal logic model checkers, are of almost no use.

What do we need?

We need a general theorem proving system with:

- Ability to mix interactive and automated proof
- Programmability for domain-specific proof tasks
- A substantial library of pre-proved mathematics

Theorem provers for floating-point

There are several theorem provers that have been used for floating-point verification, some of it in industry:

- ACL2 (used at AMD)
- Coq
- HOL Light (used at Intel)
- PVS

All these are powerful systems with somewhat different strengths and weaknesses.

Interactive versus automatic

From interactive proof checkers to fully automatic theorem provers.

AUTOMATH (de Bruijn)

Mizar (Trybulec)

. . .

PVS (Owre, Rushby, Shankar)

. . .

ACL2 (Boyer, Kaufmann, Moore)

Vampire (Voronkov)

Mathematical versus industrial

Some provers are intended to formalize pure mathematics, others to tackle industrial-scale verification

AUTOMATH (de Bruijn)

Mizar (Trybulec)

...

...

PVS (Owre, Rushby, Shankar)

ACL2 (Boyer, Kaufmann, Moore)

Interactive theorem proving (1)

In practice, most interesting problems can't be automated completely:

- They don't fall in a practical decidable subset
- Pure first order proof search is not a feasible approach

In practice, we need an interactive arrangement, where the user and machine work together.

The user can delegate simple subtasks to pure first order proof search or one of the decidable subsets.

However, at the high level, the user must guide the prover.

In order to provide custom automation, the prover should be *programmable* — without compromising logical soundness.

Interactive theorem proving (2)

The idea of a more 'interactive' approach was already anticipated by pioneers, e.g. 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 that *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

However, constructing an effective and programmable combination is not so easy.

LCF

One successful solution was pioneered in Edinburgh LCF ('Logic of Computable Functions').

The same 'LCF approach' has been used for many other theorem provers.

- Implement in a strongly-typed functional programming language (usually a variant of ML)
- Make `thm` ('theorem') an abstract data type with only simple primitive inference rules
- Make the implementation language available for arbitrary extensions.

Gives a good combination of extensibility and reliability.

Now used in Coq, HOL, Isabelle and several other systems.

LCF kernel for first order logic (1)

Define type of first order formulas:

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

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

LCF kernel for first order logic (2)

Define some useful helper functions:

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

LCF kernel for first order logic (3)

```
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 =
            itlist2 (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 =
            itlist2 (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;;
```

