HOL Light: an overview

John Harrison

TPHOLs 2009, Munich

18th August 2009, 08:00–09:00

HOL Light overview

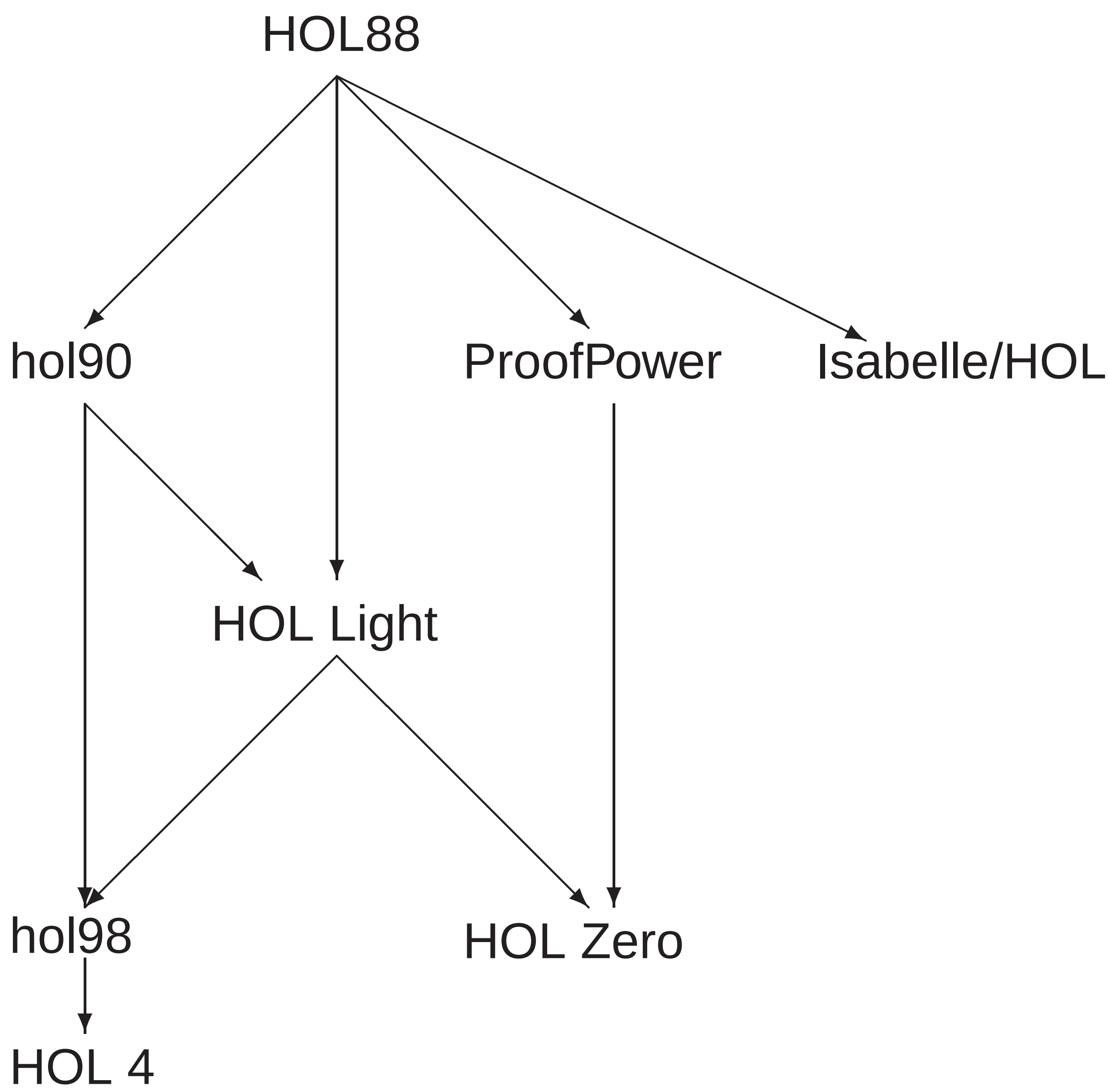
HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

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 and an uncluttered implementation.

Written in Objective CAML (OCaml).

The HOL family DAG



HOL Light's simplicity

HOL Light is a conceptually simple system that puts the user in control.

- The interface is primitive, feels spartan and not user-friendly.
- Users are dropped into a functional language toplevel.

HOL Light’s simplicity

HOL Light is a conceptually simple system that puts the user in control.

- The interface is primitive, feels spartan and not user-friendly.
- Users are dropped into a functional language toplevel.

On the other hand:

- Easy to program, extending the system with new ‘correct by construction’ automation
- Good platform for experimenting with new ideas
  - New proof styles [Harrison 1996]
  - New logical foundations [Voelker 2007]
  - New system architecture [Wiedijk 2009]

HOL Light's applications

Support for typical 'formalize computer science' applications only moderate

- No automated support for coinductive definitions
- No function spaces in recursive types
- Termination prover for recursive functions simple-minded.

