

HOL2P - A System of Classical Higher Order Logic with Second Order Polymorphism

Norbert Völker

University of Essex, England
<http://cswww.essex.ac.uk/staff/norbert>

Abstract. This paper introduces the logical system HOL2P that extends classical higher order logic (HOL) with type operator variables and universal types. HOL2P has explicit term operations for type abstraction and type application. The formation of type application terms $t [T]$ is restricted to small types T that do not contain any universal types. This constraint ensures the existence of a set-theoretic model and thus consistency.

The expressiveness of HOL2P allows category-theoretic concepts such as natural transformations and initial algebras to be applied at the level of polymorphic HOL functions. The parameterisation of terms with type operators adds genericity to theorems. Type variable quantification can also be expressed.

A prototype of HOL2P has been implemented on top of HOL-Light. Type inference is semi-automatic, and some type annotations are necessary. Reasoning is supported by appropriate tactics. The implementation has been used to check some sample derivations.

1 Introduction

Classical higher order logic (HOL) is one of the most successful logics in mechanised theorem proving. Theorem proving assistants based on it include the original HOL system [7], and its various descendants such as HOL4, Proof-Power and HOL-Light [9]. In these systems, the logic is coupled with a simply-typed polymorphic λ -calculus, that was extended with type classes in case of Isabelle/HOL [15].

The design of HOL theorem provers has been influenced by functional programming languages and their type systems. In the latter domain, there has been a lot of research into systems that go beyond simply-typed polymorphism, see for example extensions of Haskell in the Glasgow Haskell Compiler (GHC) [12]. An important driving force for these developments has been the wish to support generic programming [2,1].

The research described in this paper is an experiment in extending higher order logic beyond simply-typed polymorphism. The aim is to support more generic theorems that can be instantiated to particular polymorphic functions and type constructors. The inspiration comes from the way category-theoretic notions [13] have been applied in generic programming. In addition, the system should allow quantification over type variables as introduced by T. Melham [14].

2 A Recap of Higher Order Logic

It is assumed that the reader has a working understanding of higher-order logic. The following section provides a brief recap as a basis for discussing the HOL2P extensions. The details of the presentation follow HOL-Light.

The HOL syntax is that of a simply-typed polymorphic lambda calculus. Every term belongs to exactly one type. HOL types are either type variables, or type constructor applications:

$$\begin{array}{ll}
 T ::= \alpha & \text{type variable} \\
 | (T_1, \dots, T_n) \tau & \text{type constructor application } (n \geq 0)
 \end{array}$$

Type constructors are identified by their name and have a fixed arity. Primitive HOL type constructors are the 0-ary *bool*, a type of individuals *ind* and the binary function type operator *fun*. The latter is usually written in infix form using the “ \rightarrow ” symbol. New type constructors can be defined via a bijection to a non-empty subset of an existing type. In particular, the type *num* of natural numbers is defined as isomorphic to a subset of *ind*.

Types containing type variables are called polymorphic. Type substitution replaces type variables in a type with other types. The type resulting from a substitution is also called an instance of the original type.

HOL terms are either variables, constants, applications or abstractions.

$$\begin{array}{ll}
 t ::= v : T & \text{variable} \\
 | c : T & \text{constant} \\
 | t t & \text{application} \\
 | \lambda(v : T). t & \text{abstraction}
 \end{array}$$

Constants are uniquely identified by their name. Polymorphic constants can have differently typed instances, and each constant occurrence in a term is annotated with its type. Variables are identified by their name and HOL type. Variables with the same name but a different HOL type are unrelated. In particular, a λ -abstraction with a bound variable *v* does not bind variables in the abstraction body with the same name as *v* but a different type.

An explicit specification of all types in a term would be extremely cumbersome. Like in functional languages, HOL gets around this problem by an automated type inference process that starts off with an untyped HOL term and adds types to the constants and variables occurring in the term.

In HOL-Light, the only primitive constants are equality and Hilbert’s choice operator. New constants can be introduced as abbreviations for expressions that are build up from existing constants.

Term instantiation replaces variables in a term with other terms. Unlike type variables, term variables can be bound, so instantiation only applies to the free variables.