Derived rules

The primitive rules are very simple. 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);;
```

While this process is tedious at the beginning, we can quickly reach the stage of automatic derived rules that

- Prove propositional tautologies
- Perform Knuth-Bendix completion
- Prove first order formulas by standard proof search and translation

Real LCF-style theorem provers like HOL have many powerful derived rules.

Principia Mathematica for the computer age?

LCF is based on the observation that all proofs in ‘ordinary’ mathematics can be reduced to sequences of formulas in a simple formal system.

- Frege’s *Begriffsschrift*
- Peano’s *Rivista di Matematica*
- Russell and Whitehead’s *Principia Mathematica*

Unfortunately it’s extremely painful doing so by hand.

But with the aid of a computer, it’s much more palatable, and mistakes unlikely.

HOL Light overview

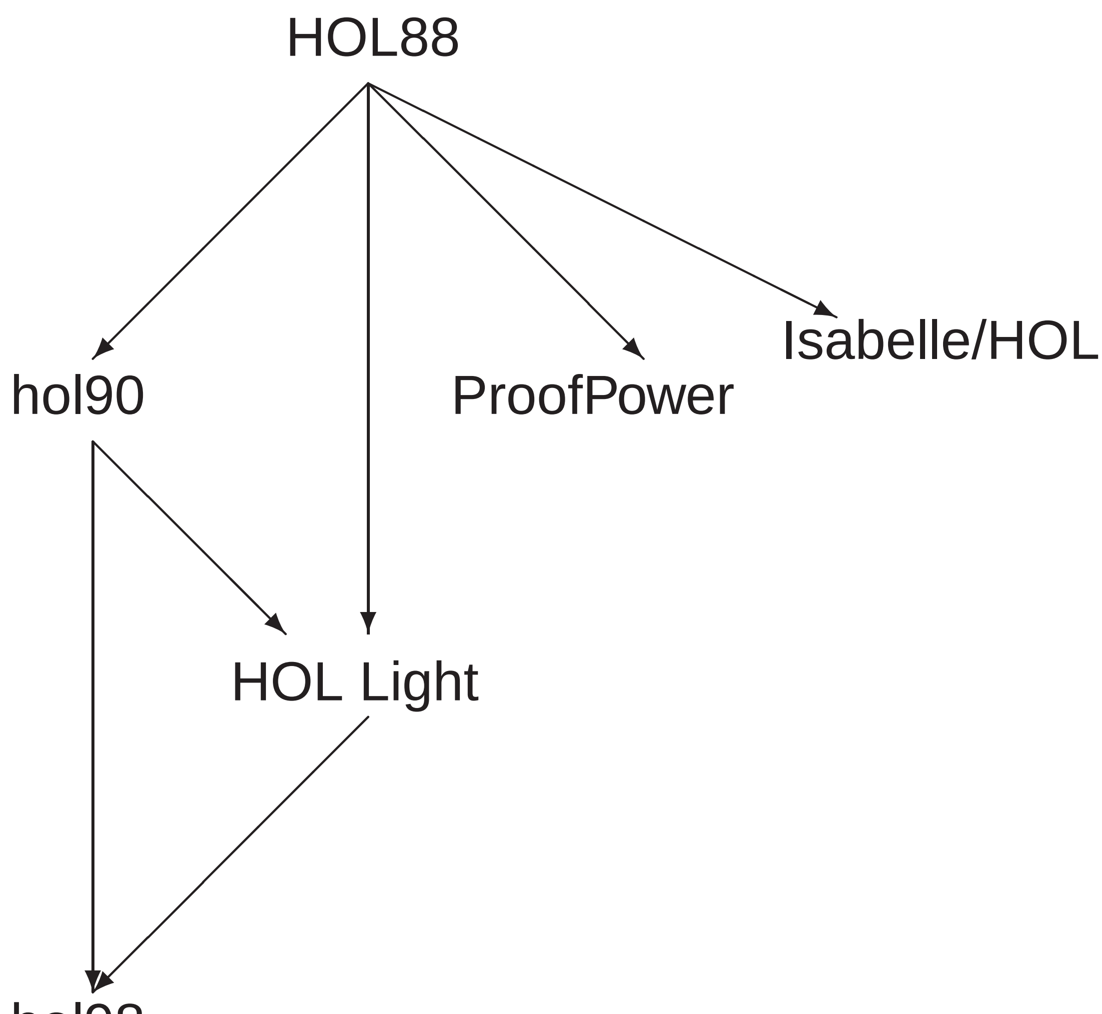
HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

HOL Light is designed to have a simple and clean logical foundation.

Versions in CAML Light and Objective CAML.

The HOL family DAG



HOL Light primitive rules (1)

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

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

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

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

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

HOL Light primitive rules (2)

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

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

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

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

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

Pushing the LCF approach to its limits

The main features of the LCF approach to theorem proving are:

- Reduce all proofs to a small number of relatively simple primitive rules
- Use the programmability of the implementation/interaction language to make this practical

Our work may represent the most “extreme” application of this philosophy.

- HOL Light’s primitive rules are very simple.
- Some of the proofs expand to about 100 million primitive inferences and can take many hours to check.

It is interesting to consider the scope of the LCF approach.

Some of HOL Light's derived rules

- 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

Breakdown to primitive inferences

```
REAL_ARITH
  ‘a <= x /\ b <= y /\
   abs(x - y) < abs(x - a) /\
   abs(x - y) < abs(x - b) /\
   (b <= x ==> abs(x - a) <= abs(x - b)) /\
   (a <= y ==> abs(y - b) <= abs(y - a))
   ==> (a = b)‘;;
```

Takes 1.3 seconds (on my laptop) and generates 40040 primitive inferences.

Real analysis details

Real analysis is especially important in our applications

- Definitional construction of real numbers
- Basic topology
- General limit operations
- Sequences and series
- Limits of real functions
- Differentiation
- Power series and Taylor expansions
- Transcendental functions
- Gauge integration

HOL floating point theory (1)

We have formalized a floating point theory in HOL with the precision as a parameter.

A floating point format is identified by a triple of natural numbers `fmt`.

The corresponding set of real numbers is `format(fmt)`, or ignoring the upper limit on the exponent, `iformat(fmt)`.

Floating point rounding returns a floating point approximation to a real number, ignoring upper exponent limits. More precisely

