Towards Verified Systems

$\boxed{\text{OVERVIEW}}$

Jonathan Bowen (Ed.)

June 4, 1993

$\boxed{\text{DRAFT OVERVIEW OF HOL BY MJCG \& AP}}$

Contents

Chapter 1

The HOL logic and system

Higher order logic is a version of predicate calculus that allows quantified variables to range over functions and predicates. The power of this logic is similar to set theory. It is sufficient for expressing most ordinary mathematical theories. The HOL system is a theorem-proving environment for higher order logic. It provides tools for proving theorems directly in the logic as well as system building facilities that enable users to implement (in a guaranteed secure fashion) their own application specific proof environments. HOL is not a generic theorem prover like Isabelle [15]; it is 'hardwired' to higher order logic. However, this logic is powerful and other formalisms can be represented inside it via their semantics. Many applications of HOL are such semantic embeddings; these are implemented with tools like parser and pretty-printer generators that are provided by HOL. Examples include support for the specification language **Z**, subsets of the hardware description languages Ella, VHDL and Silage, the programming logics TLA and UNITY, several refinement calculi and the process algebras CSP, CCS and the π-calculus.

The result of a session with the HOL system is a *theory*. This consists of types, constants, definitions, axioms and an explicit list of theorems that have been proved from the axioms and definitions. The HOL system provides tools for extending and combining theories. A typical interaction with HOL consists in combining some existing theories, making some definitions, proving some theorems and then saving the resulting new theories. The HOL system ensures that only well-formed theories can be constructed by only allowing theorems to be created by *formal proof*. All the theorems of such theories are logical consequences of the definitions and axioms of the theory.

In this chapter, which is condensed from the book *Introduction to HOL* [7], the version of higher order logic supported by HOL – the HOL logic – is described in detail, but the theorem proving infrastructure is only outlined (see 1.2.2).

1.1. The HOL logic

The HOL syntax contains syntactic categories of types and terms whose elements are intended to denote, respectively, sets and elements of sets. The formal semantics of the HOL logic is only sketched here (see 1.1.5).

The HOL logic is typed: each theory specifies a signature of type constants and term constants; these then determine sets of types and terms.

1.1.1. Types

The types of the HOL logic denote sets. Following tradition, σ, possibly decorated with subscripts or primes, is used to range over arbitrary types.

There are four kinds of types in the HOL logic. These can be described informally by the following BNF grammar, in which α ranges over type variables, c ranges over atomic types and op ranges over type operators.

$$\sigma \quad ::= \quad \underset{\substack{\uparrow \\ \text{type variables}}}{\alpha} \quad \mid \quad \underset{\substack{\uparrow \\ \text{atomic types}}}{c} \quad \mid \quad \underbrace{(\sigma_1,\ldots,\sigma_n)op}_{\substack{\uparrow \\ \text{compound types}}} \quad \mid \quad \underbrace{\sigma_1 \to \sigma_2}_{\substack{\uparrow \\ \text{function types} \\ \text{(domain } \sigma_1\text{, range } \sigma_2\text{)}}}$$

In more detail, the four kinds of types are as follows:

1. **Type variables:** these range over arbitrary sets.

2. **Atomic types:** these denote fixed sets.

3. **Compound types:** these are expressions $(\sigma_1,\ldots,\sigma_n)op$, where σ_1, $\ldots$,σ_n are the argument types and op is a *type operator* of arity n. Type operators denote operations for constructing sets. The type $(\sigma_1,\ldots,\sigma_n)op$ denotes the set resulting from applying the operation denoted by op to the sets denoted by σ_1, $\ldots$, σ_n.

4. **Function types:** If σ_1 and σ_2 are types, then $\sigma_1 \to \sigma_2$ is the function type with *domain* σ_1 and *range* σ_2. It denotes the set of all (total) functions from the set denoted by its domain to the set denoted by its range.

It turns out to be convenient to identify atomic types with compound types constructed with 0-ary type operators. For example, the atomic type *bool* of truth-values can be regarded as being an abbreviation for $()bool$. This identification will be made in the technical details that follow, but in the informal presentation atomic types will continue to be distinguished from compound types, and $()c$ will still be written as c.

Type structures

The term 'type constant' is used to cover both atomic types and type operators. It is assumed that an infinite set TyNames of the *names of type constants* is given. The greek letter ν is used to range over arbitrary members of TyNames, c will continue to be used to range over the names of atomic types (i.e. 0-ary type constants), and op is used to range over the names of type operators (i.e. n-ary type constants, where $n > 0$).

It is assumed that an infinite set TyVars of *type variables* is given. Greek letters $\alpha, \beta, \ldots$, possibly with subscripts or primes, are used to range over Tyvars. The sets TyNames and TyVars are assumed disjoint.

A *type structure* is a set Ω of type constants. A *type constant* is a pair (ν, n) where $\nu \in$ TyNames is the name of the constant and n is its arity. Thus $\Omega \subseteq \mathsf{TyNames} \times \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers). It is assumed that no two distinct type constants have the same name, i.e. whenever $(\nu, n_1) \in \Omega$ and $(\nu, n_2) \in \Omega$, then $n_1 = n_2$.

The set Types_Ω of types over a structure Ω can now be defined as the smallest set such that:

- $\mathsf{TyVars} \subseteq \mathsf{Types}_\Omega$.
- If $(\nu, 0) \in \Omega$ then $()\nu \in \mathsf{Types}_\Omega$.
- If $(\nu, n) \in \Omega$ and $\sigma_i \in \mathsf{Types}_\Omega$ for $1 \leq i \leq n$, then $(\sigma_1, \ \ldots\ , \sigma_n)\nu \in \mathsf{Types}_\Omega$.
- If $\sigma_1 \in \mathsf{Types}_\Omega$ and $\sigma_2 \in \mathsf{Types}_\Omega$ then $\sigma_1 \rightarrow \sigma_2 \in \mathsf{Types}_\Omega$.

The operator $\rightarrow$ is assumed to associate to the right, so that $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$ abbreviates $\sigma_1 \rightarrow (\sigma_2 \rightarrow \ldots \rightarrow (\sigma_n \rightarrow \sigma) \ldots)$. The notation $tyvars(\sigma)$ is used to denote the set of type variables occurring in σ.

1.1.2. Terms

The terms of the HOL logic are expressions that denote elements of the sets denoted by types. The meta-variable t is used to range over arbitrary terms, possibly decorated with subscripts or primes.

There are four kinds of terms in the HOL logic. These can be described approximately by the following BNF grammar, in which x ranges over variables and c ranges over constants.

$$t \quad ::= \quad \underset{\underset{\text{variables}}{\uparrow}}{x} \quad | \quad \underset{\underset{\text{constants}}{\uparrow}}{\mathsf{c}} \quad | \quad \underbrace{t\ t'}_{\underset{\substack{\text{function applications}\\ \text{(function } t\text{, argument } t'\text{)}}}{\uparrow}} \quad | \quad \underbrace{\lambda\, x.\ t}_{\underset{\lambda\text{-abstractions}}{\uparrow}}$$

A λ-term $\lambda\, x.\ t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in t. An application $t\ t'$ denotes the result of applying the function denoted by t to the value denoted by t'.

The BNF grammar just given omits mention of types. In fact, each term in the HOL logic is associated with a unique type. The notation t_σ is traditionally used to range over terms of type σ. A more accurate grammar of terms is the following:

$$t_\sigma \quad ::= \quad x_\sigma \quad | \quad \mathsf{c}_\sigma \quad | \quad (t_{\sigma' \rightarrow \sigma}\ t'_{\sigma'})_\sigma \quad | \quad (\lambda\, x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \rightarrow \sigma_2}$$

Just as the definition of types was relative to a particular type structure Ω, the definition of terms is relative to a given collection of typed constants over Ω. Assume that an infinite set Names of names is given. A *constant* over Ω is a pair (c, σ), where $\mathsf{c} \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$. A *signature* over Ω is just a set Σ_Ω of such constants.

The set $\mathsf{Terms}_{\Sigma_\Omega}$ of terms over Σ_Ω is defined to be the smallest set closed under the following rules of formation:

1. **Constants:**
   If $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ and $\sigma' \in \mathsf{Types}_\Omega$ is obtained from σ by substituting types for type variables, then $(\mathsf{c}, \sigma') \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *constants* and are written $\mathsf{c}_{\sigma'}$.

