HOL Light and its use in verification

John Harrison

University of Cambridge

(visiting TU München)

- History and evolution
- Primitive basis
- Derived rules and mathematical theories
- Mizar mode
- Floating point verification

HOL Light's lineage

HOL Light is a programmable interactive theorem prover based on classical higher order logic (polymorphic simple type theory). It has evolved via:

- Edinburgh LCF (Milner et al.)
- Cambridge LCF (Paulson)
- HOL (Gordon, Melham)
- hol90 (Slind)

Other LCF-style systems include:

- Nuprl (Constable et al.)
- Coq (Huet et al.)
- Isabelle (Paulson)

The spectrum of theorem provers

AUTOMATH (de Bruijn)

Stanford LCF (Milner)

Mizar (Trybulec)

. . .

. . .

PVS (Owre, Rushby, Shankar)

. . .

. . .

NQTHM (Boyer, Moore)

Otter (McCune)

The LCF approach

The key ideas are:

- All theorems created by low-level primitive rules.
- Guaranteed by using an abstract type of theorems; no need to store proofs.
- ML available for implementing derived rules by arbitrary programming.

This gives advantages of reliability and extensibility. The system's source code can be completely open. The user controls the means of production (of theorems). To improve efficiency one can:

- Encapsulate reasoning in single theorems.
- Separate proof search and proof checking.

Primitive rules (1)

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

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

(Up to alpha-equivalence.)

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

(Provided types agree, e.g. $s : \sigma \to \tau$ and $u : \sigma$.)

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

(Provided x is not free in the assumptions Γ.)

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

Primitive rules (2)

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

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

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

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

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

Principles of definition

All theories in HOL Light are derived from three axioms using only primitive rules plus extension by definitions of new constants and new types.

For example, the other logical constants are defined as follows:

$$
\begin{array}{rcl}
\top & = & (\lambda x.\ x) = (\lambda x.\ x) \\
\wedge & = & \lambda p.\ \lambda q.\ (\lambda f.\ f\ p\ q) = (\lambda f.\ f\ \top\ \top) \\
\Rightarrow & = & \lambda p.\ \lambda q.\ p \wedge q = p \\
\forall & = & \lambda P.\ P = \lambda x.\ \top \\
\exists & = & \lambda P.\ \forall Q.\ (\forall x.\ P(x) \Rightarrow Q) \Rightarrow Q \\
\vee & = & \lambda p.\ \lambda q.\ \forall r.\ (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r \\
\bot & = & \forall P.\ P \\
\neg & = & \lambda t.\ t \Rightarrow \bot \\
\exists! & = & \lambda P.\ \exists P \wedge \forall x.\ \forall y.\ P\ x \wedge P\ y \Rightarrow (x = y)
\end{array}
$$

Mathematical axioms

HOL Light's mathematics is based on one new operator ε and one new type ind.

- The axiom of extensionality: $\forall t.\, (\lambda x.\, t\ x) = t$.
- The axiom of choice via the Hilbert operator: $\forall P,\ x.\, P\ x \Rightarrow P(\varepsilon P)$.
- The axiom of infinity for the type `ind`: $\exists f : ind \to ind.\, \texttt{ONE\_ONE}\ \ f \wedge \neg(\texttt{ONTO}\ \ f)$.

That's all! After that, HOL Light tries to conform to the LCF ideal by deriving everything via definitional expansion.

There are quite a lot of derived rules that the user can invoke without bothering about their internal workings. But internally they decompose to primitive inferences.

The main 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.
- Tools for definition of (infinitary, mutually) inductive relations.
- Tools for definition of (mutually) recursive datatypes
- Linear arithmetic decision procedures over $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{N}$.
- Differentiator for real functions.

Mathematical theories

- Basic set theory, e.g. definition by recursion over finite sets.
- Forms of the Axiom of Choice, e.g. Zorn's Lemma and wellordering principle.
- Basic theory of lists, e.g. lengths, mappings, list iteration
- Integers and real numbers (constructed)
- Elementary real analysis, e.g.
  - Basic topology and general limits
  - Sequences and series
  - Limits of real functions
  - Differentiation
  - Power series and Taylor expansions
  - Transcendental functions
  - Gauge integration

Some applications

Apart from floating point verification and real analysis, we've played around with:

- Formalization of simple embedded programming languages, e.g. Dijkstra's guarded command language.
  - Parsing and prettyprinting support
  - Operational semantics
  - Weakest preconditions
  - Verification condition generation
- Formalization of theorems from logic, e.g.
  - Completeness of a proof system for propositional logic
  - Compactness, Uniformity and Löwenheim-Skolem for first order logic
  - Tarski's theorem on the undefinability of truth in first order number theory

Mizar Mode

The standard HOL proof styles (whether forward or backward) are highly *procedural*. They require a certain amount of 'programming' from the user.