```
round fmt rc x
```

returns the appropriate member of `iformat(fmt)` for an exact value `x`, depending on the rounding mode `rc`, which may be one of `Nearest`, `Down`, `Up` and `Zero`.

HOL floating point theory (2)

For example, the definition of rounding down is:

```
|- (round fmt Down x = closest
        {a | a IN iformat fmt ∧ a <= x} x)
```

We prove a large number of results about rounding, e.g.

```
|- ¬(precision fmt = 0) ∧ x IN iformat fmt
   ⇒ (round fmt rc x = x)
```

that rounding is monotonic:

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

and that subtraction of nearby floating point numbers is exact:

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   a / &2 <= b ∧ b <= &2 * a ⇒ (b - a) IN iformat fmt
```

The $(1 + \epsilon)$ property

Designers often rely on clever “cancellation” tricks to avoid or compensate for rounding errors.

But many routine parts of the proof can be dealt with by a simple conservative bound on rounding error:

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

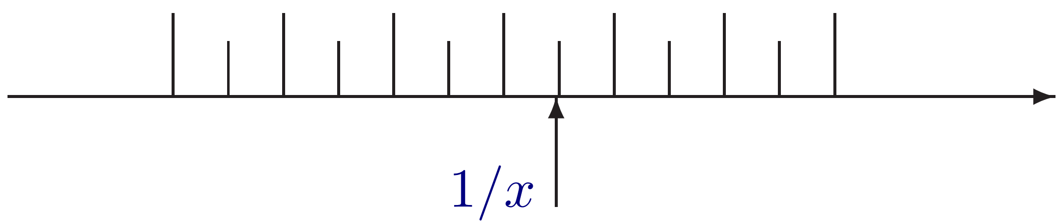
Derived rules apply this result to computations in a floating point algorithm automatically, discharging the conditions as they go.

Example 1: Difficult cases for reciprocals

Some algorithms for floating-point division, a/b, can be optimized for the special case of reciprocals ($a = 1$).

A direct analytic proof of the optimized algorithm is sometimes too hard because of the intricacies of rounding.

However, an analytic proof works for all but the 'difficult cases'.



These are floating-point numbers whose reciprocal is very close to another one, or a midpoint, making them trickier to round correctly.

Mixed analytical-combinatorial proofs

By finding a suitable set of ‘diffifult cases’, one can produce a proof by a mixture of analytical reasoning and explicit checking.

- Find the set of difficult cases S
- Prove the algorithm analytically for all $x \notin S$
- Prove the algorithm by explicit case analysis for $x \in S$

Quite similar to some standard proofs in mathematics, e.g. Bertrand’s conjecture.

This is particularly useful given that error bounds derived from the $1 + \epsilon$ property are highly conservative.

Finding difficult cases with factorization

After scaling to eliminate the exponents, finding difficult cases reduces to a straightforward number-theoretic problem.

A key component is producing the prime factorization of an integer *and* proving that the factors are indeed prime.

In typical applications, the numbers can be 49–227 bits long, so naive approaches based on testing all potential factors are infeasible.

The primality prover is embedded in a HOL derived rule `PRIME_CONV` that maps a numeral to a theorem asserting its primality or compositeness.

Certifying primality

We generate a 'certificate of primality' based on Pocklington's theorem:

```
|- 2 ≤ n ∧
   (n - 1 = q * r) ∧
   n ≤ q EXP 2 ∧
   (a EXP (n - 1) == 1) (mod n) ∧
   (∀p. prime(p) ∧ p divides q
        ⇒ coprime(a EXP ((n - 1) DIV p) - 1,n))
   ⇒ prime(n)
```

The certificate is generated 'extra-logically', using the factorizations produced by PARI/GP.

The certificate is then checked by formal proof, using the above theorem.

