HOL Light from the foundations (part 2/3)

John Harrison

Amazon Web Services

21st Sep 2023 (14:00–14:45)

HOL types

In general, a HOL type is either

- A polymorphic type variable

```
# `:A`;;
val it : hol_type = `:A`
```

HOL types

In general, a HOL type is either

- A polymorphic type variable

```
# `:A`;;
val it : hol_type = `:A`
```

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

HOL types

In general, a HOL type is either

- A polymorphic type variable

```
# `:A`;;
val it : hol_type = `:A`
```

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

- Note that certain basic types like `bool` are considered as nullary type operators.

HOL types

In general, a HOL type is either

- A polymorphic type variable

```
# `:A`;;
val it : hol_type = `:A`
```

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

- Note that certain basic types like `bool` are considered as nullary type operators.

The type system is very closely analogous to that of OCaml itself, and HOL's parser even uses similar algorithms to assign most general polymorphic types.

HOL terms

There are only four basic kinds of HOL term:

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# ‘p:bool‘;;
val it : term = ‘p‘
```

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

```
# `1`;;
val it : term = `1`
```

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

```
# `1`;;
val it : term = `1`
```

- Applications, written with juxtaposition (this is the successor function applied to 0):

```
# `SUC 0`;;
val it : term = `SUC 0`
```

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

```
# `1`;;
val it : term = `1`
```

- Applications, written with juxtaposition (this is the successor function applied to 0):

```
# `SUC 0`;;
val it : term = `SUC 0`
```

- Abstractions or lambdas, written with a backslash

```
# `\x. x + 1`;;
val it : term = `\x. x + 1`
```

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)

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

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

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

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

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

HOL's logical connectives

The usual logical connectives are given ASCII renderings:

$\bot$	`F`	Falsity
$\top$	`T`	Truth
$\neg$	`~`	Not
$\wedge$	`/\`	And
$\vee$	`\/`	Or
$\Rightarrow$	`==>`	Implies ('if ... then ...')
$\Leftrightarrow$	`<=>`	Iff ('... if and only if ...')
$\forall$	`!`	For all
$\exists$	`?`	There exists
$\exists!$	`?!`	There exists a unique

The definitions of the logical connectives

HOL Light is so foundational that even all the basic logical connectives are *defined* in terms of equality:

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

The usual properties of the connectives are *derived* from the primitive rules.

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- `type_of` to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- type_of to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions dest_var, dest_const, dest_comb and dest_abs to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- `type_of` to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions `dest_var`, `dest_const`, `dest_comb` and `dest_abs` to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

- Corresponding constructors `mk_var`, `mk_const`, `mk_comb` and `mk_abs`

```
# mk_var("p",`:bool`);;
val it : term = `p`
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- `type_of` to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions `dest_var`, `dest_const`, `dest_comb` and `dest_abs` to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

- Corresponding constructors `mk_var`, `mk_const`, `mk_comb` and `mk_abs`

```
# mk_var("p",`:bool`);;
val it : term = `p`
```

- `frees` to get the free variables in a term

```
# frees `x + y + 1`;;
val it : term list = [`x`; `y`]
```

Representing more complex terms

All the expressions in logic and mathematics are ultimately expressed using just those four basic terms, and one can explore how it is done using the destructor functions

- Binary logical connectives are just curried functions of the appropriate type:

```
# dest_comb `p /\ q`;;
val it : term * term = (`(/\) p`, `q`)
```

- Quantifiers are higher-order functions applied to an abstraction

```
# dest_comb `!x. x < x + 1`;;
val it : term * term = (`(!)`, `\x. x < x + 1`)
```

Getting help

Note that one can also get help on any predefined HOL Light functions using the `help` function, e.g.

```
# help "mk_abs";;
```

Getting help

Note that one can also get help on any predefined HOL Light functions using the `help` function, e.g.

```
# help "mk_abs";;
```

There is also a full Reference manual with the same information.

Basic and derived definitional principles

Basic principle of constant definition

The only primitive constant for the logic itself is equality `=` with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.

Basic principle of constant definition

The only primitive constant for the logic itself is equality `=` with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.
Later we add the Hilbert $\varepsilon : (\alpha \to \texttt{bool}) \to \alpha$ yielding the Axiom of Choice.

