T. F. Melham, 'The HOL Logic Extended with Quantification over Type Variables',
Formal Methods in System Design, vol. 3, nos. 1–2 (August 1993), pp. 7–24.

The HOL Logic Extended with Quantification over Type Variables

THOMAS F. MELHAM

University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street,
Cambridge, CB2 3QG, England.

Abstract. The HOL system is an LCF-style mechanized proof-assistant for conducting proofs in higher order logic. This paper discusses a proposal to extend the primitive basis of the logic underlying the HOL system with a very simple form of quantification over types. It is shown how certain practical problems with using the definitional mechanisms of HOL would be solved by the additional expressive power gained by making this extension.

Keywords: types, higher order logic, theorem proving.

1 Introduction

The HOL system [4] is a mechanized proof-assistant developed by Mike Gordon at the University of Cambridge for conducting proofs in higher order logic. The version of higher order logic mechanized by HOL is essentially Church's formulation of simple type theory [2] extended with explicit rules of definition and with object-language polymorphism of the kind developed by Milner for the LCF logic PPλ [5]. In this paper a further extension to the logic is proposed—namely, the addition of a very limited form of object-language quantification over types. The motivation for this extension comes from a particular technical problem that arises when using the definitional mechanisms provided by the HOL logic. This paper explains the problem in detail and shows how the proposed extension solves it.

It is assumed that the reader is familiar with the details of the formulation of higher order logic given in the HOL system manual [4]. The present paper aims, however, to provide enough detail for readers unfamiliar with this material to understand the general idea behind this work.

2 Types and polymorphism in HOL

Type expressions in the HOL logic have the following syntax:

$$\sigma \quad ::= \quad c \quad | \quad \alpha \quad | \quad \sigma_1 \rightarrow \sigma_2 \quad | \quad (\sigma_1, \ldots, \sigma_n)op$$

where σ, σ_1, …, σ_n range over types, c ranges over type constants, α ranges over type variables, and op ranges over n-ary type operators (for $n \geq 1$). It is the inclusion of type variables in this syntax that makes the HOL logic polymorphic. Typing of terms takes place within the context of an assignment of generic types to constants, and a constant is well-typed at any substitution instance of its generic type.

Theorems that contain polymorphic types are also true for any substitution instance of them, so there is a limited form of implicit universal quantification over types in the HOL logic. More precisely, there is an implicit universal quantification over type variables at the level of *sequents*. A sequent $\Gamma \vdash \mathrm{P}$ means that P is provable by natural deduction from the hypotheses Γ. All occurrences of a type variable in such a sequent are identified in the semantics, and there is an implicit universal quantification over the value of this variable whose scope is the entire sequent. This is reflected in the primitive rule of type instantiation shown below.

$$\frac{\Gamma \vdash \mathrm{P}}{\Gamma \vdash \mathrm{P}[\sigma_1, \ldots, \sigma_n \,/\, \alpha_1, \ldots, \alpha_n]} \quad \alpha_1,\ \ldots,\ \alpha_n \text{ not in } \Gamma.$$

where $\mathrm{P}[\sigma_1, \ldots, \sigma_n \,/\, \alpha_1, \ldots, \alpha_n]$ means the result of simultaneously substituting the type σ_i for the type variable α_i for $1 \leq i \leq n$ at every occurrence of α_i in the term P. The side condition on this rule is a consequence of the fact that if a type variable occurs in both the hypotheses and the conclusion of a sequent, then both occurrences are assumed to denote the same set. In fact, from this primitive rule one can derive the more general instantiation rule

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

which allows one to substitute types for type variables throughout an entire sequent.

There is an additional side condition on both of the rules shown above, namely the condition that no two distinct term variables in P (and, for the second rule, Γ) may become identified as a result of doing the substitution. The HOL system enforces this side condition by renaming variables when possible and otherwise rejecting the substitution.

3 Motivation

The proposal of this paper is to extend the very simple form of polymorphism described above with limited object-language quantification over types. In particular, we wish to add primitive terms to the HOL logic of the forms $\forall\alpha.\,\mathrm{P}$ and $\exists\alpha.\,\mathrm{P}$, where α is a type variable and P is a boolean term. Informally, the intended interpretation is that $\mathrm{P}[\sigma/\alpha]$ is true for *all* types σ and for *some* type σ, respectively. Note that we are not proposing an extension to the type language of HOL—the quantifications $\forall\alpha.\,\mathrm{P}$ and $\exists\alpha.\,\mathrm{P}$ are new term constructs, and not type constructs of the kind found (for example) in Girard's system *F* [3]. The extended logic proposed here resembles system Q, a transfinite type theory due to Andrews [1]. It is, however, still much weaker than Andrews' system.

The motivation for this extension originally arose in connexion with work on new derived rules of definition for HOL—particularly in connexion with work on derived rules for defining abstract data types. The natural approach to these derived rules runs into problems with the primitive type and constant specification rules. The following sections explain this motivational background. Readers primarily interested in the semantics and other details of the proposed extension may wish to skip to section 4.

3.1 Definitions in HOL

The HOL user community has a strong tradition of taking a purely definitional approach to using higher order logic. The advantage of this approach, as opposed to the axiomatic method, is that the primitive rules of definition admit only sound extensions to the logic, in the sense that they preserve the property of the logic's having a (standard) model. Making definitions is therefore guaranteed not to introduce inconsistency. The disadvantage is that this allows one to make definitions of only certain very restricted forms; all other kinds of definitions must be derived by formal proof from equivalent but more complex primitive ones.

The primitive basis of the HOL logic includes three rules of definition: the rule of *constant definition*, the rule of *constant specification*, and the rule of *type definition.* A constant definition is an equational axiom of the form $\vdash \mathsf{c} = \mathrm{M}$ that introduces a new constant c as an object-language abbreviation for a closed term M. The rule of constant specification allows one to introduce a new constant into the logic as an atomic name for a quantity already known to exist. By this rule of definition one may infer from a theorem of the form $\vdash \exists x.\, \mathrm{P}[x]$ a theorem $\vdash \mathrm{P}[\mathsf{c}]$, where c is a new constant symbol. This introduces c as an object-language name for an existing value x for which $\mathrm{P}[x]$ holds. The primitive rule of type definition allows one to extend the HOL logic with new type constants and type operators. Suppose that σ is a type and $\mathrm{P}{:}\sigma{\to}\mathit{bool}$ is the characteristic predicate of some useful nonempty subset of the set denoted by σ. A type definition introduces a new type expression ν to name this subset of σ.

The primitive rules just described disallow the direct use of many commonly-used principles of definition—for example, function definition by primitive recursion. The general-purpose language ML, however, provides a facility in HOL for supporting fully automatic derived rules of definition. Using ML, one can write programs that automatically carry out the proofs necessary to derive non-primitive forms of definition from equivalent primitive ones. Among the derived principles of definition supported by HOL are recursive concrete type definitions and primitive recursive function definitions over these types [4], as well as certain inductive definitions [6].