Typical results

```
0xFFFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFFFD 0xFE421D63446A3B34 0xFBFC17DFE0BEFF04 0xFB940B119826E598
0xFB0089D7241D10FC 0xFA0BF7D05FBE82FC 0xF912590F016D6D04 0xF774DD7F912E1F54 0xF7444DFBF7B20EAC
0xF39EB657E24734AC 0xF36EE790DE069D54 0xF286AD7943D79434 0xEDF09CCC53942014 0xEC4B058D0F7155BC
0xEC1CA6DB6D7BD444 0xE775FF856986AE74 0xE5CB972E5CB972E4 0xE58469F0234F72C4 0xE511C4648E2332C4
0xE3FC771FE3B8FF1C 0xE318DE3C8E6370E4 0xE23B9711DCB88EE4 0xE159BE4A8763011C 0xDF738B7CF7F482E4
0xDEE256F712B7B894 0xDEE24908EDB7B894 0xDE86505A77F81B25 0xDE03D5F96C8A976C 0xDDFF059997C451E5
0xDB73060F0C3B6170 0xDB6DB6DB6DB6DB6C 0xDB6DA92492B6DB6C 0xDA92B6A4ADA92B6C 0xD9986492DD18DB7C
0xD72F32D1C0CC4094 0xD6329033D6329033 0xD5A004AE261AB3DC 0xD4D43A30F2645D7C 0xD33131D2408C6084
0xD23F53B88EADABB4 0xCCCE6669999CCCD0 0xCCCE666666633330 0xCCCCCCCCCCCCCCD0 0xCBC489A1DBB2F124
0xCB21076817350724 0xCAF92AC7A6F19EDC 0xC9A8364D41B26A0C 0xC687D6343EB1A1F4 0xC54EDD8E76EC6764
0xC4EC4EC362762764 0xC3FCF61FE7B0FF3C 0xC3FCE9E018B0FF3C 0xC344F8A627C53D74 0xC27B1613D8B09EC4
0xC27B09EC27B09EC4 0xC07756F170EAFBEC 0xBDF3CD1B9E68E8D4 0xBD5EAF57ABD5EAF4 0xBCA1AF286BCA1AF4
0xB9B501C68DD6D90C 0xB880B72F050B57FC 0xB85C824924643204 0xB7C8928A28749804 0xB7A481C71C43DDFC
0xB7938C6947D97303 0xB38A7755BB835F24 0xB152958A94AC54A4 0xAFF5757FABABFD5C 0xAF4D99ADFEFCAAFC
0xAF2B32F270835F04 0xAE235074CF5BAE64 0xAE0866F90799F954 0xADCC548E46756E64 0xAD5AB56AD5AB56AC
0xAD5AAA952AAB56AC 0xAB55AAD56AB55AAC 0xAAAAB55555AAAAAC 0xAAAAAAAAAAAAAAAC 0xAAAAA00000555554
0xA93CFF3E629F347D 0xA80555402AAA0154 0xA8054ABFD5AA0154 0xA7F94913CA4893D4 0xA62E84F95819C3BC
0xA5889F09A0152C44 0xA4E75446CA6A1A44 0xA442B4F8DCDEF5BC 0xA27E096B503396EE 0x9E9B8FFFFFD8591C
0x9E9B8B0B23A7A6E4 0x9E7C6B0C1CA79F1C 0x9DFC78A4EEEE4DCB 0x9C15954988E121AB 0x9A585968B4F4D2C4
0x99D0C486A0FAD481 0x99B831EEE01FB16C 0x990C8B8926172254 0x990825E0CD75297C 0x989E556CADAC2D7F
0x97DAD92107E19484 0x9756156041DBBA94 0x95C4C0A72F501BDC 0x94E1AE991B4B4EB4 0x949DE0B0664FD224
0x942755353AA9A094 0x9349AE0703CB65B4 0x92B6A4ADA92B6A4C 0x9101187A01C04E4C 0x907056B6E018E1B4
0x8F808E79E77A99C4 0x8F64655555317C3C 0x8E988B8B3BA3A624 0x8E05E117D9E786D5 0x8BEB067D130382A4
0x8B679E2B7FB0532C 0x887C8B2B1F1081C4 0x8858CCDCA9E0F6C4 0x881BB1CAB40AE884 0x87715550DCDE29E4
0x875BDE4FE977C1EC 0x86F71861FDF38714 0x85DBEE9FB93EA864 0x8542A9A4D2ABD5EC 0x8542A150A8542A14
0x84BDA12F684BDA14 0x83AB6A090756D410 0x83AB6A06F8A92BF0 0x83A7B5D13DAE81B4 0x8365F2672F9341B4
0x8331C0CFE9341614 0x82A5F5692FAB4154 0x8140A05028140A04 0x8042251A9D6EF7FC
```

Example 2: polynomial approximation errors

Many transcendental functions are ultimately approximated by polynomials.