We also provide a more *declarative* proof style, as used in Mizar. The machine fills in the gaps in the proof for us with explicit inference steps. Here's a proof of $\forall x.\, 0 \leq x \Rightarrow ln(1+x) \leq x$:

```
let x be real;
assume &0 <= x;
then &0 < &1 + x by arithmetic;
so exp(ln(&1 + x)) = &1 + x by EXP_LN;
so suffices to show &1 + x <= exp(x)
  by EXP_MONO_LE;
thus thesis by EXP_LE_X
```

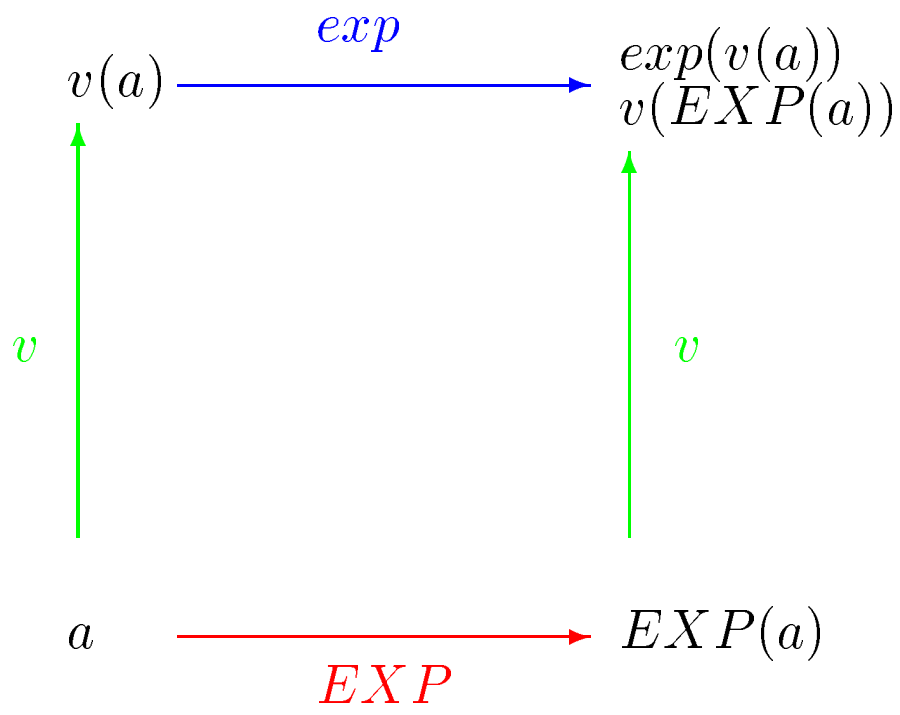
So far we haven't used this much, but in the future we hope to refine it in order to make some pure mathematics proofs at least more readable.

Floating point verification

- Floating point algorithms are fairly small, but often complicated mathematically.
- There have been errors in commercial systems, e.g. the Pentium FDIV bug in 1994.
- In the case of transcendental functions it's difficult even to say what correctness means.
- Verification using model checkers is difficult because of the need for mathematical apparatus.
- It can even be difficult using theorem provers since not many of them have good theories of real numbers etc.

Floating point correctness

We want to specify the correctness according to the following diagram:



We measure the difference between $v(EXP(a))$ and $exp(v(a))$ in 'units in the last place' of $EXP(a)$.

Our implementation language

This includes the following constructs:

command = `variable :=` *expression*
| *command* `;` *command*
| `if` *expression* `then` *command*
`else` *command*
| `if` *expression* `then` *command*
| `while` *expression* `do` *command*
| `do` *command* `while` *expression*
| `skip`
| { *expression*}

We have a simple relational semantics in HOL, and derive weakest preconditions and total correctness rules. We then prove total correctness via VC generation.

The idea is that this language can be formally linked to C, Verilog, Handel, ...

Sketch of the algorithm

The algorithm we verify is taken from a paper by Tang in *ACM Transactions on Mathematical Software*, 1989.

Similar techniques are widely used for floating point libraries, and, probably, for hardware implementations.

The algorithm relies on a table of precomputed constants. Tang's paper gives actual values as hex representations of IEEE numbers.