3.2 Type specifications

In addition to the three primitive rules of definition mentioned above, a *type specification* rule for the HOL logic is introduced in the latest edition of the HOL manual [4]. This is a principle of definition which allows one to extend the logic by introducing a new type expression ν satisfying a particular defining property. The type specification rule is

$$\frac{\Gamma_1 \vdash \exists x.\, \mathrm{P}\ x, \quad \Gamma_2 \vdash (\alpha \approx \mathrm{P}{:}\sigma{\to}\mathit{bool}) \supset \mathrm{Q}}{\Gamma_1 \cup \Gamma_2 \vdash \mathrm{Q}[\nu/\alpha]} \qquad \alpha \text{ not in } \sigma \text{ or } \mathrm{P},$$

where the notation '$\alpha \approx \mathrm{P}$' is an abbreviation for the assertion that there is a bijection between the values of type α and the set of values that satisfy P:

$$\exists f{:}\alpha{\to}\sigma.\, (\forall a_1\, a_2.\, (f\ a_1 = f\ a_2) \supset (a_1 = a_2)) \wedge (\forall r.\, \mathrm{P}\ r = (\exists a.\, r = f\ a)).$$

This (as yet unimplemented) principle of definition is intended to allow new types to be introduced by loose specifications of their properties. The idea is that one finds a non-empty subset P of an existing type σ to represent the values of desired type. One then proves that if the type α is in bijection with this set, then the desired property Q holds of α. If this can be proved, then it is consistent to extend the logic with a new type ν for which $\mathrm{Q}[\nu/\alpha]$ holds.

With the usual HOL rule of type definition, a new type is defined by an axiom that explicitly relates its values to those of a particular subset of an existing type. Roughly speaking, the type definition rule is

$$\frac{\vdash \exists x.\, \mathrm{P}\ x}{\vdash \nu \approx \mathrm{P}}$$

The defining theorem introduced by this rule explicitly states that the newly defined type ν is in bijection with the set of values that satisfy P. By contrast, all that is known about a type introduced by the type specification rule is that it has the properties given by the formula Q. Explicit information about representation is not part of the definition of a type, so types can be only loosely specified by this rule.

A natural application of this type specification rule is in introducing types whose specifications are given by a collection of constants with particular properties—for example, abstract data types with equational specifications. The property by which the new type is specified takes the following general form:

$$\exists v_1\ \ldots\ v_n.\mathrm{Q}[v_1, \ldots, v_n]$$

That is, one shows that there are values v_1, ..., v_n involving the new type (for example, operations over it) for which $\mathrm{Q}[v_1, \ldots, v_n]$ holds. A type having this property is then introduced by a type specification, and the constant specification rule can subsequently be used to introduce constants c_1, ..., c_n that denote these values. The result is a data type characterised by the theorem $\vdash \mathsf{Q}[\mathsf{c}_1, \ldots, \mathsf{c}_n]$ and defined in such a way that consequences of this defining property are all that can be proved about the type.

An example application of this approach is in defining a type *two*, about which all that is known is that it has *at least* two distinct elements. This type can be defined by first proving the theorems $\vdash \exists x.\, (\lambda x{:}\mathit{bool}.\mathsf{T})\ x$ and $\vdash (\alpha \approx (\lambda x{:}\mathit{bool}.\mathsf{T})) \supset \exists x{:}\alpha.\, \exists y{:}\alpha.\neg(x = y)$. One can then use the type specification rule to introduce the type *two* satisfying

$$\vdash \exists x{:}\mathit{two}.\, \exists y{:}\mathit{two}.\, \neg(x = y)$$

Finally, the constants one and two can be introduced for the two values asserted to exist by this theorem, giving the characterising theorem $\vdash \neg(\mathsf{one} = \mathsf{two})$. The result is a type about which we know only that it has at least two distinct values, namely one and two.

3.3 The problem

The problem that motivates the present proposal to extend the HOL logic is illustrated by the following attempt to use the type definition method described above. Suppose we wish to define in HOL an abstract data type of non-empty bit-vectors, specified algebraically using two unit vectors t and f and an associative concatenation operation c. Suppose further that we wish to use the methodology sketched above—that is, we wish first to prove that the desired characterizing property holds of some appropriate representing set of values and then to introduce the type with this property using a type specification. In this case, we would proceed by first proving

$$\begin{array}{l} \vdash (\alpha \approx (\lambda l{:}(bool)list.\, \neg(l = [\,]))) \ \supset \\ \quad \exists t{:}\alpha.\, \exists f{:}\alpha.\, \exists c{:}\alpha{\to}\alpha{\to}\alpha. \\ \qquad (\forall b_1\, b_2\, b_3.\, c\ b_1\ (c\ b_2\ b_3) = c\ (c\ b_1\ b_2)\ b_3) \wedge \\ \qquad (\forall x{:}\beta.\, \forall y{:}\beta.\, \forall op{:}\beta{\to}\beta{\to}\beta. \\ \qquad\quad (\forall a_1\, a_2\, a_3.\, op\ a_1\ (op\ a_2\ a_3) = op\ (op\ a_1\ a_2)\ a_3) \ \supset \\ \qquad\quad \exists! fn{:}\alpha{\to}\beta.\, (fn\ t = x) \wedge (fn\ f = y) \wedge \\ \qquad\qquad\qquad\qquad (\forall b_1\, b_2.\, fn\ (c\ b_1\ b_2) = op\ (fn\ b_1)\ (fn\ b_2))) \end{array}$$

This theorem says that if α is in bijection with the set of non-empty lists of booleans, then we have an abstract characterization of α as an initial algebra with two elements t and f and an associative concatenation operation c. In particular, there is a unique homomorphism from α to any other type β with two elements x and y and an associative operation *op*.

Given this theorem, one can use the type specification rule shown in the previous section to introduce a type constant *vect* with the following defining property.

$$\begin{array}{l} \vdash \exists t{:}vect.\, \exists f{:}vect.\, \exists c{:}vect{\to}vect{\to}vect. \\ \qquad (\forall b_1\, b_2\, b_3.\, c\ b_1\ (c\ b_2\ b_3) = c\ (c\ b_1\ b_2)\ b_3) \wedge \\ \qquad (\forall x{:}\beta.\, \forall y{:}\beta.\, \forall op{:}\beta{\to}\beta{\to}\beta. \\ \qquad\quad (\forall a_1\, a_2\, a_3.\, op\ a_1\ (op\ a_2\ a_3) = op\ (op\ a_1\ a_2)\ a_3) \ \supset \\ \qquad\quad \exists! fn{:}vect{\to}\beta.\, (fn\ t = x) \wedge (fn\ f = y) \wedge \\ \qquad\qquad\qquad\qquad (\forall b_1\, b_2.\, fn\ (c\ b_1\ b_2) = op\ (fn\ b_1)\ (fn\ b_2))) \end{array} \tag{1}$$

Having introduced the type *vect* with the above specification, it is natural to wish to use the constant specification rule to give names to the values t, f and c. This would yield the theorem

$$\begin{array}{l} \vdash (\forall b_1\, b_2\, b_3.\, \mathsf{c}\ b_1\ (\mathsf{c}\ b_2\ b_3) = \mathsf{c}\ (\mathsf{c}\ b_1\ b_2)\ b_3) \wedge \\ \quad (\forall x{:}\beta.\, \forall y{:}\beta.\, \forall op{:}\beta{\to}\beta{\to}\beta. \\ \qquad (\forall a_1\, a_2\, a_3.\, op\ a_1\ (op\ a_2\ a_3) = op\ (op\ a_1\ a_2)\ a_3) \ \supset \\ \qquad \exists! fn{:}vect{\to}\beta.\, (fn\ \mathsf{t} = x) \wedge (fn\ \mathsf{f} = y) \wedge \\ \qquad\qquad\qquad\qquad (\forall b_1\, b_2.\, fn\ (\mathsf{c}\ b_1\ b_2) = op\ (fn\ b_1)\ (fn\ b_2))) \end{array} \tag{2}$$

where t and f are constants of type *vect*, and the concatenation operator c is a constant of type $vect{\to}vect{\to}vect$. This final theorem is the desired algebraic characterization of bit-vectors. The general approach to defining other equationally-specified abstract data types would be similar.