This usually follows some initial reduction step to ensure that the argument is in a small range, say $x \in [a, b]$.

The *minimax* polynomials used have coefficients found numerically to minimize the maximum error over the interval.

In the formal proof, we need to prove that this is indeed the maximum error, say $\forall x \in [a, b]. \ |sin(x) - p(x)| \leq 10^{-62}|x|$.

By using a Taylor series with much higher degree, we can reduce the problem to bounding a pure polynomial with rational coefficients over an interval.

Bounding functions

If a function f differentiable for $a \leq x \leq b$ has the property that $f(x) \leq K$ at all points of zero derivative, as well as at $x = a$ and $x = b$, then $f(x) \leq K$ everywhere.

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl (f' x)) x) ∧
   f(a) <= K ∧ f(b) <= K ∧
   (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0)
        ⇒ f(x) <= K)
   ⇒ (∀x. a <= x ∧ x <= b ⇒ f(x) <= K)
```

Hence we want to be able to isolate zeros of the derivative (which is just another polynomial).

Isolating derivatives

For any differentiable function f, $f(x)$ can be zero only at one point between zeros of the derivative $f'(x)$.

More precisely, if $f'(x) \neq 0$ for $a < x < b$ then if $f(a)f(b) \geq 0$ there are no points of $a < x < b$ with $f(x) = 0$:

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl f'(x))(x)) ∧
   (∀x. a < x ∧ x < b ⇒ ¬(f'(x) = &0)) ∧
   f(a) * f(b) >= &0
   ⇒ ∀x. a < x ∧ x < b ⇒ ¬(f(x) = &0)
```

Bounding and root isolation

This gives rise to a recursive procedure for bounding a polynomial and isolating its zeros, by successive differentiation.

```
|- (∀x. a <= x ∧ x <= b ⇒ (f diffl (f' x)) x) ∧
   (∀x. a <= x ∧ x <= b ⇒ (f' diffl (f'' x)) x)
∧
   (∀x. a <= x ∧ x <= b ⇒ abs(f''(x)) <= K) ∧
   a <= c ∧ c <= x ∧ x <= d ∧ d <= b ∧ (f'(x) = &0)
   ⇒ abs(f(x)) <= abs(f(d)) + (K / &2) * (d - c) pow 2
```

At each stage we actually produce HOL theorems asserting bounds and the enclosure properties of the isolating intervals.

Success and failure

HOL Light's extensive mathematical infrastructure and complete programmability make it ideally suited for such applications.

In the hands of a skilled user — for example its author — it can be very productive. But it's not easy for beginners:

- User is confronted with a full (and probably unfamiliar) programming language.
- Many inference rules and pre-proved theorems available, and it takes a long time to learn how to use them all.

How can we improve matters? One idea is to pass to a more *declarative* style of proof script.

Proof styles

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable

Procedural proof example

```
REPEAT GEN_TAC THEN REWRITE_TAC[contl; LIM; REAL_SUB_RZERO] THEN
BETA_TAC THEN DISCH_TAC THEN X_GEN_TAC "e:real" THEN
DISCH_TAC THEN
FIRST_ASSUM(UNDISCH_TAC o assert is_conj o concl) THEN
DISCH_THEN(CONJUNCTS_THEN MP_TAC) THEN
DISCH_THEN(\th. FIRST_ASSUM(MP_TAC o MATCH_MP th)) THEN
DISCH_THEN(X_CHOOSE_THEN "d:real" STRIP_ASSUME_TAC) THEN
DISCH_THEN(MP_TAC o SPEC "d:real") THEN ASM_REWRITE_TAC[] THEN
DISCH_THEN(X_CHOOSE_THEN "c:real" STRIP_ASSUME_TAC) THEN
EXISTS_TAC "c:real" THEN ASM_REWRITE_TAC[] THEN
X_GEN_TAC "h:real" THEN DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN
ASM_CASES_TAC "&0 < abs(f(x + h) - f(x))" THENL
 [UNDISCH_TAC "&0 < abs(f(x + h) - f(x))" THEN
  DISCH_THEN(\th. DISCH_THEN(MP_TAC o CONJ th)) THEN
  DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN
  REWRITE_TAC[REAL_SUB_ADD2];
  UNDISCH_TAC "~(&0 < abs(f(x + h) - f(x)))" THEN
  REWRITE_TAC[GSYM ABS_NZ; REAL_SUB_0] THEN
  DISCH_THEN SUBST1_TAC THEN
  ASM_REWRITE_TAC[REAL_SUB_REFL; ABS_0]]);;
```

