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:

- 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.
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.
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 \texttt{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.
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;;
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;;
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(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 $p \Rightarrow p$:

```ml
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

HOL88

hol90

hol98

ProofPower

HOL Light

Isabelle/HOL
HOL Light primitive rules (1)

\[ \vdash t = t \quad \text{REFL} \]

\[ \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u} \quad \text{TRANS} \]

\[ \frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)} \quad \text{MK_COMB} \]

\[ \frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x. s) = (\lambda x. t)} \quad \text{ABS} \]

\[ \vdash (\lambda x. t)x = t \quad \text{BETA} \]
HOL Light primitive rules (2)

\[
\{ p \} \vdash p
\]

\[
\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}
\]

EQ_MP

\[
\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{ q \}) \cup (\Delta - \{ p \}) \vdash p = q}
\]

DEDUCT_ANTISYM_RULE

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

INST

\[
\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]}
\]

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

\[ \begin{align*}
'a <= x & \land b <= y \\
\text{abs}(x - y) & < \text{abs}(x - a) \\
\text{abs}(x - y) & < \text{abs}(x - b) \\
(b <= x & \implies \text{abs}(x - a) <= \text{abs}(x - b)) & \\
(a <= y & \implies \text{abs}(y - b) <= \text{abs}(y - a)) & \\
\implies (a = b) \;';
\end{align*} \]

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 $\text{format}(fmt)$, or ignoring the upper limit on the exponent, $\text{ifformat}(fmt)$.

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

$$\text{round fmt rc x}$$

returns the appropriate member of $\text{ifformat}(fmt)$ 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:

\[ |- (\text{round fmt Down } x = \text{closest} \{a \mid a \in \text{iformat fmt} \land a \leq x\}) \]

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

\[ |- \neg(\text{precision fmt } = 0) \land x \in \text{iformat fmt} \Rightarrow (\text{round fmt } \text{rc } x = x) \]

that rounding is monotonic:

\[ |- \neg(\text{precision fmt } = 0) \land x \leq y \Rightarrow \text{round fmt } \text{rc } x \leq \text{round fmt } \text{rc } y \]

and that subtraction of nearby floating point numbers is exact:

\[ |- a \in \text{iformat fmt} \land b \in \text{iformat fmt} \land a / \&2 \leq b \land b \leq \&2 \times a \Rightarrow (b - a) \in \text{iformat fmt} \]
The \((1 + \varepsilon)\) 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:

\[\begin{align*}
\text{|- normalizes fmt x } & \land \\
\neg (\text{precision fmt} = 0) & \\
\Rightarrow & \exists e. \text{abs}(e) \leq \frac{\mu rc}{2 \text{ pow}(\text{precision fmt - 1})} \land \\
\text{round fmt rc x} & = x \ast (1 + e)
\end{align*}\]

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 ‘difficult 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 $x \not\in S$
- 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:

\[
- 2 \leq n \land \\
(n - 1 = q \times r) \land \\
n \leq q^{\text{EXP} 2} \land \\
(a^{\text{EXP} (n - 1) \equiv 1} \pmod{n}) \land \\
(\forall p. \text{prime}(p) \land p \text{ divides } q \\
\quad \Rightarrow \text{coprime}(a^{\text{EXP} ((n - 1) \div p) - 1}, n)) \\
\Rightarrow \text{prime}(n)
\]

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 0xE5846F90234F72C4 0xE511C468E233C4
0xE3FC771FE3BFF1C 0xE318DE3C8E6370E4 0xE23B9711DB88EE4 0xE159BE4A8763011C
0xDFF387B7CF7F48E4
0xDEE256F712B7B894 0xDEE24908EDB7B894 0xDE8650A77F81B25 0xDE03D5F96C8A976C
0xCB21076D81B9249 0xC9A8364D41B26A0C 0xC867D6343EB1A1F4 0xC64EDD8E76EC6764
0xC4EC4EC362762764 0xC3FCF61FE700F3C 0xC152958A94AC544 0xC07756F170EAFBEC
0xBDF3CD1B9E68E8D4 0xBDD5EAF57ABD5EAF4 0xBAFC489A1DBB2F124
0xB9B501C68DD69D0C 0xB880B72F050B57FC 0xB7C8928A28749804 0xB7A481C71C43DDFC
0xB7938C694D97303 0xB38A775594AC544 0xAFF5757FABABFD5C 0xAFC48913CA4893D4
0x9E9B8B0B23A7A6E4 0x9E7C6B0C1CA79F1C 0x9E05E117D9E786D5 0x9DDBE4FE977C1EC
0x8F808E7977A99C4 0x8E988B8B3AA3A624 0x881BB1CAB40AE884 0x875BDE4FE977C1EC
0x875BDE4FE977C1EC 0x86F71861FDF38714 0x85DBB5793EA894 0x83A6A06F8AA92BF0
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.