The trouble with the construction sketched above is that the primitive rule of constant specification in HOL does not admit the direct inference of theorem (2) from theorem (1). The rule of constant specification is:

$$\frac{\vdash \exists x_1{:}\sigma_1\ \ldots\ x_n{:}\sigma_n.\, \mathrm{P}}{\vdash \mathrm{P}[\mathsf{c}_1, \ldots, \mathsf{c}_n / x_1, \ldots, x_n]} \quad \text{tyvars}(\mathrm{P}) \subseteq \text{tyvars}(\sigma_i) \text{ for } 1 \leq i \leq n.$$

The side condition on this rule requires all the type variables in the defining property P to be included in the types of the constants to be introduced. But in the defining property given by theorem (1) there is a type variable β which does not appear in the type of the existentially quantified variables t, f and c, so the constant specification rule cannot in this case be applied.

The problem illustrated by this example is closely related to the implicit quantification over type variables mentioned in the previous section. Consider a theorem of the form $\vdash \exists x{:}\sigma.\,\mathrm{P}$ where the type variable α appears in P but not in the type σ. Informally, what is meant by the theorem is 'for any type τ, we have $\vdash \exists x{:}\sigma.\,\mathrm{P}[\tau/\alpha]$'. That is, for any type τ, there is a value x such that $\mathrm{P}[\tau/\alpha]$ comes out true. In general, of course, the choice of value x that makes this term true may depend on the type τ. And if the type variable α does not appear in the type of x, then this dependence will be lost if a constant is introduced to denote this value—hence the side-condition on the rule for constant specifications. For example, in the theorem

$$\vdash \exists b.\, b = \forall x{:}\alpha.\, \forall y{:}\alpha.\, x = y$$

the value b must be either true or false, depending on whether the type α denotes a set with exactly one element or not. It is therefore unsound to admit a constant specification

$$\vdash \mathsf{b} = \forall x{:}\alpha.\, \forall y{:}\alpha.\, x = y$$

in which a fixed name b is given to this value, since then by instantiating α in two different ways one can obtain both $\vdash \mathsf{b} = \mathsf{T}$ and $\vdash \mathsf{b} = \mathsf{F}$.

Returning to the example, consider again the defining property for the type *vect* given by theorem (1). This contains a type variable β that does not appear in the types of the variables t, f and c, and so the desired constant specification cannot be made. In this particular case, however, the values t, f and c that make the specification true do not, in fact, depend on the set denoted by type variable β; there are three fixed values t, f and c that make this specification hold for *any* type β. This suggests that the defining property for *vect* should be something more like

$$\begin{array}{l}
\vdash \exists t{:}\mathit{vect}.\, \exists f{:}\mathit{vect}.\, \exists c{:}\mathit{vect}{\to}\mathit{vect}{\to}\mathit{vect}.\\
\quad (\forall b_1\, b_2\, b_3.\, c\ b_1\ (c\ b_2\ b_3) = c\ (c\ b_1\ b_2)\ b_3) \wedge\\
\quad (\forall \beta.\, \forall x{:}\beta.\, \forall y{:}\beta.\, \forall \mathit{op}{:}\beta{\to}\beta{\to}\beta.\\
\qquad (\forall a_1\, a_2\, a_3.\, \mathit{op}\ a_1\ (\mathit{op}\ a_2\ a_3) = \mathit{op}\ (\mathit{op}\ a_1\ a_2)\ a_3) \supset\\
\qquad \exists! \mathit{fn}{:}\mathit{vect}{\to}\beta.\, (\mathit{fn}\ t = x) \wedge (\mathit{fn}\ f = y) \wedge\\
\qquad\qquad\qquad (\forall b_1\, b_2.\, \mathit{fn}\ (c\ b_1\ b_2) = \mathit{op}\ (\mathit{fn}\ b_1)\ (\mathit{fn}\ b_2)))
\end{array} \tag{3}$$

where there is a universal quantification over the type variable β within the scope of the existential quantification over t, f and c.

This new theorem expresses the initiality property of the data type *vect* more accurately than theorem (1). Moreover, if the quantification over β is taken to bind the type variable β, and if the function tyvars in the constant specification rule shown above is reinterpreted as giving only the *free* type variables in a term, then it would be valid to infer from this theorem the desired constant specification

$$\begin{array}{l}
\vdash (\forall b_1\, b_2\, b_3.\, \mathsf{c}\ b_1\ (\mathsf{c}\ b_2\ b_3) = \mathsf{c}\ (\mathsf{c}\ b_1\ b_2)\ b_3) \wedge\\
\quad (\forall \beta.\, \forall x{:}\beta.\, \forall y{:}\beta.\, \forall \mathit{op}{:}\beta{\to}\beta{\to}\beta.\\
\qquad (\forall a_1\, a_2\, a_3.\, \mathit{op}\ a_1\ (\mathit{op}\ a_2\ a_3) = \mathit{op}\ (\mathit{op}\ a_1\ a_2)\ a_3) \supset\\
\qquad \exists! \mathit{fn}{:}\mathit{vect}{\to}\beta.\, (\mathit{fn}\ \mathsf{t} = x) \wedge (\mathit{fn}\ \mathsf{f} = y) \wedge\\
\qquad\qquad\qquad (\forall b_1\, b_2.\, \mathit{fn}\ (\mathsf{c}\ b_1\ b_2) = \mathit{op}\ (\mathit{fn}\ b_1)\ (\mathit{fn}\ b_2)))
\end{array}$$

This is the correct form of algebraic specification for the type *vect*.

4 Universal quantification over types

The proposal of this paper is to extend the HOL logic with universal quantification over types, so that properties like that given by theorem (3) in the previous section can be expressed directly in the logic. With this extension to HOL, the syntax of untyped terms becomes

$$\mathrm{M} \quad ::= \quad c \quad | \quad v \quad | \quad \mathrm{M\,N} \quad | \quad \lambda v.\,\mathrm{M} \quad | \quad \forall\alpha.\,\mathrm{M}$$

That is, the term language of HOL (i.e. the simply-typed λ-calculus) is extended with terms of the form $\forall\alpha.\,\mathrm{M}$, where α is a type variable and M is a (boolean) term. The intended meaning of such a term is that the proposition represented by M is true no matter what set is denoted by the type variable α.