Much stronger support (libraries and automation) for

- Formal verification of hardware and software (especially numerical algorithms).
- Mainstream mathematics like analysis and number theory (not so much abstract algebra though)

Some HOL Light theorems

For more see Freek Wiedijk's "Formalizing 100 Theorems" page.

- Jordan Curve Theorem (Tom Hales)
- Radon's theorem (Lars Schewe)
- Prime Number Theorem (John Harrison)
- Univariate Cartan theorems (Marco Maggesi et al.)

Plus many results contributing to the Flyspeck Project.

HOL Light's ASCII syntax

English	Standard	HOL Light
false, true	$\bot$, $\top$	`F, T`
not p	$\neg p$	`~p`
p and q	$p \wedge q$	`p /\ q`
p or q	$p \vee q$	`p \/ q`
p implies q	$p \Rightarrow q$	`p ==> q`
p iff q	$p \Leftrightarrow q$	`p <=> q`
for all x, p	$\forall x.\ p$	`!x. p`
exists x such that p	$\exists x.\ p$	`?x. p`
function $x \mapsto t$	$\lambda x.\ t$	`\x. t`
some x such that p	$\varepsilon x.\ p$	`@x. p`

The LCF approach to theorem proving

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

HOL Light may be the most “extreme” application of this philosophy.

- The primitive rules are very simple and few in number.
- Some large proofs expand to hundreds of millions of primitive inferences.

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;;
```

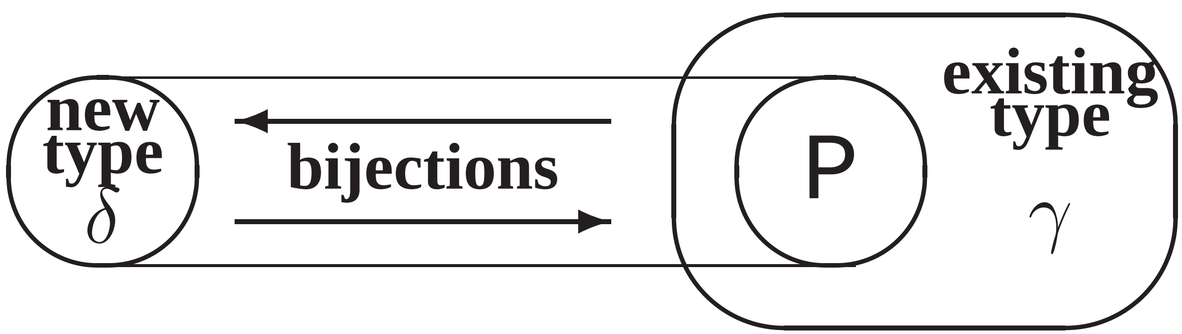
Primitive and defined types

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 $\lambda x.\ t$. Lambda-terms are a notation for functions, e.g. $\lambda x.\ x + 1$ for the successor function.

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. Read $\varepsilon x.\ P(x)$ as 'some x such that $P(x)$'.

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:

$$\top = (\lambda x.\, x) = (\lambda x.\, x)$$

$$\wedge = \lambda p.\, \lambda q.\, (\lambda f.\, f\ p\ q) = (\lambda f.\, f\ \top\ \top)$$

$$\Rightarrow = \lambda p.\, \lambda q.\, p \wedge q = p$$

$$\forall = \lambda P.\, P = \lambda x.\, \top$$

$$\exists = \lambda P.\, \forall Q.\, (\forall x.\, P(x) \Rightarrow Q) \Rightarrow Q$$

$$\vee = \lambda p.\, \lambda q.\, \forall r.\, (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r$$

$$\bot = \forall P.\, P$$

$$\neg = \lambda t.\, t \Rightarrow \bot$$

$$\exists! = \lambda P.\, \exists P \wedge \forall x.\, \forall y.\, P\ x \wedge P\ y \Rightarrow (x = y)$$

Building up derived rules

We proceed to get the full HOL Light system by setting up:

- More and more sophisticated derived inference rules, based on earlier ones.
- New types for mathematical structures, defined in terms of earlier structures.

Thus, the whole system is built in a ‘correct by construction’ way and all proofs ultimately reduce to primitives. An early step in the journey is conjunction introduction

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \wedge q} \; \texttt{CONJ}$$

Definition of CONJ

. . . which is defined as:

```
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) i
    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);;
```

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

A higher-level derived rule

The derived rule `REAL_ARITH` can prove facts of linear arithmetic automatically.

```
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)‘;;
```

But under the surface, everything is happening by primitive inference (about 50000 such inferences).

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 (e.g. the Flyspeck project)
- Quite difficult industrial applications (e.g. FP verification).

For more information:
`http://www.cl.cam.ac.uk/~jrh13/hol-light`