HOL Foundations

Arthur Grundner

8. Juni 2018

Table of Contents

1 Description

HOL is a family of proof assistants that use a similar variant of a higher-order logic (a form of classic set theory typed using simple types with rank-1 polymorphism) and is implemented by using a small inference kernel (the LCF approach). The design goal was to use a logic that is simple but provides enough expressiveness. Starting from LCF, we introduce HOL's logic and compare it with first-order set theory.

2 The evolution of LCF

2.1 Stanford LCF

The foundations of HOL proof assistants go back to 1969, when Dana Scott devised a formalism that constitutes the groundwork for the development of a proof-checking program called LCF („Logic for Computable Functions“). The intention of the formalism was to be able to reason about recursively defined functions, which were used in denotational semantics [HUW]. Scott was one of the founding members of denotational semantics, which had the goal, in a rough sense, to find mathematical objects called domains, that represent the behavior of computer programs. Scott's formalism uses typed λ-terms (thus all terms are either variables, constants, λ-abstractions or λ-applications), predicate calculus for formulae and Scott Domains (a type of partially ordered sets) for types [Gor00].

While the formalism was not published until 1993, Robin Milner developed LCF with the help of Diffie, Weyhrauch and Newey at Stanford University in 1972. The original LCF system is nowadays known as Stanford LCF. Its features comprised a powerful automatic simplification mechanism and the support for backwards, goal-directed proof [HUW]. Milner put it as follows [Gor00]:

„The proof-checking program is designed to allow the user interactively to generate formal proofs about computable functions and functionals over a variety of domains, including those of interest to the computer scientist - for example, integers, lists and computer programs and their semantics. The user's task is alleviated by two features: a subgoaling facility and a powerful simplification mechanism.“

The proofs worked as follows: First you declare a main-goal, which is then split up and simplified into sub-goals which in turn are again split up and

simplified until all goals are proven, by being reduced to axioms or theorems (this is done with the help of *tactics*, cf. Chapter 2.2.1)
Stanford LCF had two major issues though; the storage of proofs filled up the then not ample available memory very quickly and the repertoire of proof commands was immutable [HUW].

2.2 Edinburgh LCF

After moving to Edinburgh, Milner addressed these problems in his second version called Edinburgh LCF. To deal with the fast fill-up of storage, the system would just remember the results of proofs and not the proofs themselves, just as a lecturer erasing previous parts of a long proof while proving a theorem. For full customizability, Milner and his team developed the strictly typed programming language ML ('Meta Language') [HUW]. ML was conceived as a functional language, so that so-called 'tactics' and 'tacticals' could be programmed as functions.

2.2.1 Tactics

A tactic is a function, which has a goal as its input and returns a list of sub-goals along with a justification function, that maps a list of theorems to a theorem. So a tactic is an ML function, that takes a goal, that we want to prove and refines it into (often multiple) subgoals, giving a justification for the refinement. HOL provides a collection of pre-defined tactics and tacticals. In general we write a tactic with the following notation [GM93, p.366]:

$$\frac{goal}{goal_1\ goal_2\ \ldots\ goal_n}$$

Examples are:

- CONJ_TAC $\dfrac{t_1 \wedge t_2}{t_1\ t_2}$
- INDUCT_TAC (mathematical induction) $\dfrac{\forall n.t[n]}{t[0]\quad \{t[n]\}t[SUC\ n]}$,

  where the induction hypothesis is written inside curly brackets.

2.2.2 Tacticals

Tacticals on the other hand are functions that can compose tactics. As an example take the tactical THEN and let T_1 and T_2 be tactics. Then the ML expression T_1 THEN T_2 is again a tactic, that applies T_1 to some goal and then T_2 to all sub-goals produced by T_1.

Some strategies or tactics may fail, therefore an exception handling mechanism was included in ML. Additionally ML features a novel polymorphic type system: In Church's original λ-calculus, a term with type variables (variables that are placeholders for types), i.e. a polymorphic term, is a meta-level-notion, denoting a family of terms, whereas in LCF it is a *single* polymorphic term [Gor96]. For example, a function f, that can operate on the real numbers and on bool, would have to be declared twice in Church's λ-calculus, in LCF we can define $f : \alpha \to \alpha$, where α can be substituted for the bool or the real number type.
To prohibit the accidental production of wrong theorems, Milner created an abstract data type with instances of axioms as predefined values and inference rules as operations. This ensured that every object of type theorem could only be inferred by given inference rules applied to axioms or other theorems. Thus, we have a guarantee that all theorems have been correctly deduced simply because of their type and only the consistency of the axioms has to be checked or assumed.