More precisely, we take the syntax of types for the extended logic to be

$$\sigma \quad ::= \quad \mathit{bool} \quad | \quad c \quad | \quad v \quad | \quad \sigma_1 \to \sigma_2 \quad | \quad (\sigma_1, \ldots, \sigma_n)\mathit{op}$$

That is, we assume right from the start that the set of type expressions includes the special type constant *bool*. This is necessary because we wish to form quantifications $\forall\alpha.\,\mathrm{M}$ only for *boolean* terms M. The extended syntax of typed terms is then given by

$$\mathrm{M} \quad ::= \quad c_\sigma \quad | \quad v_\sigma \quad | \quad (\mathrm{M}_{\sigma_1\to\sigma_2}\,\mathrm{N}_{\sigma_1})_{\sigma_2} \quad | \quad (\lambda v_{\sigma_1}.\,\mathrm{M}_{\sigma_2})_{\sigma_1\to\sigma_2} \quad | \quad (\forall\alpha.\,\mathrm{M}_{\mathit{bool}})_{\mathit{bool}}$$

where (as usual) a constant c_σ is a well-formed term only if σ is a substitution instance of the generic type of c, and where a quantification $\forall\alpha.\,\mathrm{M}$ is well-formed only if for every occurrence of a variable v_σ in M, either the type variable α does not appear in the type σ or the occurrence of v_σ is bound in M by a textually enclosing lambda-abstraction λv_σ. In what follows, it is assumed that all terms are well-formed according to these formation rules.

A quantification $\forall\alpha.\,\mathrm{M}$ binds occurrences of the type variable α in M, so this extended syntax of terms leads to a notion of free and bound type variable, in addition to the usual notion of free and bound (term) variables. Furthermore, the standard notion of a free occurrence of a term in another term must be modified to take into account binding by type quantifiers. Following the definition given by Andrews for his system Q, we say that an occurrence of a term M_σ is *free for* the variable v_σ in a term N if it does not occur in the body of a subterm of N that has the form $\lambda v_\sigma.\,\mathrm{P}$ and if it does not occur in the body of any subterm of the form $\forall\,\alpha.\,\mathrm{P}$ where α occurs in σ. We define a free occurrence of a variable v_σ in a term N as an occurrence which is free for itself in N. In the presence of the well-formedness condition stated above for type quantifications, this notion of free variable is equivalent to the usual HOL definition: a variable v_σ is free in N if it does not appear in the body of a subterm of the form $\lambda v_\sigma.\,\mathrm{P}$.

In what follows, we assume suitably-defined notions of term and type substitution, with renaming of bound term and type variables as appropriate to avoid variable capture. In particular, we assume for type substitution that if $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ are types and $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ are distinct type variables, then the metalinguistic notation $\mathrm{M}[\vec{\sigma}/\vec{\alpha}]$ stands for the result of simultaneously substituting σ_i for α_i for $1 \le i \le n$ at every free occurrence of α_i in the term M, with the condition that no type variable in any σ_i and no free term variable in M becomes bound in the result of the substitution. In practice, an implementation will rename type and term variables in order to satisfy this condition. For example, we have the substitutions

$$(\forall\beta.\,(x{:}\alpha = x{:}\alpha))[\beta/\alpha] = \forall\beta'.\,(x{:}\beta = x{:}\beta)$$

and

$$(\forall x{:}\alpha.\, x{:}bool)[bool/\alpha] = \forall x'{:}bool.\, x{:}bool$$

Similar renamings are also carried out in the current HOL implementation; see [4].

4.1 Semantics

The HOL manual [4] contains a set-theoretic semantics of the HOL logic due to Andy Pitts. In this section, we very briefly sketch this semantics and show how it can be extended to cover the quantification over type variables described above.

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. Among other things, the universe is closed under the function space operation and contains the distinguished two-element set $2 = \{1, 0\}$ which is the meaning of the boolean type *bool*.

The semantics of types is given relative to a model $\mathcal{M}$ which assigns to each type constant an element of $\mathcal{U}$ and to each n-ary type operator a function $\mathcal{U}^n \to \mathcal{U}$. The notion of a *type-in-context* is also used in defining the semantics of types. A *type context* $\vec{\alpha}$ is just a finite list of distinct type variables, and a type-in-context $\vec{\alpha}.\sigma$ is a type σ together with a type context $\vec{\alpha}$ which contains (at least) all the type variables in σ. The meaning of a type in context $\vec{\alpha}.\sigma$, where the context $\vec{\alpha}$ is of length n, is then given by a function

$$[\![\vec{\alpha}.\sigma]\!]_{\mathcal{M}} : \mathcal{U}^n \to \mathcal{U}$$

which is defined so that for any assignment of sets $\vec{X} = (X_1, \ldots, X_n) \in \mathcal{U}^n$ to the type variables in $\vec{\alpha}$ (and hence to the type variables in σ), the element $[\![\vec{\alpha}.\sigma]\!]_{\mathcal{M}}(\vec{X})$ of $\mathcal{U}$ is the corresponding set denoted by σ. The details of the definition of $[\![\_]\!]_{\mathcal{M}}$, which is done by induction on the structure of types in the obvious way, can be found in [4].

The notion of a context is also employed in defining the meaning of terms. A *term-in-context* is written '$\vec{\alpha},\vec{x}.$M' and consists of a term M together with a type context $\vec{\alpha}$ and a finite list of variables $\vec{x}$ called a *variable context*. The variable context $\vec{x}$ of a term-in-context $\vec{\alpha},\vec{x}.$M contains all the variables that occur free in M, and the type context $\vec{\alpha}$ contains all the type variables that occur in $\vec{x}$ and M. It is evident that this notion of terms in context is easily adapted for the extended term language described above; we simply make the condition that the type context for a term-in-context $\vec{\alpha},\vec{x}.$M contains all the type variables that appear in $\vec{x}$ or appear *free* in M. Following [4], we assume that $\vec{x}$ contains no variable that appears bound in M—bound term variables may always be renamed to satisfy this condition. Likewise, we assume that the type context $\vec{\alpha}$ contains no type variables that are bound in M. This assumption is valid because the well-formedness condition on terms of the form $\forall\alpha.\,$M ensures that M contains no *free* variable v_σ where α occurs in σ. Hence bound type variables can always be renamed so as to be distinct from all type variables in $\vec{\alpha}$.