Basic principle of constant definition

The only primitive constant for the logic itself is equality = with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.
Later we add the Hilbert $\varepsilon : (\alpha \to \texttt{bool}) \to \alpha$ yielding the Axiom of Choice.
All other constants are introduced using `new_basic_definition`, the 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

$$\vdash c = t$$

Basic principle of constant definition

The only primitive constant for the logic itself is equality $=$ with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.

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

All other constants are introduced using `new_basic_definition`, the 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

$$\vdash c = t$$

This is an object-level definitional principle, in that c is a constant, not some meta-level abbreviation. It is easy to see that this is conservative, and in particular consistency-preserving.

Basic principle of type definition

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).

Basic principle of type definition

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).

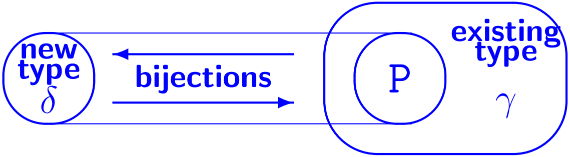
Later we add an infinite type `ind` (individuals) to assert the axiom of infinity.

Basic principle of type definition

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).

Later we add an infinite type `ind` (individuals) to assert the axiom of infinity.

All other types are introduced by `new_basic_type_definition`, the rule of type definition, to be in bijection with any nonempty subset of an existing type.



Again, this is conservative and consistency-preserving.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).

Just using axioms was compared by Russell to theft in place of honest toil.

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Many other theorem provers build such principles in as primitive, and very often get them wrong . . .

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Many other theorem provers build such principles in as primitive, and very often get them wrong . . .

HOL Light supports all these and more using safely *derived* definitional principles.

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

```
# new_inductive_definition `E(0) /\ (!n. E(n) ==> E(n + 2))`;;
val it : thm * thm * thm =
  (|- E 0 /\ (!n. E n ==> E (n + 2)),
   |- !E'. E' 0 /\ (!n. E' n ==> E' (n + 2)) ==> (!a. E a ==> E' a),
   |- !a. E a <=> a = 0 \/ (?n. a = n + 2 /\ E n))
```

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

```
# new_inductive_definition `E(0) /\ (!n. E(n) ==> E(n + 2))`;;
val it : thm * thm * thm =
  (|- E 0 /\ (!n. E n ==> E (n + 2)),
   |- !E'. E' 0 /\ (!n. E' n ==> E' (n + 2)) ==> (!a. E a ==> E' a),
   |- !a. E a <=> a = 0 \/ (?n. a = n + 2 /\ E n))
```

The function returns a triple of theorems:

- A 'rule' theorem (the inductively defined predicate is closed under the rules)
- An 'induction' or minimality theorem (the inductively defined predicate is the least such)
- A 'cases' theorem that each element arises by virtue of one of the rules.

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

The rule returns a pair of theorem, one justifying 'structural induction' over the type:

```
val btree_INDUCT : thm =
 |- !P. (!a. P (Leaf a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (Branch a0 a1))
        ==> (!x. P x)
```

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

The rule returns a pair of theorem, one justifying 'structural induction' over the type:

```
val btree_INDUCT : thm =
 |- !P. (!a. P (Leaf a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (Branch a0 a1))
        ==> (!x. P x)
```

and the other justifying definition by primitive recursion

```
val btree_RECURSION : thm =
|- !f0 f1.
        ?fn. (!a. fn (Leaf a) = f0 a) /\
             (!a0 a1. fn (Branch a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

```
let fib = define
   `fib 0 = 1 /\
    fib 1 = 1 /\
    fib (n + 2) = fib(n) + fib(n + 1)`;;
val fib : thm =
|- fib 0 = 1 /\ fib 1 = 1 /\ fib (n + 2) = fib n + fib (n + 1)
```

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

```
let fib = define
   `fib 0 = 1 /\
    fib 1 = 1 /\
    fib (n + 2) = fib(n) + fib(n + 1)`;;
val fib : thm =
|- fib 0 = 1 /\ fib 1 = 1 /\ fib (n + 2) = fib n + fib (n + 1)
```

Some tail-recursive cases can be justified even without an ordering:

```
define `collatz(n) = if n <= 1 then n
                     else if EVEN(n) then collatz(n DIV 2)
                     else collatz(3 * n + 1)`;;
```