HOL Light: A Tutorial Introduction

John Harrison

University of Cambridge

(Åbo Akademi University)

- History and evolution
- Quick rundown of features
- Real analysis theory
- Programming language semantics
- Mizar mode
- CORDIC algorithm example

HOL Light's lineage

HOL Light 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.

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.
- 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.

Real analysis theory (1)

- 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

Real analysis theory (2)

There are lots of concrete theorems, e.g.

```
|- abs(abs x - abs y) <= abs (x - y)

|- sin(x + y) =
   sin(x) * cos(y) + cos(x) * sin(y)

|- tan(&n * pi) = &0

|- &0 < x /\ &0 < y
   ==> (ln(x / y) = ln(x) - ln(y))
```

Real analysis theory (3)

and many general ones:

```
|- f contl x /\ g contl (f x)
   ==> (\x. g(f x)) contl x

|- a <= b /\
   (f(a) <= y /\ y <= f(b)) /\
   (!x. a <= x /\ x <= b ==> f contl x)
   ==> (?x. a <= x /\ x <= b /\ (f(x) = y))

|- (f diffl l)(g x) /\ (g diffl m)(x)
   ==> ((\x. f(g x)) diffl (l * m))(x)

|- a <= b /\
   (!x. a <= x /\ x <= b
         ==> (f diffl f'(x))(x))
   ==> Dint(a,b) f' (f(b) - f(a))
```

Our Programming Language (1)

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*}
| [*expression*]

The language is semantically embedded in HOL using standard techniques.

Our Programming Language (2)

We can verify the total correctness of programs according to given pre and post-conditions.

```
|- correct p c q
```

corresponds to the standard total correctness assertion $[p]\ c\ [q]$, i.e. a command c, executed in a state satisfying p, will terminate in a state satisfying q.

We can prove correctness assertions by systematically breaking down the command according to its structure. In particular, we can annotate it with 'verification conditions', and so (automatically) reduce the correctness proof to the problem of verifying some assertions about the underlying mathematical domains.

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. For example, here is 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
```

Floating point correctness (1)

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

$$
\begin{tikzcd}
v(a) \arrow[r, "sin"] & \begin{matrix} sin(v(a)) \\ v(SIN(a)) \end{matrix} \\
a \arrow[u, "v"] \arrow[r, "SIN"'] & SIN(a) \arrow[u, "v"']
\end{tikzcd}
$$

What relationship between $v(SIN(a))$ and $sin(v(a))$ should we require?

Floating point correctness (2)

There are various plausible options, all of which are easy to express formally in HOL Light:

- The answer is the closest representable number to the true answer (with round to even in case of two equally close answers)
- The above is true for all but a small proportion of possible inputs.
- The absolute error is small.
- The relative error is small.
- The error is commensurate with the likely error in the input.

The CORDIC program

```
begin
  var k,x,y,z;
  x := X;
  y := 0;
  k := 1;
  while k < N do
   ( z := srl(n) k x;
     if ult(n) z (neg(n) x) then
        (x := add(n) x z;
         y := add(m) y (logs k));
     k := k + 1
   )
end
```

where `add(n)`, `neg(n)`, `ult(n)` and `srl(n) k` are n-bit addition, 2s complement negation, unsigned comparison ($<$) and right shift by k places, respectively.

The array `logs` contains pre-stored constants.

Without the prettyprinter

This shows what the underlying semantic representation looks like:

```
Assign (\k,(x,(y,z)). k,(X,(y,z))) Seq
Assign (\k,(x,(y,z)). k,(x,(0,z))) Seq
Assign (\k,(x,(y,z)). 1,(x,(y,z))) Seq
While (\k,(x,(y,z)). k < N)
(Assign (\k,(x,(y,z)).
            k,(x,(y,srl n k x))) Seq
 If (\k,(x,(y,z)). ult n z (neg n x))
 (Assign (\k,(x,(y,z)).
             k,(add n x z,(y,z))) Seq
  Assign
     (\k,(x,(y,z)).
        k,(x,(add m y (logs k),z)))) Seq
 Assign (\k,(x,(y,z)). k + 1,(x,(y,z))))
```

However the user need not normally see this form!

The CORDIC program in C

```
int k;
unsigned long x,y,z;
x = X;
y = 0;
k = 1;
while (k < N)
 { z = x >> k;
   if (z < -x)
    { x = x + z;
      y = y + logs[k];
    }
   k = k + 1;
 }
```

(Using unsigned longs in place of the particular word sizes, for the sake of familiarity.)

The CORDIC program in Verilog

```
integer k;
reg [n:0] x,z;
reg [m:0] y;
initial;
begin
  x = X;
  y = 0;
  k = 1;
  while (k < N)
  begin
     z = x >> k;
     if (z < -x)
     begin
       x = x + z;
       y = y + logs[k];
     end
     k = k + 1;
  end
end
```

Annotations for CORDIC program

We can specify intermediate assertions later in the proof by exploiting metavariables. However it is simpler to provide annotations. We assert a loop invariant:

```
{mval(n) x < &1 /\ ...}
```

and that `N - k` decreases with each iteration.

The automatic verification condition generator (working by inference) can calculate all the other intermediate assertions for itself. We are left with four verification conditions:

- The loop invariant is true initially.
- The loop invariant is preserved if the condition in the `if` statement holds.
- The loop invariant is preserved if the condition in the `if` statement does not hold.
- The loop invariant together with $k \geq N$ implies the final postcondition.

Correctness result (1)

The four verification conditions are proved in HOL Light, with the aid of a few lemmas. This proves that the annotated program is correct according to the specification. HOL Light then proves automatically that the program with the annotations removed is still correct. The precondition of the final specification is:

```
inv(&2) <= mval(n) X /\ mval(n) X < &1 /\
&N + &2 <= &n /\ &N <= &2 pow (PRE n) /\
(!i. &0 < &i /\ &i < &N ==>
     &2 pow i * &(logs i) <= &2 pow m /\
     (abs(&(logs i) -
      &2 pow m * ln(&1 + inv(&2 pow i)))
      < &1))
```

i.e. the input value X is in the range $\frac{1}{2} \leq X < 1$, the stored constants are good enough approximations to the true logarithms, and a few conditions on the parameters hold.

Correctness result (2)

and the final postcondition guaranteed by our proof is:

```
abs(mval(m) y + ln(mval(n) X))
   <= &N * (&6 * inv(&2 pow n) +
            inv(&2 pow m)) +
      inv(&2 pow N)
```

That is, the difference between the calculated logarithm `mval(m) y` and (the negation of) the true mathematical result `ln(mval(n) X)` is bounded by $N(6.2^{-n} + 2^{-m}) + 2^{-N}$.

This can be chosen as small as desired by picking the parameters appropriately. Moreover the correct values for the stored table of logarithms can also be calculated in any particular instant, by inference (slowly!)