2.2.3 Illustration of the LCF approach

Let **thm** denote the ML type of theorems. Logical inference rules are then implemented as functions, which return an object of type **thm**.
Taking Modus Ponens as an example:

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \Rightarrow q}$$

In this case we have a function, say MP, which takes the two theorems $\Gamma \vdash p \Rightarrow q$ (which means $p \Rightarrow q$ can be proven from Γ) and $\Delta \vdash p$ as input and returns the theorem $\Gamma \cup \Delta \Rightarrow q$. In Standard ML we would write:

$$\textbf{val MP : thm} \to \textbf{thm} \to \textbf{thm}$$
$$\textbf{MP}(\Gamma \vdash a \Rightarrow b)(\Delta \vdash a) = (\Gamma \cup \Delta \vdash b)$$

The first line is the result of currying the (equivalent) type of operation $(\mathbf{thm}, \mathbf{thm}) \rightarrow \mathbf{thm}$ [Tue17].

2.3 Cambridge LCF

Following the porting of the Edinburgh LCF code to the two Lisp dialects Le Lisp and MacLisp by Gerard Huet, Larry Paulson improved much of the Lisp code. Techniques of conversion and theorem continuations were added and discrimination nets implemented. The LCF logic was extended to also include disjunction and existential quantification. The resulting system was called Cambridge LCF, due to the workplace of Paulson, and ported to Standard ML. Cambridge LCF came with full predicate logic, a comprehensive set of tactics and tacticals (including the chaining and resolution tactics). Conversions, theorem continuations, subgoaling commands and many other features were introduced, which are still part of HOL [oC18].

3 HOL

While Paulson was working on Cambridge LCF, Mike Gordon, also working at Cambridge from 1981 onwards [Wik18c], was interested in the formal verification of hardware. Inspired by a theoretical result (Expansion Theorem) by Milner concerning the behavior of a digital system with respect to its constituents, he invented a notation called LSM ('Logic of Sequential Machines'). He combined LSM with a version of Cambridge LCF, which resulted in LCF_LSM embroidered with some additional features. With the help of Ben Moskowski, terms of LSM were encoded in predicate calculus was ultimately the birth of HOL. The predicate calculus formulae and the typed λ-calculus terms were retained, but types were reinterpreted as ordinary sets instead of Scott Domains, since the latter was not necessary for hardware verification. Syntactically HOL is using types. The elements of these types are interpreted as sets or elements of sets. So the main difference between HOL and Cambridge LCF constitutes HOL's use of higher-order logic with types interpreted as sets instead of Domain Theory.
Hardware was modelled as relations between input and output signals, whereas the signals were represented by functions, that map time to values. Thus higher-order relations and quantifications became necessary, which in turn lead Gordon to use higher-order logic. Some care was put into supporting

maximal upward compatibility of LCF code.

3.1 The HOL family

After HOL, many versions followed and most of them are still being used and maintained today [HUW].

- **HOL88**

  The core version of HOL. It got ported to Common Lisp.

- **hol90**

  The development of hol90 started with reimplementing HOL in Standard ML and resulted amongst other things in a significant performance boost with regards to HOL88.

- **ProofPower**

  A commercial version of HOL, developed at ICL. It was devised mainly for security applications.

- **HOL Light**

  Started as an experimental minimalist version of HOL88, in which the entire HOL88 design was redone.

- **hol98**

  Has applications both to software and hardware verification. hol98 is in public domain and freely usable [COR03].

- **HOLZero**

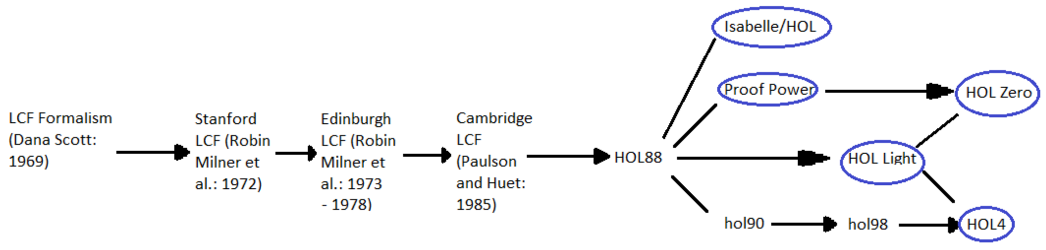
  A minimalist version of HOL.