$$\neg (\forall x. \ a \leq x \land x \leq b \Rightarrow (f \ \text{diff} \ (f'(x))) \ x) \land$$
$$f(a) \leq K \land f(b) \leq K \land$$
$$\neg (\forall x. \ a \leq x \land x \leq b \land (f'(x) = 0))$$
$$\Rightarrow f(x) \leq K$$
$$\Rightarrow (\forall x. \ a \leq x \land x \leq b \Rightarrow f(x) \leq K)$$

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 $f(x) = 0$:

$$\neg (\forall x. a \leq x \land x \leq b \Rightarrow (f \text{ diff} f'(x))(x)) \land$$
$$\neg (\forall x. a < x \land x < b \Rightarrow \neg(f'(x) = &0)) \land$$
$$f(a) \ast f(b) \geq &0$$

$$\Rightarrow \forall x. a < x \land x < b \Rightarrow \neg(f(x) = &0)$$
Bounding and root isolation

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

\[
\begin{align*}
\forall x. a \leq x \land x \leq b \implies (f \text{ diff } f'(x)) \land \\
\forall x. a \leq x \land x \leq b \implies (f' \text{ diff } f''(x)) \land \\
\forall x. a \leq x \land x \leq b \implies \text{abs}(f''(x)) \leq K) \land \\
a \leq c \land c \leq x \land x \leq d \land d \leq b \land (f'(x) = 0) \\
\implies \text{abs}(f(x)) \leq \text{abs}(f(d)) + (K / 2) \times (d - c)^2
\end{align*}
\]

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[cont1; 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 $A\rightarrow A$;
assume L: antecedent;
antisymmetry: $(\forall x\ y. x \leq y \land y \leq x \Rightarrow (x = y))$ by L;
transitivity: $(\forall x\ y\ z. x \leq y \land y \leq z \Rightarrow x \leq z)$ by L;
monotonicity: $(\forall x\ y. x \leq y \Rightarrow f\ x \leq f\ y)$ by L;
least_upper_bound:
  $(\forall X. \exists s: A. (\forall x. x \in X \Rightarrow s \leq x) \land
   (\forall s'. (\forall x. x \in X \Rightarrow s' \leq x) \Rightarrow s' \leq s))$ by L;
set $Y_{\text{def}}: Y = \{b | f\ b \leq b\}$;
$Y_{\text{thm}}: \forall b. b \in Y = f\ b \leq b$ by $Y_{\text{def}}, \text{IN_ELIM_THM}, \text{BETA_THM}$;
consider a such that
  lub: $(\forall x. x \in Y \Rightarrow a \leq x) \land
   (\forall a'. (\forall x. x \in Y \Rightarrow a' \leq x) \Rightarrow a' \leq a)$
  by least_upper_bound;
take a;
now let b be $A$;
assume $b_{\text{in}\ Y}: b \in Y$;
then L0: $f\ b \leq b$ by $Y_{\text{thm}}$;
a \leq b by $b_{\text{in}\ Y}, \text{lub}$;
so $f\ a \leq f\ b$ by monotonicity;
hence $f\ a \leq b$ by L0, transitivity;
end;
so Part1: $f(a) \leq a$ by lub;
so $f(f(a)) \leq f(a)$ by monotonicity;
so $f(a) \in Y$ by $Y_{\text{thm}}$;
so a \leq 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.

\[
(\forall x \ y \ z. \ P(x, y) \land P(y, z) \Rightarrow P(x, z)) \land \\
(\forall x \ y \ z. \ Q(x, y) \land Q(y, z) \Rightarrow Q(x, z)) \land \\
(\forall x \ y. \ Q(x, y) \Rightarrow Q(y, x)) \land \\
(\forall x \ y. \ P(x, y) \lor Q(x, y)) \\
\Rightarrow (\forall x \ y. \ P(x, y)) \lor (\forall x \ y. \ Q(x, y))
\]

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’.