2. **Variables:**
   If $x \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$, then $\texttt{var}\ x_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *variables*. The marker `var` is purely a device to distinguish variables from

constants with the same name. A variable $\texttt{var}\ x_\sigma$ will usually be written as x_σ, if it is clear from the context that x is a variable rather than a constant.

3. **Function applications:**
   If $t_{\sigma' \to \sigma} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t'_{\sigma'} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$.

4. **λ-Abstractions:**
   If $\texttt{var}\ x_{\sigma_1} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t_{\sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(\lambda\, x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \to \sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$.

Note that it is possible for constants and variables to have the same name. It is also possible for different variables to have the same name, if they have different types.

The type subscript on a term may be omitted if it is clear from the structure of the term or the context in which it occurs what its type must be.

Function application is assumed to associate to the left, so that $t\ t_1\ t_2\ \ldots\ t_n$ abbreviates $(\ \ldots\ ((t\ t_1)\ t_2)\ \ldots\ t_n)$.

The notation $\lambda\, x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda\, x_1.\ (\lambda\, x_2.\ \cdots\ (\lambda\, x_n.\ t)\ \cdots\ )$.

A term is called polymorphic if it contains a type variable. Otherwise it is called monomorphic. Note that a term t_σ may be polymorphic even though σ is monomorphic—for example, $(f_{\alpha \to b}\ x_\alpha)_b$, where b is an atomic type. The expression $\mathit{tyvars}(t_\sigma)$ denotes the set of type variables occurring in t_σ.

An occurrence of a variable x_σ is called *bound* if it occurs within the scope of a textually enclosing $\lambda\, x_\sigma$, otherwise the occurrence is called *free*. Note that $\lambda\, x_\sigma$ does not bind $x_{\sigma'}$ if $\sigma \neq \sigma'$. A term in which all occurrences of variables are bound is called *closed*.

1.1.3. Standard notions

Up to now the syntax of types and terms has been very general. To represent the standard formulae of logic it is necessary to impose some specific structure. In particular, every type structure must contain an atomic type *bool* which is intended to denote the distinguished two-element set of truth-values. Logical formulae are then identified with terms of type *bool*. In addition, various logical constants are assumed to be in all signatures. These requirements are formalized by defining the notion of a standard signature.

Standard type structures

A type structure Ω is *standard* if it contains the atomic types *bool* (of booleans or truth-values) and *ind* (of individuals). In the literature, the symbol o is often used instead of *bool* and ι instead of *ind*. It will be assumed from now on that type structures are standard.

Standard signatures

A signature Σ_Ω is *standard* if it contains $\Rightarrow_{\mathit{bool} \to \mathit{bool} \to \mathit{bool}}$, $=_{\alpha \to \alpha \to \mathit{bool}}$ and $\varepsilon_{(\alpha \to \mathit{bool}) \to \alpha}$. The first of these is intended to denote logical implication and the second is intended to denote equality. The third is Hilbert's epsilon operator and builds the Axiom of Choice into the HOL logic. A term $\varepsilon_{(\alpha \to \mathit{bool}) \to \alpha}\ p_{\alpha \to \mathit{bool}}$ denotes some value for which the predicate $p_{\alpha \to \mathit{bool}}$ is true (if no such value exists, then an arbitrary value of type α is chosen). See the definition of the conditional Cond in 1.1.8 for an example of the use of ε.

Remark This particular choice of primitive constants is arbitrary. The standard collection of logical constants includes T ('true'), F ('false'), $\Rightarrow$ ('implies'), $\neg$ ('not'), $\wedge$ ('and'), $\vee$ ('or'), $\forall$ ('for all'), $\exists$ ('there exists'), $=$ ('equals'), and ε ('a'). This set is redundant, since it can be defined (in a sense explained in 1.1.10) from various subsets. In practice, it is necessary to work with the full set of logical constants, and the particular subset taken as primitive is not important. The interested reader can explore this topic further by reading Andrews' book [1] and the references it contains.

Terms of type *bool* are called *formulae*. The following notational abbreviations are used:

Notation	Meaning
$t_\sigma = t'_\sigma$	$=_{\sigma\to\sigma\to bool}\ t_\sigma\ t'_\sigma$
$t \Rightarrow t'$	$\Rightarrow_{bool\to bool\to bool}\ t_{bool}\ t'_{bool}$
$\varepsilon x_\sigma.\ t$	$\varepsilon_{(\sigma\to bool)\to\sigma}(\lambda x_\sigma.\ t)$

These notations are special cases of general abbreviatory conventions supported by the HOL system. The first two are infixes and the third is a binder.

1.1.4. Sequents

The HOL logic is based on *sequents*. Fixing a (standard) signature Σ_Ω, a sequent is a pair (Γ, t) where Γ is a finite set of formulae over Σ_Ω and t is a single formula over Σ_Ω.[1] The set of formulae Γ forming the first component of a sequent is called its set of *assumptions* and the term t forming the second component is called its *conclusion*. When it is not ambiguous to do so, a sequent $(\{\}, t)$ is written as just t.

1.1.5. Semantics

Part III of *Introduction to HOL* [7] contains a set-theoretic semantics of the HOL logic due to Andy Pitts. This is only briefly outlined here (the exposition uses material from a paper by Tom Melham [10]).

The semantics of HOL is defined in terms of a particular set $\mathcal{U}$ called the *universe*, the elements of which are the sets denoted by the (monomorphic) type expressions. The universe is assumed to have the following properties.

Inhab Each element of $\mathcal{U}$ is a non-empty set.

Sub If $X \in \mathcal{U}$ and $\{\} \neq Y \subseteq X$, then $Y \in \mathcal{U}$.

Prod If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \times Y \in \mathcal{U}$. The set $X \times Y$ is the cartesian product, consisting of ordered pairs (x, y) with $x \in X$ and $y \in Y$, and with the usual set-theoretic coding of ordered pairs, that is $(x, y) = \{\{x\}, \{x, y\}\}$.

Pow If $X \in \mathcal{U}$, then the powerset $P(X) = \{Y : Y \subseteq X\}$ is also an element of $\mathcal{U}$.

Infty $\mathcal{U}$ contains a distinguished infinite set I.

Choice There is a distinguished element $\mathrm{ch} \in \prod_{X\in\mathcal{U}} X$. The elements of the product $\prod_{X\in\mathcal{U}} X$ are (dependently typed) functions: thus for all $X \in \mathcal{U}$, X is non-empty by **Inhab** and $\mathrm{ch}(X) \in X$ witnesses this.

[1]Note that the type subscript is omitted from terms when it is clear from the context that they are formulae, i.e. have type *bool*.

In set theory, functions are identified with their graphs, which are certain sets of ordered pairs. Thus the set $X \rightarrow Y$ of all functions from a set X to a set Y is a subset of $P(X \times Y)$; and it is a non-empty set when Y is non-empty. So **Sub**, **Prod** and **Pow** together imply that $\mathcal{U}$ also satisfies

Fun If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \rightarrow Y \in \mathcal{U}$.

By iterating **Prod**, one has that the cartesian product of any finite, non-zero number of sets in $\mathcal{U}$ is again in $\mathcal{U}$. $\mathcal{U}$ also contains the cartesian product of no sets, which is to say that it contains a one-element set (by virtue of **Sub** applied to any set in $\mathcal{U}$—**Infty** guarantees there is one); for definiteness, a particular one-element set will be singled out.

Unit $\mathcal{U}$ contains a distinguished one-element set $1 = \{0\}$.

Similarly, because of **Sub** and **Infty**, $\mathcal{U}$ contains two-element sets, one of which will be singled out.

Bool $\mathcal{U}$ contains a distinguished two-element set $2 = \{0, 1\}$.

The semantics of types is given relative to a model M which assigns to each type constant an element of $\mathcal{U}$ and to each n-ary type operator a function $\mathcal{U}^n \rightarrow \mathcal{U}$. A model M of Ω is *standard* if $M(bool)$ and $M(ind)$ are respectively the distinguished sets 2 and I in the universe $\mathcal{U}$.

The notion of a *type-in-context* is used in defining the semantics of types. A *type context* αs is just a finite list of distinct type variables, and a type-in-context $\alpha s.\sigma$ is a type σ together with a type context αs which contains (at least) all the type variables in σ. The meaning of a type in context $\alpha s.\sigma$, where the context αs is of length n, is then given by a function

$$[\![\alpha s.\sigma]\!]_M : \mathcal{U}^n \rightarrow \mathcal{U}$$

which is defined so that for any assignment of sets $Xs = (X_1, \ldots, X_n) \in \mathcal{U}^n$ to the type variables in αs (and hence to the type variables in σ), the element $[\![\alpha s.\sigma]\!]_M(Xs)$ of $\mathcal{U}$ is the corresponding set denoted by σ. The formal definition of $[\![\_]\!]_M$ is by induction on the structure of types [7].