- **HOL4**

  Includes a large library of theorem proving code. It is still implemented in Standard ML.

- **Isabelle/HOL**

  Technically, Isabelle is not part of the HOL family, only a descendant of it. It was the result of trying to provide a systematic way of implementing 'LCF-style' systems.


LCF Formalism (Dana Scott: 1969)
Stanford LCF (Robin Milner et al.: 1972)
Edinburgh LCF (Robin Milner et al.: 1973 - 1978)
Cambridge LCF (Paulson and Huet: 1985)
HOL88
Isabelle/HOL
Proof Power
HOL Zero
HOL Light
hol90
hol98
HOL4

From LCF to HOL. In blue circles the systems which are still maintained and developed are highlighted (based on [HUW]).

3.2 HOL's logic and novelties

HOL uses higher-order logic, that is an extended form of first-order (predicate) logic. In first-order logic we can only quantify over individuals, like

$$\forall x \in \mathbb{N} : (x + 0 = x).$$

In second-order logic, we can also quantify over sets, functions and predicates themselves as in

$$\forall Q \exists P \exists f \exists x \exists y \; P(f(x)) \rightarrow Q(y)$$

or in the principle of mathematical induction [dh]. Higher-order logic in general admits quantification over sets or predicates, that are nested arbitrarily deep (when working specifically with n-th order logic, the nesting can only be n layers deep) [Wik18b].
Notably, with the advent of HOL, some syntactic simplifications were introduced, as supporting conventional notation as $\forall x.P[x]$ (which is then mapped down to $\forall(\lambda x P[x])$; cf. *'binders'*). Also a mechanism for defining new types was included. The development of theories based on conservative extension became the norm; that is if a statement, that doesn't include the new concept, wasn't provable before the extension, it will remain unprovable.
The logical basis of the HOL logic contains two important theories: **bool** and **ind**. All other important theories can be defined in terms of those two theories. A theory is a collection of types, type operators, constants, definitions, axioms and theorems [Gor01].

3.2.1 The theory bool

The theory bool contains the type *bool*, four axioms for higher-order logic and three primitive constants (equality $=$, implication $\Rightarrow$, choice ε). Equality and implication are well-known, choice on the other hand is somewhat more intricate.

3.2.2 The Choice- or Hilbert's ε-operator

Let $t[x]$ be a term of type $\sigma \to bool$ with a free variable x, then $\varepsilon x.t[x]$ denotes some member a of the set σ, for which the expression $t[a]$ yields *true*. If there is no such element, then $\varepsilon x.t[x]$ represents an arbitrary element of the set σ. The constant ε is a version of Hilbert's ε-operator. For example $\varepsilon n.n < 5$ represents some unspecified number below 5, $\varepsilon n.(n^2 = 25) \wedge (n \geq 0)$ denotes 5, whereas $\varepsilon n.\neg(n = n)$ denotes an unspecified number. The ε-operator is quite powerful, one can for example define the if-then-else operator or simulate λ-abstraction with it. As an example, the Peano-Lawvere axiom asserts that, with a given tuple (n_0, f), $n_0 \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$, there exists a unique sequence s such that

$$(s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n))).$$

With the ε-operator we can define a function g, that has some tuple (n_0, f) of the beforementioned form as its input and returns the unique sequence s:

$$\mathbf{Rec}(n_0, f) = \varepsilon s.\ (s(0) = n_0) \wedge (\forall n.\ s(n+1) = f(s(n))).$$

From the axiom it follows that the first term is valid after replacing every occurrence of s by $\mathbf{Rec}(n_0, f)$.
An important thing to add is, that with the ε-operator, we have implicitly built in the Axiom of Choice [Gor01].