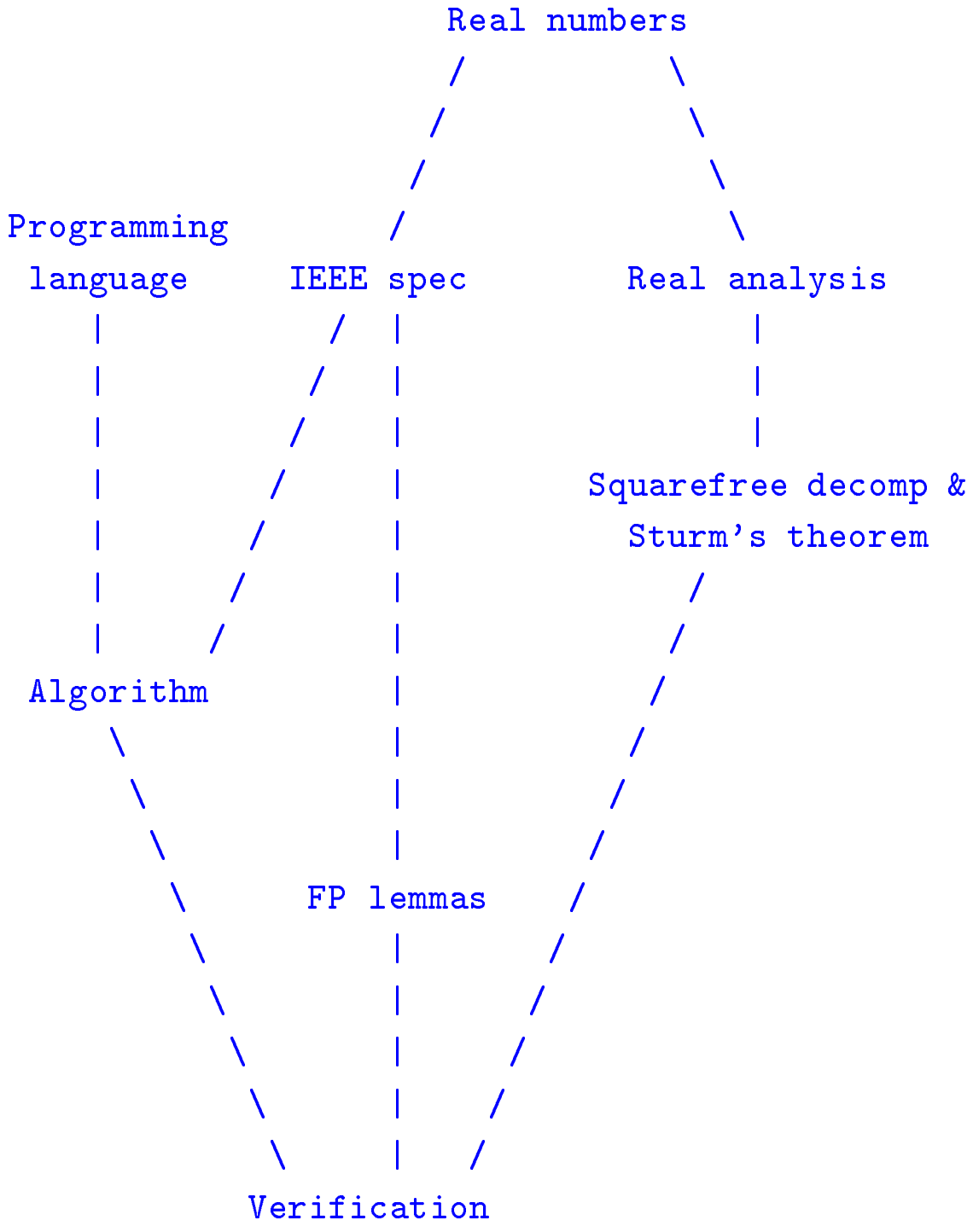
We can split the operations into three steps:

- Perform range reduction
- Use polynomial approximation
- Reconstruct answer using tables

Code for the algorithm

```
if Isnan(X) then E := X
else if X == Plus_infinity then E := Plus_infinity
else if X == Minus_infinity then E := Plus_zero
else if abs(X) > THRESHOLD_1 then
  if X > Plus_zero then E := Plus_infinity
  else E := Plus_zero
else if abs(X) < THRESHOLD_2 then E := Plus_one + X
else
 (N := INTRND(X * Inv_L);
  N2 := N % Int_32;
  N1 := N - N2;
  if abs(N) >= Int_2e9 then
    R1 := (X - Tofloat(N1) * L1) - Tofloat(N2) * L1
  else
    R1 := X - Tofloat(N) * L1;
  R2 := Tofloat(--N) * L2;
  M := N1 / Int_32;
  J := N2;
  R := R1 + R2;
  Q := R * R * (A1 + R * A2);
  P := R1 + (R2 + Q);
  S := S_Lead(J) + S_Trail(J);
  E1 := S_Lead(J) + (S_Trail(J) + S * P);
  E := Scalb(E1,M)
 )
```

Organization of HOL proof



Floating point lemmas (1)

A large part of the proof (though not the most difficult part!) involves analyzing the way rounding errors build up, and how in special situations, the error can be zero.