HOL theorems take the form of sequents ($n \geq 0$):

$$P_1, \dots, P_n \vdash P$$

where the hypotheses P_i as well as the conclusion P are terms of type *bool*. Theorems can be generated by various inference rules. In HOL-Light, there are ten primitive inference rules. The following rules are of most interest here:

- Rule **ABS** says that in order to show equality of two λ -abstractions with identical bound variables, it is sufficient to show the equality of the abstraction bodies. The rule has the side condition that the bound variable x does not occur free in any of the assumptions Γ :

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash \lambda x. s = \lambda x. t} \quad (\text{ABS})$$

- Rule **MK_COMB** says that in order to show equality of two applications, it is sufficient to show equality of the operators and the operands.

$$\frac{\Gamma_1 \vdash f = g \quad \Gamma_2 \vdash x = y}{\Gamma_1 \cup \Gamma_2 \vdash f x = g y} \quad (\text{MK_COMB})$$

- Rule **BETA** is the trivial case of β -conversion from typed λ -calculus. The general case can be derived from this by applying the instantiation rule **INST** which is described later.

$$\frac{}{\Gamma \vdash (\lambda x. t)x = t} \quad (\text{BETA})$$

- Rule **INST_TYPE** describes the fact that type variables are schematic, which means that the substitution of types variables in a theorem results in a new theorem. Note that instantiation applies to both hypotheses and the conclusion.

$$\frac{\Gamma[\alpha_1, \dots, \alpha_n] \vdash p[\alpha_1, \dots, \alpha_n]}{\Gamma[T_1, \dots, T_n] \vdash p[T_1, \dots, T_n]} \quad (\text{INST_TYPE})$$

- Rule **INST** is analogous but describes the instantiation of variables by terms of the same type.

$$\frac{\Gamma[x_1, \dots, x_n] \vdash p[x_1, \dots, x_n]}{\Gamma[t_1, \dots, t_n] \vdash p[t_1, \dots, t_n]} \quad (\text{INST})$$

HOL also has type and term definition principles. These yield theorems that connect the new type constructors to non-empty subsets of existing types, and the new term constants to existing terms. In both cases, the extended theory can be shown to have the same set-theoretic semantic model as the original one. Lastly, in case of HOL-Light, there are just three axioms: eta-conversion from

λ -calculus, an axiom for Hilbert's choice which make the logic classical, and an axiom expressing infinity of type *ind*.

HOL theorem proving systems have earned a reputation as being extremely trustworthy. This stems to a significant degree from their clean system architecture that follows the LCF approach [6], as well as the combination of a small number of basic inference rules and axioms with a few definitional extension principles that are logically safe.

Limitations of Simply-Typed Polymorphism

A lot of theorem proving can be done within the simply-typed HOL polymorphic framework. However, there are some cases where its limitations do bite. Generic Programming is a calculus that allows programs to be parameterised with type operators. While there has been some research on formalising this framework in HOL [5], the simple-type restriction does not allow to apply it at the level of polymorphic HOL functions.

As an example, consider a predicate *functor* which asserts that a HOL function ϕ is a functor from the category of HOL functions to itself. According to its mathematical definition, this means that ϕ should preserve the identity ($id = \lambda x. x$) and function composition:

$$\phi \text{ id} = \text{id} \tag{1}$$

$$\phi (f \circ g) = \phi f \circ \phi g \tag{2}$$

A typical example for such a function ϕ is the map operation on lists:

$$\begin{aligned} \text{map_list} &:: (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list} \\ \text{map_list } f \ [] &= [] \\ \text{map_list } f \ (\text{cons } x \ xs) &= \text{cons } (f \ x) \ (\text{map_list } f \ xs) \\ \text{map_list } \text{id} &= \text{id} \\ \text{map_list } (f \circ g) &= \text{map_list } f \circ \text{map_list } g \end{aligned}$$

There are two problems if one tries to define a general HOL predicate *functor* that characterises those functions ϕ that satisfy rules (1, 2):

1. From abstracting the type constructor *list* in the *map_list* example, one would expect *functor* to be parameterised with a unary type operator variable. There are no type operator variables in HOL.
2. In general, the three occurrences of ϕ in equation (2) all have different types. It is not possible to have differently typed instances of one variable in HOL.