The notion of a context is also employed in defining the meaning of terms. A *term-in-context* is written '$\alpha s, xs.t$' and consists of a term t together with a type context αs and a finite list of variables xs called a *variable context*. The variable context xs of a term-in-context $\alpha s, xs.t$ contains all the variables that occur free in t, and the type context αs contains all the type variables that occur in xs and t.

For the semantics of terms, a model consists of a type model (as described above) together with a function that assigns to each constant c with generic type σ an element of the set of functions

$$\prod_{Xs \in \mathcal{U}^n} [\![\alpha s.\sigma]\!]_M(Xs)$$

where n is the length of the type context αs. For a given model M, the meaning of a term-in-context $\alpha s, xs.t$, where αs has length n, xs has length m, and t has type τ, is given by a function $[\![\_]\!]_M$ defined by induction on terms such that:

$$[\![\alpha s, xs.t]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs)\right) \to [\![\alpha s.\tau]\!]_M(Xs)$$

where $xs = x_1, \ldots, x_m$ and σ_i is the type of the corresponding variable x_i. The idea is that given an assignment of sets

$$Xs = (X_1, \ldots, X_n) \in \mathcal{U}^n$$

to the type variables in αs (and hence to the free type variables in t) and given an assignment of elements

$$ys = (y_1, \ldots, y_m) \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs)$$

to the variables in xs (and hence to the variables that occur free in the term t), the result of $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ will be an appropriate element of the set $[\![\alpha s.\tau]\!]_M(Xs)$ denoted by the type of t.

A model M of Σ_Ω will be called *standard* if

- $M(\Rightarrow, bool{\to}bool{\to}bool) \in (2{\to}2{\to}2)$ is the standard implication function, sending $b, b' \in 2$ to

$$(b \Rightarrow b') = \begin{cases} 0 & \text{if } b = 1 \text{ and } b' = 0 \\ 1 & \text{otherwise} \end{cases}$$

- $M(=, \alpha{\to}\alpha{\to}bool) \in \prod_{X \in \mathcal{U}} . X{\to}X{\to}2$ is the function assigning to each $X \in \mathcal{U}$ the equality test function, sending $x, x' \in X$ to

$$(x =_X x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$$

- $M(\varepsilon, (\alpha{\to}bool){\to}\alpha) \in \prod_{X \in \mathcal{U}} .(X{\to}2){\to}X$ is the function assigning to each $X \in \mathcal{U}$ the choice function sending $f \in (X{\to}2)$ to

$$\mathrm{ch}_X(f) = \begin{cases} \mathrm{ch}(f^{-1}\{1\}) & \text{if } f^{-1}\{1\} \neq \{\} \\ \mathrm{ch}(X) & \text{otherwise} \end{cases}$$

  where $f^{-1}\{1\} = \{x \in X : f(x) = 1\}$. (Note that $f^{-1}\{1\}$ is in $\mathcal{U}$ when it is non-empty, by the property **Sub** of the universe $\mathcal{U}$. The function ch is given by property **Choice**.)

A sequent with hypotheses $\Gamma = \{t_1, \ldots, t_p\}$ and conclusion t is *satisfied* by a model M if any assignment of values to free variables that makes all the hypotheses true in M also makes the conclusion true in M. In particular, M satisfies the sequent if for all $Xs \in \mathcal{U}^n$ and all $ys \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs)$,

$$[\![\alpha s, xs.t_1]\!]_M(Xs)(ys) = 1, \quad \ldots, \quad [\![\alpha s, xs.t_p]\!]_M(Xs)(ys) = 1$$

imply that

$$[\![\alpha s, xs.t]\!]_M(Xs)(ys) = 1,$$

where $\alpha s, xs$ is any valid context for each of t, $t_1, \ldots, t_p$ with αs of length n, $xs = x_1, \ldots, x_m$, and σ_i the type of the corresponding variable x_i. $\Gamma \models_M t$ is written to mean that M satisfies the sequent with hypotheses Γ and conclusion t.

1.1.6. Deductive systems

A *deductive system* $\mathcal{D}$ is a set of pairs $(L, \mathcal{S})$ where L is a (possibly empty) list of sequents and $\mathcal{S}$ is a sequent.

A sequent $\mathcal{S}$ follows from a set of sequents Δ by a deductive system $\mathcal{D}$ if and only if there exist sequents $\mathcal{S}_1, \ldots, \mathcal{S}_n$ such that:

1. $\mathcal{S} = \mathcal{S}_n$, and
2. for all i such that $1 \leq i \leq n$, (a) either $\mathcal{S}_i \in \Delta$, or (b) $(L_i, \mathcal{S}_i) \in \mathcal{D}$ for some list L_i such that $L_i \subseteq \Delta \cup \{\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}\}$.

The sequence $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is called a *proof* of $\mathcal{S}$ from Δ with respect to $\mathcal{D}$.

The notation $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$ means that the sequent $(\{t_1, \ldots, t_n\},\ t)$ follows from Δ by $\mathcal{D}$. If either $\mathcal{D}$ or Δ is clear from the context then it may be omitted. In the case that there are no hypotheses (i.e. $n = 0$), just $\vdash t$ is written.

In practice, a particular deductive system is usually specified by a number of (schematic) *rules of inference*, which take the form

$$\frac{\Gamma_1 \ \vdash\ t_1 \qquad \cdots \qquad \Gamma_n \ \vdash\ t_n}{\Gamma \ \vdash\ t}$$

The sequents above the line are called the *hypotheses* of the rule and the sequent below the line is called its *conclusion*. Such a rule is schematic because it may contain metavariables standing for arbitrary terms of the appropriate types. Instantiating these metavariables with actual terms, one gets a list of sequents above the line and a single sequent below the line which together constitute a particular element of the deductive system. The instantiations allowed for a particular rule may be restricted by imposing a *side condition* on the rule.

The HOL deductive system

The deductive system of the HOL logic is specified by eight rules of inference, given below. The first three rules have no hypotheses; their conclusions can always be deduced. The identifiers in square brackets are the names of the ML functions in the HOL system that implement the corresponding inference rules. Any side conditions restricting the scope of a rule are given immediately below it.

Assumption introduction [ASSUME]

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

Reflexivity [REFL]

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

Beta-conversion [BETA_CONV]

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

- Where $t_1[t_2/x]$ is the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution.

Substitution [SUBST]

$$\frac{\Gamma_1 \ \vdash \ t_1 = t_1' \qquad \cdots \qquad \Gamma_n \ \vdash \ t_n = t_n' \qquad\qquad \Gamma \ \vdash \ t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \ \vdash \ t[t_1', \ldots, t_n']}$$

- Where $t[t_1, \ldots, t_n]$ denotes a term t with some free occurrences of subterms t_1, ..., t_n singled out and $t[t_1', \ldots, t_n']$ denotes the result of replacing each selected occurrence of t_i by t_i' (for $1 \leq i \leq n$), with suitable renaming of variables to prevent free variables in t_i' becoming bound after substitution.

Abstraction [ABS]

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

- Provided x is not free in Γ.

Type instantiation [INST_TYPE]

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

- Where $t[\sigma_1, \ldots, \sigma_n / \alpha_1, \ldots, \alpha_n]$ is the result of substituting, in parallel, the types σ_1, ..., σ_n for type variables α_1, ..., α_n in t, with the two restrictions: (i) none of the type variables α_1, ..., α_n occur in Γ; and (ii) no distinct variables in t become identified after the instantiation.

Discharging an assumption [DISCH]

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

- Where $\Gamma - \{t_1\}$ is the set subtraction of $\{t_1\}$ from Γ.

Modus Ponens [MP]

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

In addition to these eight rules, there are also five *axioms* which could have been regarded as rules of inference without hypotheses. This is not done, however, since it is most natural to state the axioms using some defined logical constants and the principle of constant definition has not yet been described. The axioms and the definitions of the extra logical constants they involve are given shortly.

The particular set of rules and axioms chosen to axiomatize the HOL logic is rather arbitrary. It is partly based on the rules that were used in the LCF logic PPλ, since HOL was implemented by modifying the LCF system.