Declarative proof example

```
let f be A->A;
assume L:antecedent;
antisymmetry: (!x y. x <= y /\ y <= x ==> (x = y)) by L;
transitivity: (!x y z. x <= y /\ y <= z ==> x <= z) by L;
monotonicity: (!x y. x <= y ==> f x <= f y) by L;
least_upper_bound:
    (!X. ?s:A. (!x. x IN X ==> s <= x) /\
               (!s’. (!x. x IN X ==> s’ <= x) ==> s’ <= s)) by L;
set Y_def: Y = {b | f b <= b};
Y_thm: !b. b IN Y = f b <= b by Y_def,IN_ELIM_THM,BETA_THM;
consider a such that
    lub: (!x. x IN Y ==> a <= x) /\
         (!a’. (!x. x IN Y ==> a’ <= x) ==> a’ <= a)
    by least_upper_bound;
take a;
now let b be A;
    assume b_in_Y: b IN Y;
    then L0: f b <= b by Y_thm;
    a <= b by b_in_Y, lub;
    so f a <= f b by monotonicity;
    hence f a <= b by L0, transitivity;
    end;
so Part1: f(a) <= a by lub;
so f(f(a)) <= f(a) by monotonicity;
so f(a) IN Y by Y_thm;
so a <= f(a) by lub;
hence thesis by Part1, antisymmetry;
```

The rise of declarative proof

Mizar pioneered the declarative style of proof. It was subsequently incorporated into other provers:

- Mizar mode for HOL (Harrison)
- DECLARE system (Syme)
- SPL system (Zammitt)
- Isar mode for Isabelle (Wenzel)

A good ‘by’ is hard to find

The main difficulty is arriving at a ‘by’ that can fill in gaps in the proof at a reasonable level.

Mizar has a first order prover that is very limited in its ability to deal with quantifiers. Like Abrial’s B prover, it doesn’t even use unification.

On the other hand, it is very good at dealing with equality, and it works surprisingly well in practice.

Together with Freek Wiedijk, we have reverse-engineered it for HOL.

But much remains to be done to find a prover that smoothly fills most ‘obvious’ gaps — the Mizar prover is just one component.

Not obvious enough?

For some purposes, we might want to avoid making the prover too powerful if it results in a proof that is difficult for a human to grasp.

$$
\begin{aligned}
&(\forall x\ y\ z.\ P(x,y) \land P(y,z) \Rightarrow P(x,z)) \land \\
&(\forall x\ y\ z.\ Q(x,y) \land Q(y,z) \Rightarrow Q(x,z)) \land \\
&(\forall x\ y.\ Q(x,y) \Rightarrow Q(y,x)) \land \\
&(\forall x\ y.\ P(x,y) \lor Q(x,y)) \\
&\Rightarrow (\forall x\ y.\ P(x,y)) \lor (\forall x\ y.\ Q(x,y))
\end{aligned}
$$

The above, due to Łoś, is trivial for most major first order provers, but probably not for most people.

Summary

- We need general theorem proving for some applications; it can be based on first order set theory or higher-order logic.
- In practice, we need a combination of interaction and automation for difficult proofs.
- LCF gives a good way of realizing a combination of soundness and extensibility.
- Different proof styles may be preferable, and they can be supported on top of an LCF-style core.
- A declarative proof style supported by a powerful ‘by’ prover probably offers the best hope of ‘theorem proving for the masses’.