The conclusion is that simply-typed polymorphic higher order logic is not expressive enough for the formalisation of such a *functor* predicate on polymorphic HOL functions. Similar problems arise with the application of other category-theoretic concepts such as natural transformations or initial algebras.

3 The Logic HOL2P

3.1 HOL2P Types

The type system of HOL2P extends simply typed HOL with:

- Universal types ($\Pi \alpha. T$) which bind a type variable α in a type T .
- Type variables with an arity greater than zero. These will normally be referred to as “type operator variables” in order to distinguish them from the usual 0-ary type variables.

Universal types are known from the polymorphic λ -calculus (“System F”) [4]. By supporting nested type quantification, they make it possible to define functions with “truly polymorphic” arguments that can be instantiated with different types. Type operator variables have been used in some programming language type systems [16].

The syntax of HOL2P types is as follows ($n \geq 0$):

$T ::= (\alpha :: \textit{rank})$	type variable (rank = <i>large</i> or rank = <i>small</i>)
$(T_1, \dots, T_n) \tau$	type constructor application
$(T_1, \dots, T_n) \theta$	type operator variable application
$\Pi (\alpha :: \textit{small}). T$	universal type

For an n -fold universal type, there is the usual abbreviated notation:

$$\Pi \alpha_1 \dots \alpha_n. T = \Pi \alpha_1. (\dots (\Pi \alpha_n. T))$$

The *rank* of a type variable is either *large* or *small*. A type will be called *small* if it does not contain any large type variables and no universal types. The instantiation of small type variables is restricted to small types. Small HOL2P types that do not contain any type operator variables correspond to the normal HOL types.

The introduction of universal types that bind type variables leads to a distinction between *free* and *bound type* type variables in a type. A bound type variable is always small. Type operator variables can not be bound in HOL2P and are thus always free. A type is called *closed* if it has no free type variables and no type operator variables. Two types are *α -convertible* if they can be made identical by a replacement of bound type variables with fresh new type variables. There is no β -conversion of types, as the HOL2P type system does not have a corresponding application operation.

When writing down HOL2P formulas and types, all free type variables are assumed by default to be large, while the bound type variables are small. With this convention in mind, the explicit indication of the rank of type variables can usually be omitted. As an example, here is a HOL2P type for the *functor* predicate from above:

$$\textit{functor} :: (\Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \theta \rightarrow \beta \theta) \rightarrow \textit{bool} \quad (3)$$

The definition of type substitution in HOL2P has to take bindings of type variables into account. Only free type variables can be instantiated. Also, the capture of type variables needs to be avoided. The latter can be ensured as usual by performing α -conversions that introduce fresh bound type variables as necessary. Lastly, as mentioned above, a small type variable may only be instantiated with a small type.

The substitution of an n -ary type operator variable θ can take two forms.

- The replacement is an n -ary type constructor τ . In this case, the replacement of an occurrence $(T_1 \dots T_n) \theta$ is simply the type $(T_1, \dots, T_n) \tau$.
- The replacement can be an n -ary type operator that is specified using the universal type syntax $(\Pi \alpha_1 \dots \alpha_n. T)$ with T a small type. In this case, the replacement of $(T_1 \dots T_n) \theta$ is the type T with occurrences of each variable α_i replaced by the corresponding argument type T_i .

As an example for a type substitution, consider the type of predicate $\text{functor}(\tilde{3})$. Replacing the 1-ary type variable θ with the type constructor list yields:

$$(\Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}) \rightarrow \text{bool}$$

while a substitution of θ with the 1-ary type operator specified by $(\Pi \alpha. (\alpha \text{ list}) \text{ list})$ gives the following instance:

$$(\Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \text{ list}) \text{ list} \rightarrow (\beta \text{ list}) \text{ list}) \rightarrow \text{bool}$$

HOL2P has the same primitive type constructors as HOL, and new type constructors can be introduced that are isomorphic to non-empty subsets of existing types. Such type definitions will be restricted to small types, as this allows the simple syntactic characterisation of small types given above.