Soundness theorem

The rules of the the HOL *deductive system are* sound *for the notion of satisfaction defined in 1.1.5: for any instance of the rules of inference, if a (standard) model satisfies the hypotheses of the rule it also satisfies the conclusion.*

1.1.7. Theories

A HOL *theory* $\mathcal{T}$ is a 4-tuple $\langle \mathsf{Struc}_{\mathcal{T}}, \mathsf{Sig}_{\mathcal{T}}, \mathsf{Axioms}_{\mathcal{T}}, \mathsf{Theorems}_{\mathcal{T}} \rangle$, where:

(i) $\mathsf{Struc}_{\mathcal{T}}$ is a type structure called the type structure of $\mathcal{T}$;

(ii) $\mathsf{Sig}_{\mathcal{T}}$ is a signature over $\mathsf{Struc}_{\mathcal{T}}$ called the signature of $\mathcal{T}$;

(iii) $\mathsf{Axioms}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the axioms of $\mathcal{T}$;

(iv) $\mathsf{Theorems}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the theorems of $\mathcal{T}$, with the property that every member follows from $\mathsf{Axioms}_{\mathcal{T}}$ by the HOL deductive system.

The sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$ of types and terms of a theory $\mathcal{T}$ are, respectively, the sets of types and terms constructable from the type structure and signature of $\mathcal{T}$:

$\mathsf{Types}_{\mathcal{T}} = \mathsf{Types}_{\mathsf{Struc}_{\mathcal{T}}}$ and $\mathsf{Terms}_{\mathcal{T}} = \mathsf{Terms}_{\mathsf{Sig}_{\mathcal{T}}}$.

A model of a theory $\mathcal{T}$ is specified by giving a (standard) model M of the underlying signature of the theory with the property that M satisfies all the sequents which are axioms of $\mathcal{T}$. Because of the Soundness Theorem, it follows that M also satisfies any sequents in the set of given theorems, $\mathsf{Theorems}_{\mathcal{T}}$.

The theory `MIN`

The *minimal theory* `MIN` is defined by:

$$\mathtt{MIN} = \langle \{(bool, 0),\ (ind, 0)\},\ \{\Rightarrow_{bool\to bool\to bool}, =_{\alpha\to\alpha\to bool}, \varepsilon_{(\alpha\to bool)\to\alpha}\},\ \{\},\ \{\}\rangle$$

Although the theory `MIN` contains only the minimal standard syntax, by exploiting the higher order constructs of HOL one can construct a rich collection of terms over it. The following theory introduces names for some of those terms that denote useful logical operations.

The theory `LOG`

The theory `LOG` has the same type structure as `MIN`. Its signature contains the constants in `MIN` and the following constants:

T_{bool}	F_{bool}	
$\neg_{bool\to bool}$	$\wedge_{bool\to bool\to bool}$	$\vee_{bool\to bool\to bool}$
$\forall_{(\alpha\to bool)\to bool}$	$\exists_{(\alpha\to bool)\to bool}$	
$\mathsf{One\_One}_{(\alpha\to\beta)\to bool}$	$\mathsf{Onto}_{(\alpha\to\beta)\to bool}$	$\mathsf{Type\_Definition}_{(\alpha\to bool)\to(\beta\to\alpha)\to bool}$

The following special notation is used in connection with these constants:

Notation	Meaning
$t_1\ \wedge\ t_2$	$\wedge\ t_1\ t_2$
$t_1\ \vee\ t_2$	$\vee\ t_1\ t_2$
$\forall\, x_\sigma.\ t$	$\forall(\lambda\, x_\sigma.\ t)$
$\forall\, x_1\ x_2\ \cdots\ x_n.\ t$	$\forall\, x_1.\ (\forall\, x_2.\ \cdots\ (\forall\, x_n.\ t)\ \cdots)$
$\exists\, x_\sigma.\ t$	$\exists(\lambda\, x_\sigma.\ t)$
$\exists\, x_1\ x_2\ \cdots\ x_n.\ t$	$\exists\, x_1.\ (\exists\, x_2.\ \cdots\ (\exists\, x_n.\ t)\ \cdots)$

The axioms of the theory LOG consist of the following sequents:

$$\begin{array}{ll}
\vdash & \mathsf{T} = ((\lambda\, x_{bool}.\ x) = (\lambda\, x_{bool}.\ x)) \\
\vdash & \forall = \lambda\, P_{\alpha\to bool}.\ \ P = (\lambda\, x.\ \mathsf{T}) \\
\vdash & \exists = \lambda\, P_{\alpha\to bool}.\ \ P(\varepsilon\ P) \\
\vdash & \mathsf{F} = \forall\, b_{bool}.\ \ b \\
\vdash & \neg = \lambda\, b.\ \ b \Rightarrow \mathsf{F} \\
\vdash & \wedge = \lambda\, b_1\ b_2.\ \forall\, b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b \\
\vdash & \vee = \lambda\, b_1\ b_2.\ \forall\, b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b) \\
\vdash & \mathsf{One\_One} = \lambda\, f_{\alpha\to\beta}.\ \forall\, x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2) \\
\vdash & \mathsf{Onto} = \lambda\, f_{\alpha\to\beta}.\ \forall\, y.\ \exists\, x.\ y = f\ x \\
\vdash & \mathsf{Type\_Definition} = \ \lambda\, P_{\alpha\to bool}\ rep_{\beta\to\alpha}.\mathsf{One\_One}\ rep \ \ \wedge \\
 & \qquad\qquad\qquad (\forall\, x.\ P\ x\ = \ (\exists\, y.\ x = rep\ y))
\end{array}$$

Finally, as for the theory MIN, the set $\mathsf{Theorems}_{\mathsf{LOG}}$ is taken to be empty.

Note that the axioms of the theory LOG are essentially *definitions* of the new constants of LOG as terms in the original theory MIN. The mechanism for making such extensions of theories by definitions of new constants is described in 1.1.10. The first seven axioms define the logical constants for truth, universal quantification, existential quantification, falsity, negation, conjunction and disjunction. The next two axioms define the properties of a function being one-one and onto; they will be used to express the axiom of infinity (see 1.1.7), amongst other things. The last axiom defines a constant used for type definitions (see 1.1.10).

The theory INIT

The theory INIT is obtained by adding the following five axioms to the theory LOG.

BOOL_CASES_AX	$\vdash \forall\, b.\ (b = \mathsf{T}) \vee (b = \mathsf{F})$
IMP_ANTISYM_AX	$\vdash \forall\, b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$
ETA_AX	$\vdash \forall f_{\alpha\to\beta}.\ (\lambda\, x.\ f\ x) = f$
SELECT_AX	$\vdash \forall\, P_{\alpha\to bool}\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$
INFINITY_AX	$\vdash \exists\, f_{ind\to ind}.\ \mathsf{One\_One}\ f\ \ \wedge\ \ \neg(\mathsf{Onto}\ f)$

The theory INIT is the initial theory of the HOL logic. A theory which extends INIT will be called a *standard theory.* It can be shown [7] that there is a unique standard model of INIT that satisfies the five axioms and interprets the constants defined in LOG as follows:

- $[\![\mathsf{T}_{bool}]\!] = 1 \in 2$
- $[\![\forall_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\forall]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} = X \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\exists_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\exists]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} \neq \{\} \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{F}_{bool}]\!] = 0 \in 2$

- $[\![\neg_{bool\to bool}]\!] \in 2\to 2$ sends $b \in 2$ to

$$[\![\neg]\!](b) = \begin{cases} 1 & \text{if } b = 0 \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\wedge_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\wedge]\!](b)(b') = \begin{cases} 1 & \text{if } b = 1 = b' \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\vee_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\vee]\!](b)(b') = \begin{cases} 0 & \text{if } b = 0 = b' \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{One\_One}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y) \in \mathcal{U}^2$ and $f \in (X\to Y)$ to

$$[\![\mathsf{One\_One}]\!](X,Y)(f) = \begin{cases} 0 & \text{if } f(x) = f(x') \text{ for some } x \neq x' \text{ in } X \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Onto}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y) \in \mathcal{U}^2$ and $f \in (X\to Y)$ to

$$[\![\mathsf{Onto}]\!](X,Y)(f) = \begin{cases} 1 & \text{if } \{f(x) : x \in X\} = Y \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Type\_Definition}_{(\alpha\to bool)\to(\beta\to\alpha)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to 2)\to(Y\to X)\to 2$ sends $(X,Y) \in \mathcal{U}^2$, $f \in (X\to 2)$ and $g \in (Y\to X)$ to

$$[\![\mathsf{Type\_Definition}]\!](X,Y)(f)(g) = \begin{cases} 1 & \text{if } [\![\mathsf{One\_One}]\!](Y,X)(g) = 1 \\ & \text{and } f^{-1}\{1\} = \{g(y) : y \in Y\} \\ 0 & \text{otherwise.} \end{cases}$$

1.1.8. Built-in theories and notations

The logical core of the HOL system is the theory `INIT`, however a number of useful theories are predefined or available as libraries (see 1.2). Some of these are associated with special notations that are supported by the parser and pretty-printer. These notations parse into standard terms and are thus only ‘syntactic sugar’. Their informal meaning will be given here; full details, including the underlying logical representation and associated theories, can be found in *Introduction to HOL* [7].