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

$$\prod_{\vec{X} \in \mathcal{U}^n} [\![\vec{\alpha}.\sigma]\!]_{\mathcal{M}}(\vec{X})$$

where n is the length of the type context $\vec{\alpha}$. For a given model $\mathcal{M}$, the meaning of a term-in-context $\vec{\alpha},\vec{x}.\mathrm{M}$, where $\vec{\alpha}$ has length n, $\vec{x}$ has length m, and M has type τ, is given by a function $[\![\_]\!]_{\mathcal{M}}$ defined by induction on terms such that

$$[\![\vec{\alpha},\vec{x}.\mathrm{M}]\!]_{\mathcal{M}} \in \prod_{\vec{X}\in\mathcal{U}^n} \left(\prod_{j=1}^{m}[\![\vec{\alpha}.\sigma_j]\!]_{\mathcal{M}}(\vec{X})\right) \to [\![\vec{\alpha}.\tau]\!]_{\mathcal{M}}(\vec{X}),$$

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

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

to the type variables in $\vec{\alpha}$ (and hence to the free type variables in M) and an assignment of elements

$$\vec{y} = (y_1, \ldots, y_m) \in [\![\vec{\alpha}.\sigma_1]\!]_{\mathcal{M}}(\vec{X}) \times \cdots \times [\![\vec{\alpha}.\sigma_m]\!]_{\mathcal{M}}(\vec{X})$$

to the variables in $\vec{x}$ (and hence to the variables that occur free in the term M), the result of $[\![\vec{\alpha},\vec{x}.\mathrm{M}]\!]_{\mathcal{M}}(\vec{X})(\vec{y})$ will be an appropriate element of the set $[\![\vec{\alpha}.\tau]\!]_{\mathcal{M}}(\vec{X})$ denoted by the type of M. Again, full details of the definition of $[\![\_]\!]_{\mathcal{M}}$ can be found in [4].

A sequent with hypotheses $\Gamma = \{\mathrm{P}_1, \ldots, \mathrm{P}_p\}$ and conclusion P is *satisfied* by a model $\mathcal{M}$ if any assignment of values to free variables that makes all the hypotheses true in $\mathcal{M}$ also makes the conclusion true in $\mathcal{M}$. In particular, $\mathcal{M}$ satisfies the sequent if for all $\vec{X} \in \mathcal{U}^n$ and for all $\vec{y} \in [\![\vec{\alpha}.\sigma_1]\!]_{\mathcal{M}}(\vec{X}) \times \cdots \times [\![\vec{\alpha}.\sigma_m]\!]_{\mathcal{M}}(\vec{X})$,

$$[\![\vec{\alpha},\vec{x}.\mathrm{P}_1]\!]_{\mathcal{M}}(\vec{X})(\vec{y}) = 1, \quad \ldots, \quad [\![\vec{\alpha},\vec{x}.\mathrm{P}_p]\!]_{\mathcal{M}}(\vec{X})(\vec{y}) = 1$$

imply that

$$[\![\vec{\alpha},\vec{x}.\mathrm{P}]\!]_{\mathcal{M}}(\vec{X})(\vec{y}) = 1,$$

where $\vec{\alpha}, \vec{x}$ is any valid context for each of P, P_1, ..., P_p with $\vec{\alpha}$ of length n, $\vec{x} = x_1, \ldots, x_m$, and σ_i the type of the corresponding variable x_i. We write $\Gamma \models_{\mathcal{M}} \mathrm{P}$ to mean that $\mathcal{M}$ satisfies the sequent with hypotheses Γ and conclusion P.

It is completely straightforward to extend the definition of $[\![\_]\!]_{\mathcal{M}}$ sketched above, and hence the notion of satisfaction, to cover type quantifications $\forall\alpha.\,\mathrm{M}$. Suppose $\vec{\alpha},\vec{x}.\forall\alpha.\,\mathrm{M}$ is a term-in-context, with $\vec{\alpha}$ of length n, and $\vec{x} = x_1, \ldots, x_m$ with σ_i the type of x_i. Then, since the term M has type *bool* we have

$$[\![\vec{\alpha},\alpha,\vec{x}.\mathrm{M}]\!]_{\mathcal{M}} \in \prod_{\vec{X}\in\mathcal{U}^{n+1}} \left(\prod_{j=1}^{m}[\![\vec{\alpha},\alpha.\sigma_j]\!]_{\mathcal{M}}(\vec{X})\right) \to 2$$

Note that α is distinct from all the type variables in $\vec{\alpha}$, by the assumption that this type context contains no type variable bound in $\forall\alpha.\,\mathrm{M}$. This ensures that $\vec{\alpha},\alpha,\vec{x}.\mathrm{M}$ is in fact a well-formed term-in-context. Now define

$$[\![\vec{\alpha},\vec{x}.\forall\alpha.\,\mathrm{M}]\!]_{\mathcal{M}} \in \prod_{\vec{X}\in\mathcal{U}^n} \left(\prod_{j=1}^{m}[\![\vec{\alpha}.\sigma_j]\!]_{\mathcal{M}}(\vec{X})\right) \to 2$$

by taking $[\![\vec{\alpha},\vec{x}.\forall\alpha.\,\mathrm{M}]\!]_{\mathcal{M}}(\vec{X})(\vec{y})$ to be the value 1 if and only if the function in $\mathcal{U} \to 2$ given by $X \mapsto [\![\vec{\alpha},\alpha,\vec{x}.\mathrm{M}]\!](\vec{X}, X)(\vec{y})$ yields 1 for all $X \in \mathcal{U}$. That is, '$\forall\alpha.\,\mathrm{M}$' is true iff M is true for any assignment of a set $X \in \mathcal{U}$ to α.

4.2 Primitive inference rules

Having added propositions $\forall\alpha.\,\mathrm{M}$ as primitive terms to HOL, the next step is to extend the logic with new primitive inference rules for reasoning with them. The required rules are the introduction and elimination rules for universal quantification over types shown below.

$$\forall\text{-I:}\quad \frac{\Gamma \vdash \mathrm{P}{:}\mathit{bool}}{\Gamma \vdash \forall\alpha.\,\mathrm{P}} \quad \alpha \text{ not free in } \Gamma.$$

$$\forall\text{-E:}\quad \frac{\Gamma \vdash \forall\alpha.\,\mathrm{P}}{\Gamma \vdash \mathrm{P}[\sigma/\alpha]}$$

Note that in both rules, it is assumed that $\forall\alpha.\,\mathrm{P}$ is a well formed term—that is, that P contains no free variable v_σ with α occurring in σ.

In what follows, we sketch the proof that the elimination rule ∀-E is sound. That is, we show that if a model satisfies the hypothesis of the rule, then it also satisfies the conclusion. The proof depends on the series of lemmas given below.

Lemma 4.1 *Suppose α does not occur in $\vec{\alpha}, \vec{x}$ and α does not occur in σ, and suppose that $\forall\alpha.\,\mathrm{P}$ is well-formed. Then if $\vec{\alpha}, \vec{x}$ is a valid context for each of $\mathrm{P}[\sigma/\alpha]$, $\mathrm{P}_1, \ldots, \mathrm{P}_p$, then $\vec{\alpha}, \vec{x}$ is also a valid context for each of $\forall\alpha.\,\mathrm{P}$, $\mathrm{P}_1, \ldots, \mathrm{P}_p$.*

Proof. Obvious, by the definitions of well-formedness and valid term contexts. Note that we assume that no bound type variables in P have been renamed in $\mathrm{P}[\sigma/\alpha]$. One can always contrive to make this the case by doing appropriate renamings beforehand. □

Lemma 4.2 *If $\vec{\alpha}, \alpha.\sigma$ is a type-in-context where α does not occur in σ, then*

$$[\![\vec{\alpha}, \alpha.\sigma]\!]_{\mathcal{M}}(\vec{X}, X) = [\![\vec{\alpha}.\sigma]\!]_{\mathcal{M}}(\vec{X})$$

for all $X \in \mathcal{U}$ and $\vec{X} \in \mathcal{U}^n$ where n is the length of $\vec{\alpha}$.