We define the error `error(x)` resulting from rounding a real number to a floating point value.

Because of the regular way in which the operations are defined, all the operations then relate to their abstract mathematical counterparts according to the same pattern:

```
|- Finite(a) ∧ Finite(b) ∧
   abs(Val(a) + Val(b)) < threshold(float_format)
   ⇒ Finite(a + b) ∧
       (Val(a + b) = (Val(a) + Val(b)) +
       error(Val(a) + Val(b)))
```

The comparisons are even more straightforward:

```
|- Finite(a) ∧ Finite(b)
   ⇒ (a < b = Val(a) < Val(b))
```

Floating point lemmas (2)

We have several lemmas quantifying the error, of which the most useful is the following:

```
|- abs(x) < threshold(float_format) ∧
   abs(x) < (&2 pow j / &2 pow 125)
   ⇒ abs(error(x)) <= &2 pow j / &2 pow 150
```

There are many important situations, however, where the operations are exact, because the result is exactly representable. Trivially, for example, the negation and absolute value functions are always exact:

```
|- Finite(a)
   ⇒ Finite(abs(a)) ∧ (Val(abs(a)) = abs(Val(a)))
```

Also, if a result only has 24 significant digits (modulo some care in the denormal case), then it is exactly representable:

```
|- (abs(x) = (&2 pow e / &2 pow 149) * &k) ∧
   k < 2 EXP 24 ∧ e < 254
   ⇒ ∃a. Finite(a) ∧ (Val(a) = x)
```

Floating point lemmas (3)

Any calculation whose result is exactly representable has an error of zero:

```
|- Finite(a) ∧ Finite(b) ∧
   Finite(c) ∧ (Val(c) = Val(a) * Val(b))
   ⇒ Finite(a * b) ∧
       (Val(a * b) = Val(a) * Val(b))
```

Another important case of exact operations is subtraction of nearby values with the same sign:

```
|- Finite(a) ∧ Finite(b) ∧
   &2 * abs(Val(a) - Val(b)) <= abs(Val(a))
   ⇒ Finite(a - b) ∧
       (Val(a - b) = Val(a) - Val(b))
```

This is a classic result in floating point error analysis.

We also have a type of machine integers, and prove various obvious results about how the arithmetic operations on those work.

Error in polynomial approximation

This part is tricky. In brief, these are the steps:

- Prove that the error in a high-order Taylor series is much better than we need.
- Consider the difference between this and the minimax polynomial actually used.
- Locate the zeros of (the squarefree decomposition of) its derivative.
- Prove using Sturm's theorem that these are all the zeros.
- Hence get a bound on the error by evaluation at the endpoints of the interval and the points of zero derivative, using some elementary real analysis.

Tang makes a small slip over the necessary interval.

The final result

Under the various 'definitional' assumptions, we confirm Tang's bottom-line result:

```
(Isnan(X) ⇒ Isnan(E)) ∧
(X == Plus_infinity ∨
 Finite(X) ∧
 exp(Val X) >= threshold(float_format)
 ⇒ E == Plus_infinity) ∧
(X == Minus_infinity ⇒ E == Plus_zero) ∧
(Finite(X) ∧ exp(Val X) < threshold(float_format)
 ⇒ Isnormal(E) ∧
     abs(Val(E) - exp(Val X))
     < (&54 / &100) * Ulp(E) ∨
     (Isdenormal(E) ∨ Iszero(E)) ∧
     abs(Val(E) - exp(Val X))
     < (&77 / &100) * Ulp(E))
```

This is somewhat more explicit than Tang's statement regarding overflow.

Conclusions

- HOL Light successfully implements the LCF approach to theorem proving. Its primitives are very simple, but its derived rules are enough for some non-toy proofs.
- Particular proofs or verifications tend to point up the weak and strong points of systems. For example
  - Programmability and the automation of linear arithmetic are invaluable, as is the presence of a decent real analysis theory.
  - Better tools are needed for nonlinear reasoning. Explicit calculations are still very slow.
- In the floating point proof, we confirm (and strengthen) the main results of the hand proof. But we detect a few slips and uncover subtle issues. This class of proofs is a good target for verification.