Pairs and tuples

Pairs are written as (t_1, t_2); tuples $(t_1, t_2, \ldots, t_n)$ parse to iterated pairs $(t_1, (t_2, \ldots, t_n))$. If t_1 has type σ_1 and t_2 has type σ_2 then (t_1, t_2) has type $(\sigma_1, \sigma_2)prod$, which may be writen as $\sigma_1 \times \sigma_2$.

Pairs may also be used as part of 'variable structures' in quantifications and λ-binding. For example, $\lambda(m, n).\ m + n$ and $\forall(m, n).\ m \leq n\ \vee\ n < m$.

Conditionals

The conditional $(t \rightarrow t_1 \mid t_2)$ intuitively means 'if t then t_1 else t_2' and abbreviates $\mathsf{Cond}\ t\ t_1\ t_2$, where Cond has type $bool{\rightarrow}\alpha{\rightarrow}\alpha{\rightarrow}\alpha$ and is defined by:

$$\mathsf{Cond}\ b\ x_1\ x_2\ =\ \varepsilon x.\ ((b = \mathsf{T}) \Rightarrow (x = x_1))\ \wedge\ ((b = \mathsf{F}) \Rightarrow (x = x_2))$$

Numerals and strings

Among the predefined types supplied with the HOL system are *num* (natural numbers) and *string* (strings). With each of these types there are infinite families of constants. In the case of *num* these are 0, 1, 2, etc.; in the case of *string* these have the form '$c_1 c_2 \cdots c_n$', where each c_i is a letter or numeral. The HOL parser recognises such numbers and strings as constants of the appropriate theory.

Restricted quantification

The terms $\forall\, x{::}t_1.\ t_2$ and $\exists\, x{::}t_1.\ t_2$ abbreviate $\forall\, x.\ t_1 \Rightarrow t_2$ and $\exists\, x.\ t_1\ \wedge\ t_2$, respectively. The restriction t_1 acts like a type in dependently typed systems and allows terms like $\forall\, m\ n.\ \forall\, i{::}\mathsf{from}(m, n).\ m \leq i\ \wedge\ i \leq n$ to be written (where from is a predicate on pairs of numbers). Combinations of variable structures and restriction are allowed.

Less useful, but also supported, are restricted ε-terms and λ-terms: $\varepsilon x{::}t_1.\ t_2$ abbreviates $\varepsilon x.\ t_1\ \wedge\ t_2$ and $\lambda\, x{::}t_1.\ t_2$ abbreviates $\lambda\, x.\ (t_1 \rightarrow t_2 \mid \varepsilon v.\mathsf{T})$. As with the restricted quantifiers, the bound variable can also be a variable structure.

`let`-terms

A basic `let`-term has the form `let` $x = t_1$ `in` t_2 and abbreviates $(\lambda\, x.\ t_2)t_1$. A local function binding like `let` $f\ x\ =\ t_1$ `in` t_2 abbreviates $(\lambda f.\ t_2)(\lambda\, x.\ t_1)$. The parameters of such function definitions can be paired; for example `let` $add(m, n) = m + n$ `in` $add(0, 1)$ abbreviates $(\lambda\, add.\ add(0, 1))(\lambda(m, n).\ m + n)$.

Multiple local bindings are allowed. Two equivalent forms are supported:

`let` $(x_1, x_2,\ \cdots\ , x_n) = (t_1, t_2,\ \cdots\ , t_n)$ `in` t
`let` $x_1 = t_1$ `and` $x_2 = t_2\ \cdots\ x_n = t_n$ `in` t

The second of these allows function definitions, for example:

`let` $x = 1$ `and` $add(m, n) = m + n$ `in` $add(0, x)$

Lists and sets

Theories of lists and sets are predefined; lists are built-in to HOL, but sets are a library. A list is a term of type σ *list*; individual lists may be input with the notation $[t_1;\ \ldots\ ;\ t_n]$. The empty list is $[\,]$. A set is a term of type σ *set*; finite sets may be input with the notation $\{t_1,\ \ldots\ , t_n\}$. The empty set is $\{\}$. The set abstraction notation $\{t_1 \mid t_2\}$ is also allowed and denotes the set of t_1s such that t_2. For example $\{x + y \mid x < 10\ \wedge\ y < 10\}$ denotes the set of sums of pairs of numbers less than 10.

1.1.9. Consistency

A (standard) theory is *consistent* if it is not the case that every sequent over its signature can be derived from the theory's axioms using the HOL logic, or equivalently, if the particular sequent $\vdash \mathsf{F}$ cannot be so derived.

The existence of a (standard) model of a theory is sufficient to establish its consistency. For by the Soundness Theorem, any sequent that can be derived from the theory's axioms will be satisfied by the model, whereas the sequent $\vdash \mathsf{F}$ is never satisfied in any standard model. So in particular, the initial theory `INIT` is consistent.

However, it is possible for a theory to be consistent but not to possess a standard model. This is because the notion of a *standard* model is quite restrictive—in particular there is no choice how to interpret the integers and their arithmetic in such a model. The famous incompleteness theorem of Gödel ensures that there are sequents which are satisfied in all standard models (i.e. which are 'true'), but which are not provable in the HOL logic.

1.1.10. Extensions of theories

A theory $\mathcal{T}'$ is said to be an *extension* of a theory $\mathcal{T}$ if and only if $\mathsf{Struc}_{\mathcal{T}} \subseteq \mathsf{Struc}_{\mathcal{T}'}$, $\mathsf{Sig}_{\mathcal{T}} \subseteq \mathsf{Sig}_{\mathcal{T}'}$, $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Axioms}_{\mathcal{T}'}$ and $\mathsf{Theorems}_{\mathcal{T}} \subseteq \mathsf{Theorems}_{\mathcal{T}'}$.

The mechanisms for making extensions of theories in HOL are: (i) extension by a constant definition, (ii) extension by a constant specification and (iii) extension by a type definition. These all produce *definitional extensions* in the sense that they extend a theory by adding new constants and types which are defined in terms of properties of existing ones. Their key property is that the extended theory possesses a standard model if the original theory does. So a series of these extensions starting from the theory `INIT` is guaranteed to result in a theory with a standard model, and hence in a consistent theory. It is also possible to extend theories simply by adding new uninterpreted constants and types. This preserves consistency, but is unlikely to be useful without additional axioms. However, when adding arbitrary new axioms, there is no guarantee that consistency is preserved.

Extension by constant definition

A *constant definition* over a signature Σ_Ω is a formula of the form $\mathsf{c}_\sigma = t_\sigma$, such that:

(i) c is not the name of any constant in Σ_Ω;

(ii) t_σ a closed term in $\mathsf{Terms}_{\Sigma_\Omega}$;

(iii) all the type variables occurring in t_σ also occur in σ.