3.2 HOL2P Terms

HOL2P extends the terms of HOL with type abstractions and type applications. This gives the following syntax of terms:

$t ::= (v : T)$	variable
$(c : T)$	constant
$t t$	application
$\lambda (v : T). t$	abstraction
$\Lambda \alpha. t$	type abstraction
$t [T]$	type application

There are two important restrictions on the formation of terms:

- (R1) As in System F, the formation of a type abstraction term $\Lambda \alpha. t$ has the side condition that the type variable α must not occur freely in the type of free variable of t .
- (R2) In a type application term $t [T]$, T must be a small type.

The typing rules for HOL2P variables, constants, applications and abstractions are as in HOL. For type abstractions and type application terms, we have:

$$\begin{array}{l} \Lambda \alpha. (t : T) \quad : \Pi \alpha. T \\ (t : \Pi \alpha. S) [T] : S[T \setminus \alpha] \end{array}$$

It follows that the type variable α in a type abstraction term $(\Lambda \alpha. t)$ as well as all type variables within the argument type T of a type application $(t [T])$ must be small. This will be assumed whenever a HOL2P term is written down.

With the exception of type operator variables, the types and terms of HOL2P are a subset of those of System F. The typing rules of HOL2P agree with those in System F, apart from an implicit “type β -reduction” that needs to be applied to the System F type of a type application term so as to obtain the corresponding HOL2P type.

Type membership in HOL2P is unique modulo α -equivalence of types. In particular, if T and T' are two α -convertible types and t is a term, then $t : T$ holds if and only if $t : T'$.

Having both term and type variable bindings in HOL2P terms means that α -equivalence needs to allow for a renaming of either kind of bound variables. It also gives rise to a β -conversion of type abstraction/application combinations discussed in Section 3.3.

Because of restriction (R1) on the formation of type abstractions, the free variables in a type abstraction $\Lambda \alpha. t$ or application $t [T]$ are the same as the free variables of the base term t . This implies that the two new term formation operations do not interfere with term substitutions, i.e. for a term variable v and a replacement term r of the same type as v , we have:

$$\begin{array}{l} (\Lambda \alpha. t) [r \setminus v] = \Lambda \alpha. t [r \setminus v] \\ (t [T]) [r \setminus v] = (t [r \setminus v]) [T] \end{array}$$

For type instantiations of HOL2P terms, the possible capture of type variables in type abstractions needs to be taken into account. This can be solved as usual by α -conversions that introduce fresh bound type variables.

In the area of theorem proving, universal types are well known from systems based on intuitionistic logic such as COQ [17]. For classical higher order logic, there is the problem that its combination with the full polymorphic λ -calculus is inconsistent [3]. In HOL2P, this is avoided by permitting only a subset of polymorphic λ -calculus terms - (R2) restricts HOL2P terms such that only type abstractions over small types can occur.

HOL2P is also restricted in the sense that the formation of universal types is limited to 0-ary type variables, despite the presence of type operator variables in the system. Because of these restrictions, the term “second order polymorphism” seems appropriate for HOL2P.

3.3 Rules

All HOL inference rules also apply in HOL2P. As an example, the inference rule REFL

$$\frac{}{\Gamma \vdash t = t} \quad (\text{REFL})$$

expresses the reflexivity of equality on HOL2P terms. In case of rule `INST_TYPE`, HOL2P extends the HOL version with type operator variable instantiation, and the instantiation of type (operator) variables has to preserve the smallness of types.

HOL2P has three additional inference rules compared to HOL. These support equational reasoning with type abstractions and applications.

- The congruence rule for type variable abstraction is analogous to the HOL congruence rule `ABS` for term abstractions. It also has a similar side condition in that the type variable α must not occur freely in the assumption list Γ :

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash \Lambda \alpha. s = \Lambda \alpha. t} \quad (\text{TYABS})$$

- The congruence rule for type application is reminiscent of the HOL rule `MK_CONG`, but the assumption about equality of the argument terms is replaced by a side condition that the two argument types S and T have to be α -equivalent.

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash s [S] = t [T]} \quad (\text{TYAPP})$$

- There is also a β -conversion rule for combinations of type abstraction and applications terms. This is similar to the HOL rule `BETA`. Again, the general case of this rule can be derived from the special case by suitable instantiations.

$$\frac{}{\Gamma \vdash (\Lambda \alpha. t) [\alpha] = t} \quad (\text{TYBETA})$$

Lastly, the three HOL axioms and the type and term definition principles also carry over to HOL2P. As already explained, there is a pragmatic restriction in that type definitions are not allowed to involve universal types, as this avoids the need to annotate type constructors with a small/large label.