With just these three constants, we can define the constants $\top$ (truth), $\bot$ (falsity), $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\forall$ (universal quantification), $\exists$ (existential quantification) and $\exists!$ (unique existence quantification) [GM93]. As an example we define existential quantification (the \$ can be skipped, it only means that syntactically the succeeding symbol should not be read as a binder):

$$\vdash \$\exists = \lambda P.\ P(\$\varepsilon\ P)$$

Definitions are always of the form $\vdash c = t$, where c is a new constant and t is a term without any free variables, not already containing c. The term $\$\varepsilon P$ returns some element x of a specific type α, for which $P(x)$ yields 'true', given that it exists. Thus $P(\$\varepsilon\ P)$ is true iff there exists an element x of the specified type, that fulfills the predicate P. In light of the definition, the constant $\exists$ is to be read as a function $(\alpha \to \mathit{bool}) \to \mathit{bool}$, where α is a type variable (thus making $\exists$ a polymorphic constant) [Gor01]. Due to the definitions, we can read the existential and universal quantifier just as in predicate calculus.
There are also some more useful but less important constants included in bool and also some definitions concerning amongst other things the injectivity and surjectivity of functions.
The theory bool contains four axioms [Gor01]:

$$\begin{gathered}\vdash \forall b.\ (b = \top) \vee (b = \bot)\\ \vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)\\ \vdash \forall f.\ (\lambda x.\ f\ x) = f\\ \vdash \forall P\ x.\ P\ x \Rightarrow P(\$\varepsilon\ P).\end{gathered}$$

The forth axiom for example means that, for all predicates P and all variables x holds: $P(x) \Rightarrow P(\$\varepsilon\ P)$. That is, if $P(x)$ holds for some x, then there exists an x such that $P(x)$ holds.

3.2.3 The theory ind

The theory ind introduces a new primitive type *ind* (for 'individuals') and the Axiom of Infinity as the fifth and final axiom of higher-order logic [GM93]:

$$\vdash \exists f : \mathit{ind} \to \mathit{ind}.\ (\mathbf{One\_One}\ f) \wedge \neg(\mathbf{Onto}\ f)$$

It states the existence of an injective and not surjective function on the type of *ind*. If *ind* would denote a finite set, then injectivity of the function f would imply its surjectivity, thus rendering the axiom invalid. So the Axiom of Infinity asserts that *ind* denotes an infinite set, a construction that would be impossible using only the theory bool with the inference rules from the next subsection. The inference rules together with the four axioms of the theory bool and the Axiom of Infinity are together sufficient for devoloping all standard mathematics [GM93].

3.2.4 Inference Rules

There are eight inference rules in HOL [Gor01]. Let $t, t_1, t_2, \ldots$ be arbitrary terms.

1.) ASSUME (Assumption introduction)

$$\frac{}{t \vdash t}$$

2.) REFL (Reflexivity)

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

3.) BETA_CONV (Beta-conversion)

$$\frac{}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$$

with the restriction, that no free variables of t_2 become bound after the substitution for x.

4.) SUBST (Substitution)

$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad \Gamma_2 \vdash t[t_1]}{\Gamma_1 \cup \Gamma_2 \vdash t[t_2]}$$

By $t[t_1]$ we denote, that there are some free occurrences of t_1 in t, which are then replaced by t_2 in $t[t_2]$. As in the case of Beta-Conversion, no free variables of t_2 shall be bound after substitution.

5.) ABS (Abstraction)

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$

when x is not a free variable in Γ.

6.) INST_TYPE (Type Instantiation)

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

when none of $\alpha_1, \ldots, \alpha_n$ occur in Γ.

7.) DISCH (Discharging an assumption)

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}$$

8.) MP (Modus Ponens)

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

As an application of the inference rules, we prove the theorem ADD_ASSUM (Adding an assumption):

$$\frac{\Gamma \vdash t}{\Gamma, t' \vdash t}$$

Proof. 1. $t' \vdash t'$ [ASSUME]
2. $\Gamma \vdash t$ [hypothesis]
3. $\Gamma \vdash t' \Rightarrow t$ [DISCH on Line 2]
4. $\Gamma, t' \vdash t$ [MP on Lines 3, 1] □

4 Comparison of HOL with First-Order Set Theory

As was mentioned in Chapter 3, HOL is fundamentally based on typed higher-order logic, more generally on type theory. In type theory functions are taken as the most basic operators/primitives, as in matter of fact in simply typed lambda calculus the function operator is the *only* type operator [Wik18d]. Due to the pervasiveness of functions especially in Computer Science it seems natural to take functions as primitives [Gor96]. For example, the natural numbers can be defined in type theory as an inductive type $\mathbb{N}$, having two constructors [hc]:

$$1\colon \mathbb{N}$$
$$S\colon \mathbb{N} \to \mathbb{N}$$

On the other hand, in set theory they are defined as nested sets of the empty set:

$$\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}, \ldots\}$$

In type theory, tools for indexing terms, structuring data and checking types are easily provided. Furthermore proofs and general laws are often shorter and simpler in typed higher-order logic [Gor96]. As an example the Axiom of Replacement is an axiom schema in first-order type theory, where for each formula ϕ with output of either true or false an instance of the axiom has to be included; thus making it an axiom schema and less preferable to the type-theoretic formulation.
Axiom of Replacement in set theory [Wik18a]:

$$\begin{aligned}&\forall \omega_1, \ldots, \omega_n \, \forall A([\forall x \in A \exists! y \; \phi(x, y, \omega_1, \ldots, \omega_n, A)] \\ &\qquad\qquad \Rightarrow \exists B \forall y [y \in B \Leftrightarrow \exists x \in A \; \phi(x, y, \omega_1, \ldots, \omega_n, A)])\end{aligned}$$

Axiom of Replacement in type theory [Gor96]:

$$\forall \; f \; s. \, \exists t. \, \forall y. \; y \in t \;=\; ((\exists x. \; x \in s) \wedge (y = f(x)))$$

In short, the Axiom of Replacement asserts that the image of a set under some mapping is again a set. The set-theoretic formulation can surely be written more compactly; the point is to show the concise formulation of an axiom of ZFC (Zermelo-Fraenkel set theory) in type theory.

It is not difficult to build set theory on top of type theory. For this, we can declare a type V and a constant $\in$: $V \times V \to$ *bool*, which has the interpretation of the element symbol of set theory. With the notion of 'being an element of' we can postulate the axioms of set theory (ZFC) [Gor96].

On the other hand there are reasons in favor of set theory. First of all there is no standard formulation for typed higher-order logic. There are many different versions, which differ widely in notation and underlying mathematical notions of truth (e.g. constructive vs. classical logic). In set theory there is very little variation. Most mathematicians use ZFC as a foundation for mathematics. Furthermore there are areas of science, where types do not fit as well and seem unmotivated. Type theory is also relatively unknown to most mathematicians [Gor96].
As a final, remarkable difference we note that in type theory there is less variation of language possible; if A is a set, then a subset of A can't be a set as well, but will be completely different entity. Or if n is a natural number, it is no integer at the same time. Elements must usually belong to only one type, which doesn't correspond to the basic notions in mathematics [hb]. Nonetheless, with this viewpoint Russell's paradox can be resolved.

References

[COR03] CORDIS. Proof and specification assisted design environment, prosper; hol98 improvements, January 2003.

[dh] dezakin (https://math.stackexchange.com/users/89487/dezakin). What are some examples of third, fourth, or fifth order logic sentences? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1052118 (version: 2014-12-04).

[GM93] Mike JC Gordon and Tom F Melham. Introduction to hol, 1993.

[Gor96] Mike Gordon. Set theory, higher order logic or both? In *International Conference on Theorem Proving in Higher Order Logics*, pages 191–201. Springer, 1996.

[Gor00] Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–186, 2000.

[Gor01] Mike Gordon. Hol - a machine oriented formulation of higher order logic, 2001.

[hb] Luca Bressan (https://math.stackexchange.com/users/89879/luca bressan). Why is it worth spending time on type theory? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/569543 (version: 2013-11-16).

[hc] John Colanduoni (https://math.stackexchange.com/users/60879/john colanduoni). Constructing the natural numbers without set theory. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1290605 (version: 2015-05-20).

[HUW] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving.

[oC18] University of Cambridge. Publications on lcf and hol, April 2018.

[Tue17] Thomas Tuerk. Interactive theorem proving (itp) course web version, May 2017.

[Wik18a] Wikipedia contributors. Axiom schema of replacement — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Axiom_schema_of_replacement&oldid=826369933`, 2018. [Online; accessed 7-June-2018].

[Wik18b] Wikipedia contributors. Higher-order logic — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Higher-order_logic&oldid=832883216`, 2018. [Online; accessed 7-June-2018].

[Wik18c] Wikipedia contributors. Michael j. c. gordon — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Michael_J._C._Gordon&oldid=827846646`, 2018. [Online; accessed 7-June-2018].

[Wik18d] Wikipedia contributors. Simply typed lambda calculus — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Simply_typed_lambda_calculus&oldid=843199501`, 2018. [Online; accessed 7-June-2018].