Given a theory $\mathcal{T}$ and such a constant definition over $\mathsf{Sig}_{\mathcal{T}}$, then the *definitional extension* of $\mathcal{T}$ by $\mathsf{c}_\sigma = t_\sigma$ is the theory $\mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle$ defined by:

$$\mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle \;=\; \langle \mathsf{Struc}_{\mathcal{T}},\ \mathsf{Sig}_{\mathcal{T}} \cup \{(\mathsf{c}, \sigma)\},\ \mathsf{Axioms}_{\mathcal{T}} \cup \{\mathsf{c}_\sigma = t_\sigma\},\ \mathsf{Theorems}_{\mathcal{T}} \rangle$$

Note that the mechanism of extension by constant definition has already been used implicitly in forming the theory `LOG` from the theory `MIN` in 1.1.7. Thus with the notation of this section one has

$$
\begin{array}{lll}
\mathtt{LOG} = \mathtt{MIN} & +_{def}\langle \mathsf{T} & = \ ((\lambda\, x_{bool}.\ x) = (\lambda\, x_{bool}.\ x))\rangle \\
& +_{def}\langle \forall & = \ \lambda\, P_{\alpha\to bool}.\ \ P = (\lambda\, x.\ \mathsf{T})\rangle \\
& +_{def}\langle \exists & = \ \lambda\, P_{\alpha\to bool}.\ \ P(\varepsilon\ P)\rangle \\
& +_{def}\langle \mathsf{F} & = \ \forall\, b_{bool}.\ \ b\rangle \\
& +_{def}\langle \neg & = \ \lambda\, b.\ \ b \Rightarrow \mathsf{F}\rangle \\
& +_{def}\langle \wedge & = \ \lambda\, b_1\ b_2.\ \forall\, b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b\rangle \\
& +_{def}\langle \vee & = \ \lambda\, b_1\ b_2.\ \forall\, b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)\rangle \\
& +_{def}\langle \mathsf{One\_One} & = \ \lambda\, f_{\alpha\to\beta}.\ \forall\, x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1{=}x_2)\rangle \\
& +_{def}\langle \mathsf{Onto} & = \ \lambda\, f_{\alpha\to\beta}.\ \forall\, y.\ \exists\, x.\ y = f\ x\rangle \\
& +_{def}\langle \mathsf{Type\_Definition} & = \ \lambda\, P_{\alpha\to bool}\ rep_{\beta\to\alpha}. \\
& & \quad \mathsf{One\_One}\ rep\ \ \wedge \\
& & \quad (\forall\, x.\ P\ x\ \ =\ \ (\exists\, y.\ x = rep\ y))\rangle
\end{array}
$$

Remark Condition (iii) in the definition of what constitutes a correct constant definition is an important restriction without which consistency could not be guaranteed. To see this, consider the term $\exists\, f_{\alpha\to\alpha}.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$, which expresses the proposition that (the set of elements denoted by the) type α is infinite. The term contains the type variable α, whereas the type of the term, *bool*, does not. Thus by (iii)

$$\mathsf{c}_{bool} = \exists\, f_{\alpha\to\alpha}.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$$

is not allowed as a constant definition. The problem is that the meaning of the right hand side of the definition varies with α, whereas the meaning of the constant on the left hand side is fixed, since it does not contain α. Indeed, if we were allowed to extend the consistent theory `INIT` by this definition, the result would be an inconsistent theory. For instantiating α to *ind* in the right hand side results in a term that is provable from the axioms of `INIT`, and hence $\mathsf{c}_{bool} = \mathsf{T}$ is provable in the extended theory. But equally, instantiating α to *bool* makes the negation of the right hand side provable from the axioms of `INIT`, and hence $\mathsf{c}_{bool} = \mathsf{F}$ is also provable in the extended theory. Combining these theorems, one has that $\mathsf{T} = \mathsf{F}$, i.e. F is provable in the extended theory.

Extension by constant specification

Constant specifications introduce constants (or sets of constants) that satisfy arbitrary given (consistent) properties. For example, a theory could be extended by a constant specification to have two new constants b_1 and b_2 of type *bool* such that $\neg(\mathsf{b}_1 = \mathsf{b}_2)$. This specification does not uniquely define b_1 and b_2, since it is satisfied by either $\mathsf{b}_1 = \mathsf{T}$ and $\mathsf{b}_2 = \mathsf{F}$, or $\mathsf{b}_1 = \mathsf{F}$ and $\mathsf{b}_2 = \mathsf{T}$. To ensure that such specifications are consistent, they can only be made if it has already been proved that the properties which the new constants are to have are consistent. This rules out, for example, introducing three boolean constants b_1, b_2 and b_3 such that $\mathsf{b}_1 \neq \mathsf{b}_2$, $\mathsf{b}_1 \neq \mathsf{b}_3$ and $\mathsf{b}_2 \neq \mathsf{b}_3$.

Suppose $\exists\, x_1 \ldots x_n.\ t$ is a formula, with $x_1, \ldots, x_n$ distinct variables. If $\vdash\ \exists\, x_1 \ldots x_n.\ t$, then a constant specification allows new constants c_1, ..., c_n to be introduced satisfying:

$$\vdash\ \ t[\mathsf{c}_1, \ldots, \mathsf{c}_n/x_1, \ldots, x_n]$$

where $t[\mathsf{c}_1, \ldots, \mathsf{c}_n/x_1, \ldots, x_n]$ denotes the result of simultaneously substituting $\mathsf{c}_1, \ldots, \mathsf{c}_n$ for free occurrences of $x_1, \ldots, x_n$, respectively. Of course, the type of each constant c_i must be the same as the type of the corresponding variable x_i. To ensure that this extension mechanism preserves the property of possessing a model, a further more technical requirement is imposed on these types: they must each contain all the type variables occurring in t.

Formally, a *constant specification* for a theory $\mathcal{T}$ is given by:

Data $\langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda\, x_{1_{\sigma_1}} \ldots x_{n_{\sigma_n}}.\ t_{bool}\rangle$

Conditions

(i) $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are distinct names that are not the names of any constants in $\mathsf{Sig}_{\mathcal{T}}$.

(ii) $\lambda\, x_{1_{\sigma_1}} \ldots x_{n_{\sigma_n}}.\ t_{bool} \ \in\ \mathsf{Terms}_{\mathcal{T}}$.

(iii) $tyvars(t_{bool}) \ =\ tyvars(\sigma_i)$ for $1 \leq i \leq n$.

(iv) $\exists\, x_{1_{\sigma_1}} \ldots x_{n_{\sigma_n}}.\ t \ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by such a constant specification is denoted by:

$$\mathcal{T}{+_{spec}}\langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda\, x_{1_{\sigma_1}} \ldots x_{n_{\sigma_n}}.\ t_{bool}\rangle$$

and is defined to be the theory:

$$\langle \mathsf{Struc}_{\mathcal{T}},\ \mathsf{Sig}_{\mathcal{T}} \cup \{\mathsf{c}_{1_{\sigma_1}}, \ldots, \mathsf{c}_{n_{\sigma_n}}\},\ \mathsf{Axioms}_{\mathcal{T}} \cup \{t[\mathsf{c}_1, \ldots, \mathsf{c}_n/x_1, \ldots, x_n]\},\ \mathsf{Theorems}_{\mathcal{T}}\rangle$$

Extension by a constant definition, $\mathsf{c}_\sigma = t_\sigma$, is a special case of extension by constant specification. For let t' be the formula $x_\sigma = t_\sigma$, where x_σ is a variable not occurring in t_σ. Then clearly $\vdash\ \exists\, x_\sigma.\ t'$ and one can apply the method of constant specification to obtain the theory $\mathcal{T}{+_{spec}}\langle \mathsf{c}, \lambda\, x_\sigma.\ t'\rangle$. But since $t'[\mathsf{c}_\sigma/x_\sigma]$ is just $\mathsf{c}_\sigma = t_\sigma$, this extension yields exactly the theory $\mathcal{T}{+_{def}}\langle \mathsf{c}_\sigma = t_\sigma\rangle$.

Extension by type definition

It is useful to have a mechanism for introducing new types which are subtypes of existing ones. Such types are defined in HOL by introducing a new type constant and asserting an axiom that characterizes it as denoting a set in bijection with a non-empty subset of an existing type (called the *representing type*). For example, the type `num` is defined to be equal to a countable subset of the type `ind`, which is guaranteed to exist by `INFINITY_AX`.

As well as defining types, it is also convenient to be able to define type operators. An example would be a type operator *inj* which maps a set to the set of one-to-one (i.e. injective) functions on it. The subset of $\sigma{\rightarrow}\sigma$ representing $(\sigma)inj$ would be defined by the predicate $\mathsf{One\_One}$. Another example would be a binary cartesian product type operator *prod*. This is defined by choosing a representing type containing two type variables, say $\sigma[\alpha_1, \alpha_2]$, such that for any types σ_1 and σ_2, a subset of $\sigma[\sigma_1, \sigma_2]$ represents the cartesian product of σ_1 and σ_2.