Proof. By structural induction on σ. □

Lemma 4.3 *Suppose $\forall\alpha.\,\mathrm{P}$ is a well-formed term (and hence P contains no free variable v_τ where α occurs in τ). If $\vec{\alpha}, \vec{x}.\mathrm{P}[\sigma/\alpha]$ is a term-in-context where α does not occur in $\vec{\alpha}, \vec{x}$ and α does not occur in σ, then for all $\vec{X} \in \mathcal{U}^n$, where n is the length of $\vec{\alpha}$*

$$[\![\vec{\alpha}, \vec{x}.\mathrm{P}[\sigma/\alpha]]\!]_{\mathcal{M}}(\vec{X}) = [\![\vec{\alpha}, \alpha, \vec{x}.\mathrm{P}]\!]_{\mathcal{M}}(\vec{X}, [\![\vec{\alpha}.\sigma]\!](\vec{X}))$$

Proof. By structural induction on P, using lemma 4.2 to establish that the left- and right-hand sides are functions with the same domain. □

Theorem 4.4 *The ∀-E rule is sound. That is, for any model $\mathcal{M}$ and type σ, if $\Gamma \models_{\mathcal{M}} \forall\alpha.\,\mathrm{P}$, then $\Gamma \models_{\mathcal{M}} \mathrm{P}[\sigma/\alpha]$.*

Proof. We may suppose that α appears nowhere in Γ and that σ does not contain α, as the bound type variable α can be renamed if necessary without changing the semantics. Now suppose that $\Gamma \models_{\mathcal{M}} \forall\alpha.\,\mathrm{P}$ with $\Gamma = \{\mathrm{P}_1, \ldots, \mathrm{P}_p\}$. Then for any $\vec{\alpha}, \vec{x}$ which is a context of each of the terms $\forall\alpha.\,\mathrm{P}$, P_1, …, P_p where the length of $\vec{\alpha}$ is n, $\vec{x} = x_1, \ldots, x_m$, and x_i has type σ_i, we have that for all $\vec{X} \in \mathcal{U}^n$ and $\vec{y} \in [\![\vec{\alpha}.\sigma_1]\!]_{\mathcal{M}}(\vec{X}) \times \cdots \times [\![\vec{\alpha}.\sigma_m]\!]_{\mathcal{M}}(\vec{X})$, if

$$[\![\vec{\alpha}, \vec{x}.\mathrm{P}_k]\!]_{\mathcal{M}}(\vec{X})(\vec{y}) = 1 \quad \text{for } 1 \le k \le p$$

then $[\![\vec{\alpha}, \vec{x}.\forall\alpha.\,\mathrm{P}]\!]_{\mathcal{M}}(\vec{X})(\vec{y}) = 1$, which by the semantics of $\forall\alpha.\,\mathrm{P}$ given above means that for all $X \in \mathcal{U}$ we have

$$[\![\vec{\alpha}, \alpha, \vec{x}.\mathrm{P}]\!]_{\mathcal{M}}(\vec{X}, X)(\vec{y}) = 1.$$

Now, suppose that $\vec{\alpha}', \vec{x}'$ is a context for each of $\mathrm{P}[\sigma/\alpha]$, P_1, …, P_p. We may assume that $\vec{\alpha}', \vec{x}'$ does not contain α, since α does not appear in any of $\mathrm{P}[\sigma/\alpha]$, P_1, …, P_p. Let n' be the length of $\vec{\alpha}'$, $\vec{x}' = x'_1, \ldots, x'_{m'}$, and σ'_i be the type of the corresponding variable x'_i. Assume that for some $\vec{X}' \in \mathcal{U}^{n'}$ and $\vec{y}' \in [\![\vec{\alpha}.\sigma'_1]\!]_{\mathcal{M}}(\vec{X}') \times \cdots \times [\![\vec{\alpha}'.\sigma'_{m'}]\!]_{\mathcal{M}}(\vec{X}')$ we have for the hypotheses that

$$[\![\vec{\alpha}', \vec{x}'.\mathrm{P}_k]\!]_{\mathcal{M}}(\vec{X}')(\vec{y}') = 1 \quad \text{for } 1 \le k \le p.$$

By lemma 4.1 we have that $\vec{\alpha}', \vec{x}'$ is a valid context for each of $\forall\alpha.\,\mathrm{P}$, P_1, …, P_p, so the above assumption implies that for all $X \in \mathcal{U}$

$$[\![\vec{\alpha}', \alpha, \vec{x}'.\mathrm{P}]\!]_{\mathcal{M}}(\vec{X}', X)(\vec{y}') = 1 \tag{4}$$

We then have by lemma 4.3 that

$$[\![\vec{\alpha}', \vec{x}'.\mathrm{P}[\sigma/\alpha]]\!]_{\mathcal{M}}(\vec{X}')(\vec{y}') = [\![\vec{\alpha}', \alpha, \vec{x}'.\mathrm{P}]\!]_{\mathcal{M}}(\vec{X}', [\![\vec{\alpha}'.\sigma]\!](\vec{X}'))(\vec{y}')$$

and hence by (4) that $[\![\vec{\alpha}', \vec{x}'.\mathrm{P}[\sigma/\alpha]]\!]_{\mathcal{M}}(\vec{X}')(\vec{y}') = 1$, as required. □

The proof of soundness for the introduction rule ∀-I is equally straightforward and so is omitted here.

4.3 Derived inference rules

Various higher-level inference rules for universal quantification over types are derivable from the primitive introduction and elimination rules given in the previous section. We give the derivations of some of the more important rules below. The standard HOL names are used for the inference steps shown in these derivations.

The significance of the following congruence rule is that it is needed to extend the implementation of HOL's conversion-based rewriting tools; see [4] for the details.

Theorem 4.5 *Universal quantification over types satisfies the following congruence rule*

$$\forall\text{-EQ:}\quad \frac{\Gamma \vdash (\mathrm{M}{:}\mathit{bool}) = (\mathrm{N}{:}\mathit{bool})}{\Gamma \vdash (\forall\alpha.\,\mathrm{M}) = (\forall\alpha.\,\mathrm{N})} \quad \alpha \text{ not free in } \Gamma.$$

Proof. By the following derivation.

