Formal Verification Methods
4: HOL Light

John Harrison
Intel Corporation

Marktoberdorf 2003

Sat 2nd August 2003 (11:25 – 12:10)

Summary

- HOL and its relatives
- Types, terms and theorems
- Definitional principles
- Inference rules
- Logical connectives
- Mathematics
- Derived rules

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, but the provable theorems are the same as in other versions of HOL.

Versions in CAML Light and Objective CAML.

Used for some formal verification work at Intel.

The HOL family DAG

$$\begin{tikzcd}
 & \text{HOL88} \arrow[dl] \arrow[dd] \arrow[dr] \arrow[drr] & & \\
\text{hol90} \arrow[dr] \arrow[dd] & & \text{ProofPower} & \text{Isabelle/HOL} \\
 & \text{HOL Light} \arrow[dl] & & \\
\text{hol98} \arrow[d] & & & \\
\text{HOL 4} & & &
\end{tikzcd}$$

HOL types

HOL is based on simply typed lambda calculus, with type variables to give simple parametric polymorphism.

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;;
```

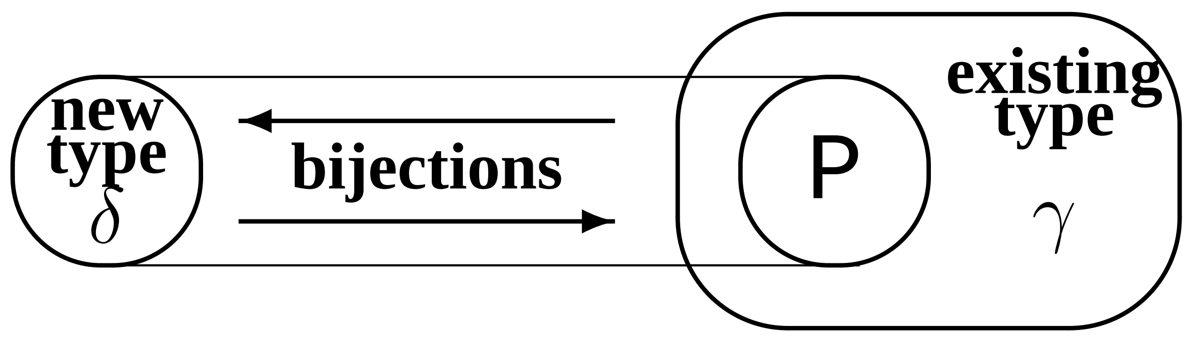
Primitive and defined types

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.



HOL terms

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: $v : ty$, $c : ty$, $f\ x$ and $\lambda x.\ t$.

The abstract type interface ensures that only well-typed terms can be constructed.

Primitive constants

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 $\alpha \to \alpha \to \texttt{bool}$.

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

Later we add the Hilbert $\varepsilon : (\alpha \to \texttt{bool}) \to \alpha$ yielding the Axiom of Choice.

Constant definitions

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 are defined not postulated.

Formulas and theorems

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.

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)

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

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

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

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

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

Simple equality reasoning

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:

```
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;;
```

Logical connectives

Even the logical connectives themselves are defined:

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

These are *not* constructive type theory's Curry-Howard definitions.

Simple Boolean reasoning

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);;
```

Inductive definitions

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, and none of the additional axioms introduced next.

Going classical

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 using a Diaconescu-like proof rather than being postulated separately.

HOL Light mathematics

We 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 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

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

Conclusions

HOL Light is perhaps the purest example of the LCF methodology that is actually useful.

- Minimal logical core
- Almost all concepts defined

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.

Principia Mathematica for the computer age?