Types in HOL must denote non-empty sets. Thus it is only consistent to define a new type isomorphic to a subset specified by a predicate p, if there is at least one thing for which p holds, i.e. $\vdash \exists x.\ p\ x$. For example, it would be inconsistent to define a binary type operator *iso* such that $(\sigma_1, \sigma_2)iso$ denoted the set of one-to-one functions from σ_1 *onto* σ_2 because for some values of σ_1 and σ_2 the set would be empty; for example $(ind, bool)iso$ would denote the empty set. To avoid this, a precondition of defining a new type is that the representing subset is non-empty.

To summarize, a new type is defined by specifying an existing type, then specifying a subset of this type, then proving that this subset is non-empty and finally specifying that the new type is isomorphic to this subset. In more detail, defining a new type $(\alpha_1, \ldots, \alpha_n)op$ consists in:

1. Specifying a type, σ say, whose type variables are included in α_1, ..., α_n. The type σ is called the *representing type*, and the type $(\alpha_1, \ldots, \alpha_n)op$ is intended to be isomorphic to a subset of σ.

2. Specifying a closed term, p say, of type $\sigma \rightarrow bool$ and whose type variables are included in α_1, ..., α_n. The term p is called the *characteristic function*. This defines the subset of σ to which $(\alpha_1, \ldots, \alpha_n)op$ is to be isomorphic.[2]

3. Proving $\vdash \exists x_\sigma.\ p\ x_\sigma$.

4. Asserting an axiom saying that $(\alpha_1, \ldots, \alpha_n)op$ is isomorphic to the subset of σ selected by p.

To make this formal, the theory `LOG` provides the polymorphic constant $\mathsf{Type\_Definition}$ defined in 1.1.7. The formula $\exists f_{(\alpha_1,\ldots,\alpha_n)op \rightarrow \sigma}.\ \mathsf{Type\_Definition}\ p\ f$ asserts that there exists a one-to-one map f from $(\alpha_1, \ldots, \alpha_n)op$ onto the subset of elements of σ for which p is true. Hence, the axiom that characterizes $(\alpha_1, \ldots, \alpha_n)op$ is:

$$\vdash \exists f_{(\alpha_1,\ldots,\alpha_n)op \rightarrow \sigma}.\ \mathsf{Type\_Definition}\ p\ f$$

Defining a new type $(\alpha_1, \ldots, \alpha_n)op$ in a theory $\mathcal{T}$ thus consists of introducing op as a new n-ary type operator and the above axiom as a new axiom. Formally, a *type definition* for a theory $\mathcal{T}$ is given by:

Data $\langle (\alpha_1, \ldots, \alpha_n)op,\ \sigma,\ p_{\sigma \rightarrow bool} \rangle$

Conditions

(i) (op, n) is not the name of a type constant in $\mathsf{Struc}_\mathcal{T}$.

(ii) σ is a type containing the type variables α_1, ..., α_n with $\sigma \in \mathsf{Types}_\mathcal{T}$.

(iii) $p_{\sigma \rightarrow bool}$ is a closed term in $\mathsf{Terms}_\mathcal{T}$ whose type variables occur in $\alpha_1, \ldots, \alpha_n$.

(iv) $\exists x_\sigma.\ p\ x\ \in\ \mathsf{Theorems}_\mathcal{T}$.

[2]The reason for restricting p to be closed, i.e. to have no free variables, is that otherwise for consistency the defined type operator would have to *depend* upon (i.e. be a function of) those variables.

The extension of a standard theory $\mathcal{T}$ by a such a type definition is written

$$\mathcal{T} +_{tydef} \langle (\alpha_1, \ldots, \alpha_n)op, \sigma, p \rangle$$

and defined to be the theory

$$\begin{array}{l}\langle \mathsf{Struc}_{\mathcal{T}} \cup \{(op, n)\}, \\ \ \mathsf{Sig}_{\mathcal{T}}, \\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{\exists f_{(\alpha_1,\ldots,\alpha_n)op\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f\}, \\ \ \mathsf{Theorems}_{\mathcal{T}}\rangle\end{array}$$

This method of type definition was suggested by Mike Fourman.

There is also a notion of type specification [7] for making 'loose specifications' of types. This is analogous to constant specification, but is not yet implemented and so is not described here.

The primitive defining mechanisms of the HOL logic are tedious to use, so a variety of derived mechanisms have been implemented to generate primitive definitions automatically from higher level inputs.

Primitive recursive definitions

It follows from the definition of numbers in HOL that every primitive recursion specifies a function. A tool is provided to convert such recursive specifications into proper (non-recursive) definitions.

Concrete types

Types similar to programmining language records can be introduced by supplying an equation of the form:

$$op \ = \ C_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \ \cdots \ | \quad C_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

where each ty_i^j is either a type expression already defined as a type in the current theory (this type expression must not contain op) or is the name op itself. A specification of this form describes an n-ary type operator op, where n is the number of distinct type variables in the types ty_i^j on the right hand side of the equation. If $n = 0$ then op is a type constant; otherwise op is an n-ary type operator. The concrete type described has m distinct constructors $C_1, \ldots, C_m$ where $m \geq 1$. Each constructor C_i takes k_i arguments, where $k_i \geq 0$; and the types of these arguments are given by the type expressions ty_i^j for $1 \leq j \leq k_i$. If one or more of the type expressions ty_i^j is the type op itself, then the equation specifies a *recursive* type. In any specification of a recursive type, at least one constructor must be non-recursive—i.e. all its arguments must have types which already exist in the current theory.

The logical type described by an input of the form shown above is intended to denote the set of all values which can be finitely generated using the constructors $C_1, \ldots, C_m$, where each constructor is one-to-one and any two different constructors yield different values. Every value of this type will be denoted by some term of the form:

$$C_i\ t_i^1\ \ldots\ t_i^{k_i}$$

where t_i^j is a term of type ty_i^j for $1 \leq j \leq k_i$. In addition, any two terms:

$$C_i\ t_i^1\ \ldots\ t_i^{k_i} \qquad \text{and} \qquad C_j\ t_j^1\ \ldots\ t_j^{k_j}$$

denote equal values exactly when their constructors are the same (i.e. $i = j$) and these constructors are applied to equal arguments (i.e. $t_i^n = t_j^n$ for $1 \leq n \leq k_i$).

The type definition package in HOL (which is due to T.F. Melham [9]) converts a type specification into a primitive type definition and automatically derives tools for making recursive definitions over the new type and performing proofs by structural induction.

1.2. The HOL system

The primary interface to HOL is the functional programming language ML (the name 'ML' is an acronym for 'Meta Language'). There is also a graphics interface implemented in Centaur [16] that can be mounted on top of the ML interface. Theorem proving tools are functions in ML. It is intended that users of HOL will build their own application-specific theorem-proving infrastructure by writing programs in ML.

HOL can be used for directly proving theorems but more often its role is as a theorem proving environment for implementing special purpose formal verification systems.

HOL provides considerable built-in theorem-proving infrastructure, including a powerful rewriting subsystem based on Paulson's higher-order rewriting combinators [13].

There is a library facility containing useful theories and tools that have been packaged for general use. So far about thirty libraries have been contributed by users from both universities and industry. Syntax processing libraries are provided to enable application-specific languages to be embedded in higher order logic. A decision procedure for tautologies and a semi-decision procedure for linear arithmetic are also provided as libraries (these procedures, which were written by Richard Boulton, work by performing sequences of primitive inferences and are thus guaranteed to be logically sound).

The HOL library grows with each new release of the system. In addition to the library facility, HOL also maintains a repository of contributed material that is not required to meet the same quality control standards as the library. This provides a vehicle for users to distribute prototypes, documents, etc.

The HOL system comes with comprehensive documentation. There is a detailed description of the system, which includes the formal semantics of the version of higher order logic used, a manual for the ML programming language and a description of the theorem proving infrastructure. The HOL reference manual documents every ML function in the system. The text of this manual can be accessed by a help system and an X-windows browsing tool. There is also a tutorial introduction and a training course (including exercises and solutions). All the documentation is public domain and the LaTeX sources are distributed with the system. Some of the libraries are public domain, but others are copyrighted by their authors.

1.2.1. The history of HOL

The approach to mechanizing formal proof used in HOL is due to Robin Milner [8]. He designed a system called LCF (Logic for Computable Functions), which was intended for interactive automated reasoning about higher order recursively defined functions.

The original LCF was implemented at Edinburgh in the early 1970s, and is now referred to as 'Edinburgh LCF'. Its code was ported from Stanford Lisp to Franz Lisp by Gérard Huet at INRIA, and was used in a French research project called 'Formel'. Huet's Franz Lisp version of LCF was further developed at Cambridge by Larry Paulson, and became known as 'Cambridge LCF' [14].