1. $\Gamma, \forall\alpha.\,\mathrm{M} \vdash \forall\alpha.\,\mathrm{M}$ [`ASSUME`, `ADD_ASSUM`]

2. $\Gamma, \forall\alpha.\,\mathrm{M} \vdash \mathrm{M}$ [$\forall$-E rule: 1]
3. $\Gamma, \forall\alpha.\,\mathrm{M} \vdash \mathrm{N}$ [`EQ_MP`: hypothesis, 2]
4. $\Gamma, \forall\alpha.\,\mathrm{M} \vdash \forall\alpha.\,\mathrm{N}$ [$\forall$-I rule: 3]
5. $\Gamma \vdash \forall\alpha.\,\mathrm{M} \supset \forall\alpha.\,\mathrm{N}$ [`DISCH`: 4]
6. $\Gamma \vdash \forall\alpha.\,\mathrm{N} \supset \forall\alpha.\,\mathrm{M}$ [similar to steps 1–5]
7. $\Gamma \vdash \forall\alpha.\,\mathrm{M} = \forall\alpha.\,\mathrm{N}$ [`EQ_IMP_RULE`: 5, 6]

Note that the side condition on the $\forall$-EQ rule is necessary for the steps in this proof that use the introduction rule $\forall$-I to be valid. $\square$

The following theorem shows that the existing HOL type instantiation rule, described above in section 2, can be derived in the extended logic.

Theorem 4.6 *The following type instantiation rule is derivable.*

$$\forall\text{-INST:} \quad \frac{\Gamma \vdash \mathrm{P}}{\Gamma \vdash \mathrm{P}[\sigma/\alpha]} \quad \alpha \text{ not free in } \Gamma.$$

where there is the additional side condition that no two distinct term variables in P *become identified as a result of doing the substitution (see section 2).*

Proof. Let $x_1, \ldots, x_n$ be all the free variables in P whose types contain α.

1. $\Gamma \vdash \mathrm{P}$ [hypothesis]
2. $\Gamma \vdash \forall x_1.\ \cdots\ \forall x_n.\,\mathrm{P}$ [`GENL`: 1]
3. $\Gamma \vdash \forall\alpha.\forall x_1.\ \cdots\ \forall x_n.\,\mathrm{P}$ [$\forall$-I rule: 2]
4. $\Gamma \vdash (\forall x_1.\ \cdots\ \forall x_n.\,\mathrm{P})[\sigma/\alpha]$ [$\forall$-E rule: 3]
5. $\Gamma \vdash \forall x_1[\sigma/\alpha].\ \cdots\ \forall x_n[\sigma/\alpha].\,(\mathrm{P}[\sigma/\alpha])$ [defn. of type substitution: 4]
6. $\Gamma \vdash \mathrm{P}[\sigma/\alpha]$ [`SPECL`: 5]

Note that step 2 is valid because the side condition that α is not free in Γ ensures that none of the variables $x_1, \ldots, x_n$, each of whose types contains α, is free in Γ. Step 3 is valid because the generalizations in step 2 ensure that '$\forall\alpha.\forall x_1.\ \cdots\ \forall x_n.\,\mathrm{P}$' will be well-formed. Step 5 is valid because the additional side condition mentioned above ensures that no x_i need be renamed to avoid capture by type substitution of other variables free in P. $\square$

It is straightforward use this $\forall$-INST rule to derive the full rule for simultaneous type instantiation shown in section 2. This means that a primitive type instantiation rule, as present in the existing HOL logic, is redundant in the extended system—it can instead be a derived rule. The rule shown in section 2 for instantiation throughout a sequent is also derivable.

Theorem 4.7 *The following alpha-conversion rule holds.*

$$\forall\text{-ACONV:} \quad \frac{}{\vdash (\forall\alpha.\,\mathrm{M}) = (\forall\beta.\,\mathrm{M}[\beta/\alpha])} \quad \beta \text{ not free in } \forall\alpha.\,\mathrm{M}.$$

Proof. By the following derivation.

1. $\forall\alpha.\,\mathrm{M} \vdash \forall\alpha.\,\mathrm{M}$ [ASSUME]
2. $\forall\alpha.\,\mathrm{M} \vdash \mathrm{M}[\beta/\alpha]$ [$\forall$-E rule: 1]
3. $\forall\alpha.\,\mathrm{M} \vdash \forall\beta.\,\mathrm{M}[\beta/\alpha]$ [$\forall$-I rule: 2]
4. $\vdash \forall\alpha.\,\mathrm{M} \supset \forall\beta.\,\mathrm{M}[\beta/\alpha]$ [DISCH: 3]
5. $\vdash (\forall\beta.\,\mathrm{M}[\beta/\alpha]) \supset \forall\alpha.\,\mathrm{M}$ [similar to steps 1–4]
6. $\vdash \forall\alpha.\,\mathrm{M} = (\forall\beta.\,\mathrm{M}[\beta/\alpha])$ [EQ_IMP_RULE: 4, 5]

The side condition that β is not free in $\forall\alpha.\,\mathrm{M}$ is necessary for step 3 to be valid. It also ensures that $\mathrm{M}[\beta/\alpha][\alpha/\beta] = \mathrm{M}$, which is used in step 5. □

5 Existential quantification over types

Adding universal quantification over types to the HOL logic immediately suggests that existential quantification is possible as well. The simplest approach is to make a term of the form '$\exists\alpha.\,\mathrm{P}$' an abbreviation for '$\neg\forall\alpha.\,\neg\mathrm{P}$'. This cannot, of course, be done as an object-language abbreviation using the existing HOL definitional mechanisms. It could, however, be implemented in HOL as a parser-supported metalinguistic abbreviation. Alternatively, the term language could be further extended to

$$\mathrm{M} \quad ::= \quad c \quad | \quad v \quad | \quad \mathrm{M\,N} \quad | \quad \lambda v.\,\mathrm{M} \quad | \quad \forall\alpha.\,\mathrm{M} \quad | \quad \exists\alpha.\,\mathrm{M}$$

with formation and typing rules similar to those given in the previous section for universal quantification. Meaning could then be given to terms of the form '$\exists\alpha.\,\mathrm{M}$' by the addition of the following primitive axiom-scheme:

$$\exists\text{-DEF:} \quad \frac{}{\vdash (\exists\alpha.\,\mathrm{P}) = \forall Q.\,(\forall\alpha.\,\mathrm{P} \supset Q) \supset Q}$$

This is the only primitive rule required for existential quantification over types. Appropriate introduction and elimination rules for $\exists\alpha.\,\mathrm{P}$ are easily derived using it.

Theorem 5.1 *The following introduction and elimination rules are derivable.*

$$\exists\text{-I:} \quad \frac{\Gamma \vdash \mathrm{P}[\sigma/\alpha]{:}bool}{\Gamma \vdash \exists\alpha.\,\mathrm{P}}$$

$$\exists\text{-E:} \quad \frac{\Gamma \vdash \exists\alpha.\,\mathrm{P}, \quad \Gamma, \mathrm{P}[\beta/\alpha] \vdash \mathrm{Q}}{\Gamma \vdash \mathrm{Q}} \quad \beta \textit{ not free in } \Gamma \textit{ or } \mathrm{Q}.$$

Proof. For the introduction rule, the derivation is

1. $\Gamma \vdash \mathrm{P}[\sigma/\alpha]$ [hypothesis]
2. $\forall\alpha.\,\mathrm{P} \supset Q \vdash \forall\alpha.\,\mathrm{P} \supset Q$ [ASSUME]
3. $\forall\alpha.\,\mathrm{P} \supset Q \vdash \mathrm{P}[\sigma/\alpha] \supset Q$ [$\forall$-E rule: 2]
4. $\Gamma, \forall\alpha.\,\mathrm{P} \supset Q \vdash Q$ [MP: 3, 1]
5. $\Gamma \vdash \forall Q.\,(\forall\alpha.\,\mathrm{P} \supset Q) \supset Q$ [DISCH, GEN: 4]

6. $\Gamma \vdash \exists\alpha.\,\mathrm{P}$ [$\exists$-DEF]

For the elimination rule, the derivation is

1. $\Gamma \vdash \forall Q.\,(\forall\alpha.\,\mathrm{P} \supset Q) \supset Q$ [$\exists$-DEF: hypothesis]
2. $\Gamma \vdash (\forall\alpha.\,\mathrm{P} \supset \mathrm{Q}) \supset \mathrm{Q}$ [SPEC: 1]
3. $\Gamma \vdash \mathrm{P}[\beta/\alpha] \supset \mathrm{Q}$ [DISCH: hypothesis]
4. $\Gamma \vdash \forall\beta.\,\mathrm{P}[\beta/\alpha] \supset \mathrm{Q}$ [$\forall$-I rule: 3]
5. $\Gamma \vdash \forall\alpha.\,\mathrm{P} \supset \mathrm{Q}$ [$\forall$-E rule, $\forall$-I rule: 4]
6. $\Gamma \vdash \mathrm{Q}$ [MP: 2, 5]

The side condition that β is not free in Γ or Q is needed for the validity of steps 4 and 5 in the second derivation. □

Theorem 5.2 *The following classical definition of $\exists\alpha.\,\mathrm{P}$ holds.*

$$\exists\text{-CONV:}\quad \frac{}{\vdash (\exists\alpha.\,\mathrm{P}) = \neg\forall\alpha.\neg\mathrm{P}}$$

Proof. By the following derivation.

1. $\exists\alpha.\,\mathrm{P} \vdash \forall Q.\,(\forall\alpha.\,\mathrm{P} \supset Q) \supset Q$ [ASSUME, $\exists$-DEF]
2. $\exists\alpha.\,\mathrm{P} \vdash (\forall\alpha.\,\mathrm{P} \supset \mathsf{F}) \supset \mathsf{F}$ [SPEC: 1]
3. $\exists\alpha.\,\mathrm{P} \vdash \neg\forall\alpha.\,\neg\mathrm{P}$ [NOT_INTRO: 2]
4. $\neg\exists\alpha.\,\mathrm{P} \vdash \neg\exists\alpha.\,\mathrm{P}$ [ASSUME]
5. $\mathrm{P} \vdash \exists\alpha.\,\mathrm{P}$ [ASSUME, $\exists$-I rule]
6. $\neg\exists\alpha.\,\mathrm{P} \vdash \neg\mathrm{P}$ [MP, CCONTR: 4, 5]
7. $\neg\exists\alpha.\,\mathrm{P} \vdash \forall\alpha.\,\neg\mathrm{P}$ [$\forall$-I rule: 6]
8. $\neg\forall\alpha.\,\neg\mathrm{P} \vdash \exists\alpha.\,\mathrm{P}$ [CONTRAPOS: 7]
9. $\vdash \exists\alpha.\,\mathrm{P} = \neg\forall\alpha.\,\neg\mathrm{P}$ [EQ_IMP_RULE: 3, 8]

Note that the fact that the HOL logic is classical is used in this proof □

The derivations of congruence and alpha-conversion rules for existential quantification over types are straightforward and will therefore not be given here.

We conclude this section with some preliminary discussion of the idea that extending HOL with existential quantification over types may make it possible to simplify the type specification rule discussed above in section 3.2. In particular, by using existential type quantification the type specification rule may be expressed as

$$\frac{\Gamma \vdash \exists\alpha.\,\mathrm{Q}}{\Gamma \vdash \mathrm{Q}[\nu/\alpha]}$$

That is, if one can prove that there exists a type α with the property given by Q, then it is consistent to extend the logic with a new type expression ν for which $\mathrm{Q}[\nu/\alpha]$ holds. This new formulation of the type specification rule seems somewhat more direct than the one shown in section 3.2, as it does not need the notion of a bijection between sets of values. The non-emptyness condition of the old type specification rule is also not needed; its role in ensuring soundness is subsumed by the hypothesis $\vdash \exists\alpha.\,\mathrm{Q}$ of the new rule.

With this new formulation, however, the question arises as to whether one can always prove the hypothesis $\Gamma \vdash \exists\alpha.\,\mathrm{Q}$ required for any desired application of the rule. The existential introduction rule ∃-I may be used to prove propositions of this form, but this rule requires a witness for the type α asserted to exist. Moreover, this witness must already be expressible in the type language of HOL, so the type specification rule is of only limited use. It may, therefore, be necessary to add the following additional primitive rule

$$\frac{\Gamma \vdash \exists x:\sigma.\,\mathrm{P}\;x}{\Gamma \vdash \exists\alpha.\,\alpha \approx \mathrm{P}}$$

in order to make the new type specification rule shown above actually useful. This rule states that for any non-empty set of values, there exists a type whose elements are in bijection with this set. There need not, of course, be an actual type expression that denotes this set.

6 Implementation

The extensions proposed in the preceding sections have not yet been implemented. Adding terms of the forms $\forall\alpha.\,\mathrm{P}$ and $\exists\alpha.\,\mathrm{P}$ is an extension of the primitive term language of the HOL logic, so an implementation based on the existing HOL88 or HOL90 [7] systems would require fairly extensive modifications. The type checker would have to be extended, as would the parser and pretty-printer. (Type checking and type inference for the extended system are still decidable.) The definitions of many of the syntax functions would have to be modified—for example the functions dealing with alpha-conversion, term substitution, type substitution, and free and bound variables. All this should be fairly straightforward, though reimplementing operations like substitution is likely to be error-prone. It is hoped that a prototype system can be be built in the near future, most likely based on HOL90, so that practical experiments can be carried out.

Acknowledgements

I am grateful to Andy Pitts for discussions about extending the set-theoretic semantics of HOL. I have found Andrews' book [1] invaluable in working out many of the technical details of the extension to HOL proposed here. The anonymous referees made valuable suggestions for improvements to both the form and content of this paper. This work was carried out at the University of Cambridge under SERC grant number GR/F 36675.

References

[1] P. B. Andrews, *A Transfinite Type Theory with Type Variables*, Studies in Logic and the Foundations of Mathematics series (North-Holland, 1965).

[2] A. Church, 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, vol. 5 (1940), pp. 56–68.

[3] J.-Y. Girard, 'The System F of Variable Types, Fifteen Years Later', *Theoretical Computer Science*, vol. 45 (1986), pp. 159–192.

[4] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A theorem proving environment for higher order logic* (Cambridge University Press, 1993).

[5] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, vol. 78 (Springer-Verlag, 1979).

[6] T. Melham, 'A Package for Inductive Relation Definitions in HOL', in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 350–357.

[7] K. Slind, 'An Implementation of Higher Order Logic', Research Report 91/419/03, Department of Computer Science, University of Calgary (January 1991).