The HOL system is implemented on top of an early version of Cambridge LCF, and consequently many features of both Edinburgh and Cambridge LCF were inherited by HOL. For example, the formulation of higher order logic used is not the classical one due to Church [4], but incorporates LCF-style type variables. This provides, within the logic, some of the meta-theoretic notations used informally by Church. A second influence of LCF is the explicit management of logical theories. These support the splitting of complicated specifications into a coherent structure. A feature of HOL not found in LCF is the separation of consistency-preserving definitional principles from arbitrary axioms. Most developments using HOL are purely definitional and are thus guaranteed to be consistent.

The original version of HOL is called HOL88 and is in the public domain. It can be obtained by `ftp` from sites in the UK and USA (see below). HOL88 is implemented in Lisp and runs on any platform that supports Franz Lisp or Common Lisp (e.g. IBM PC, Sun, MIPS, HP workstation, Apple Macintosh). HOL88 uses an early version of ML derived from LCF. A new language, derived from this early ML, called 'Standard ML', was designed and implemented by a team lead by Robin Milner during the 1980s [12].

Two new versions of HOL implemented in Standard ML are available: HOL90 from the University of Calgary is a public domain system intended to be used with Standard ML of New Jersey; ICL HOL is a commercial system intended to support applications in the security critical area and particularly with specifications written in Z; it is implemented in Poly/ML. HOL90 provides, within Standard ML, essentially the same facilities as HOL88 and is intended to eventually replace it. ICL HOL is somewhat different (although the underlying concepts are the same). All three systems support the same logic; they only differ in the theorem proving infrastructure provided.

1.2.2. Overview of the theorem-proving infrastructure

ML is an interactive typed functional programming language. It has a type system that forms the basis of the security of theorem-proving in HOL [11].

Note that there is a potential for confusing the type system of the logic (see 1.1.1) and the completely separate type system of the metalanguage ML.

There are three ML types that form the interface to the logic: `type`, `term` and `thm`. Values of these types are data-structures that represent types, terms and theorems of the HOL logic in ML. Functions are provided in ML to manipulate types and terms, for example there is a function `dest_comb` that splits a function application t_1 t_2 into the component terms t_1 and t_2. The inverse of this destructor is an ML function `mk_comb`.

Values of ML type `thm` represent theorems of the HOL logic. There are five predefined ML identifiers of type `thm`: `BOOL_CASES_AX`, `IMP_ANTISYM`, `ETA_AX`, `SELECT_AX` and `INFINITY_AX`; these correspond to the five axioms in the theory `INIT` (see 1.1.7).

The ML type system ensures that the only way to generate more theorems is to apply ML functions that return values of type `thm`. In the core of the system there are only eight such functions: `ASSUME`, `REFL`, `BETA_CONV`, `SUBST`, `ABS`, `INST_TYPE`, `DISCH` and `MP`; these correspond to the eight rules of inference of the HOL deductive system (see 1.1.6).

The only way of creating values of ML type `thm` is to apply a sequence of these functions, i.e. a sequence of applications of inference rules. Thus all values of ML type `thm` are theorems of the HOL deductive system. It is possible to generate a trace of the applications of the primitive rules and so obtain a formal proof in the sense of 1.1.6; this is useful for independent proof auditing. The explicit proof facility is available in HOL88 Version 2.02.

In practice, it would be very tedious if one started with only the five axioms and eight rules of inference. When the HOL system is built hundreds of theorems are pre-proved. Theorems are stored in theories on disc in *theory files*. Many useful theories are generated automatically and saved when the system is build. For example, theories of lists, sets, bags, trees, strings, various kinds of numbers (including real numbers constructed via a type definition based on Dedekind cuts), n-bit words, group theory, fixedpoints, order structures etc. Some of these theories are in the main system and some in libraries.

Many theorem proving tools are predefined; when invoked these can cause thousands of primitive inference steps to be performed automatically. Some of these tools are in the main system and some are in libraries. For example, there is a semi decision procedure for a fragment of arithmetic. This takes a term t of ML type `term` as an argument and then computes – by a sequence of primitive inferences – the theorem $\vdash t$ of type `thm`. This is unlike other systems in which programs implementing complex inference mechanisms, like decision procedures, are simply trusted.

In LCF-style systems like HOL, one only needs to trust the programs implementing the core of the system (e.g. the eight primitive inference rules); derived rules are guaranteed to be sound because when they are invoked they expand to a sequence of calls of the primitives. Even the need to trust the core can be eliminated by explicitly generating a formal proof and having it independently checked.

The LCF methodology offers very high security, but does incur a performance penalty due to the expansion of every derived rule into sequences of primitive inference steps. However, specialized programming techniques and heavy optimization have made this penalty surprisingly small [3].

Rewriting

A particularly important collection of proof tools concern rewriting, i.e. the repeated application of equational theorems $\vdash t_1 = t_2$ to replace instances of t_1 by the corresponding instance of t_2. Such equations arise in many ways, e.g. as definitions of constants or as laws like associativity and commutativity. HOL provides a number of 'brute force' tools for repeatedly rewriting with lists of equations.

The rewriting strategy may be adjusted to scan in various orders through terms, such as bottom up or top-down. HOL also provides tools for the fine grain control of rewriting. For example, the unrestricted use of commutativity laws leads to infinite loops, so one may only want to apply such laws in restricted ways. The mechanism of *conversions*, developed by Paulson [13], is available for such cases. Knuth Bendix completion is available as a derived rule (it was contributed by Konrad Slind).

Goal directed proof: tactics and tacticals

Theorems are not normally proved in HOL by applying inference rules directly (although sometimes powerful derived ones like decision procedures are used this way). It is more usual to use the built-in subgoal package to manage the search for a proof in a goal directed fashion. This is based around the notion of *tactics* originally developed by Milner for LCF. The idea is that one starts with a sequent, called a *goal*, and then uses subgoaling functions (called tactics) to split it into subgoals, subsubgoals etc. Eventually all the subgoals will be instances of already proved theorems and can be trivially solved. The subgoal package then automatically generates a theorem corresponding to the original goal. This subgoaling process can either be driven by executing ML commands explicitly, or it can by driven by pointing and clicking on parts of goals displayed on the screen via the Centaur interface [16].

Just as ML functions representing rules of inference can be combined to obtain complex derived rules, so tactics can be combined (using operators called *tacticals*) to obtain more complex tactics. HOL comes equipped with predefined tactics for rewriting and for applying decision procedures (e.g. for tautolgies and subsets of arithmetic). Application specific verification systems can be implemented by defining special purpose tactics, e.g. for verification condition generation [6].

1.2.3. Getting and using HOL

The HOL system can be obtained from `ted.cs.uidaho.edu` (129.101.100.20) by anonymous `ftp`; it is in the directory `~ftp/pub/hol`. It is also available from `ftp.cl.cam.ac.uk` (128.232.0.56) in the directory `hol`.

There is an electronic mailing list for discussing HOL and disseminating news about it. This list is joined by sending email to: `info-hol-request@ted.cs.uidaho.edu`.

There is an annual HOL users meeting. The tradition is that this alternates between Europe and North America. In 1991 the meeting was at the University of California at Davis [2]. In 1992 the meeting was at IMEC in Leuven, Belgium [5], in 1993 it was in Canada at the University of British Columbia and in 1994 it will be at the University of Malta.

Bibliography

REFERENCES

1 P. D. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Computer Science and Applied Mathematics Series. Academic Press, 1986.

2 M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors. *The 1991 International Workshop on the HOL Theorem Proving System and its Applications*, University of California at Davis, August 1991. IEEE Computer Society Press.

3 R. J. Boulton. On efficiency in theorem provers which fully expand proofs into primitive inferences. Technical Report 248, University of Cambridge Computer Laboratory, February 1992.

4 A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

5 L. J. M. Claesen and M. J. C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, 21–24 September 1992. IFIP transactions A-20, Elseview North-Holland.

6 M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

7 M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

8 M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

9 T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

10 T. F. Melham. The HOL logic extended with quantification over type variables. In *Higher Order Logic Theorem Proving and its Applications*, pages 3–17, Leuven, Belgium, 21–24 September 1992. IFIP transactions A-20, Elsevier North-Holland.

11 R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

12 R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

13 L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

14 L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

15 L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

16 L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, volume 17(5) of *Software Engineering Notes*, Tyson's Corner, Va, USA, 1992. ACM Press.