4 Outline of HOL2P Semantics

Simple-typed higher order logic has a set-theoretic semantics due to A. Pitts [8]. Recently, J. Harrison [10] has proof checked a variation of this semantics in HOL-Light. Our presentation takes some inspiration from the latter source. In particular, we follow its lead by constructing only a standard model for the three primitive HOL type constructors *bool*, *ind* and function space. It is straightforward (but somewhat tedious) to extend the approach so as to cater for other HOL type constructors introduced by type definitions.

The HOL2P semantics will be given in four steps:

1. A recap of the standard semantics for HOL types and terms.
2. The construction of a universe for HOL2P types.
3. The semantics of HOL2P types.
4. The semantics of HOL2P terms.

4.1 The Semantics of HOL Terms

According to the well-known semantics of simple-typed HOL [8], there exists a universe \mathcal{U}_0 of sets such that each *monomorphic* HOL type T is modelled by a set $\llbracket T \rrbracket \in \mathcal{U}_0$. In addition, it can be assumed that the interpretation is standard:

- the model of type *bool* is a special two-element set `boolset`,
- the model of type *ind* is a special infinite set `indset` and
- the model of a function type $A \rightarrow B$ is the set theoretic function space between the models of A and B :

$$\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \rightarrow_{set} \llbracket B \rrbracket$$

The subscript *set* in \rightarrow_{set} is used to distinguish the set theoretic function space from the HOL function space.

The semantics $\llbracket T \rrbracket_\rho$ of a general, possibly *polymorphic*, HOL type T is parameterised with a valuation function ρ which associates each type variable with an element of \mathcal{U}_0 .

The meaning $\llbracket t \rrbracket_{\sigma, \rho}$ of a HOL term t is additionally parameterised with a valuation σ that associates each variable with an element of some set in \mathcal{U}_0 .

The term model has to be consistent with the type model:

$$\llbracket t : T \rrbracket_{\sigma, \rho} \in \llbracket T \rrbracket_\rho$$

4.2 A Universe for HOL2P Types

Because of restriction (R2), the type variable α in a HOL2P universal type $\Pi \alpha. T$ can only be instantiated with a small HOL2P type. Ignoring type operator variables for the moment, small HOL2P types correspond to types of HOL, and thus can be interpreted as U_0 elements. This suggests that HOL2P universal types can be modelled as products over U_0 . This observation leads us to introduce a new set-theoretic universe \mathcal{U}_1 with the following properties:

- \mathcal{U}_1 extends \mathcal{U}_0 : $\mathcal{U}_0 \subseteq \mathcal{U}_1$
- \mathcal{U}_1 is closed with respect to the function space operator: $A \in \mathcal{U}_1 \wedge B \in \mathcal{U}_1 \rightarrow A \rightarrow_{set} B \in \mathcal{U}_1$
- \mathcal{U}_1 is closed with respect to the formation of set theoretic products indexed by \mathcal{U}_0 , i.e. for $f : \mathcal{U}_0 \rightarrow \mathcal{U}_1$ we have $\Pi_{u \in \mathcal{U}_0} f u \in \mathcal{U}_1$.

Similar to the case of \mathcal{U}_0 , such a set \mathcal{U}_1 can be constructed iteratively by starting from the base set \mathcal{U}_0 and adding in iteration $(n + 1)$ all the sets that can be formed from the elements of iteration n by applying either the function space operator or products indexed over \mathcal{U}_0 .

4.3 The Semantics of HOL2P Types

The interpretation of HOL2P types has to take type operator variables of arity $n > 0$ into account. These will be modelled as n -ary functions of type $\mathcal{U}_1^n \rightarrow \mathcal{U}_1$. This means that the semantics $\llbracket T \rrbracket_\rho$ of a HOL2P type T is parameterised with a valuation function ρ which maps each type (operator) variable of arity $n \geq 0$ to an element of $\mathcal{U}_1^n \rightarrow \mathcal{U}_1$.

Small HOL2P types should be modelled as elements of the \mathcal{U}_0 subset of \mathcal{U}_1 . This is ensured by imposing two restrictions on the valuation functions ρ used in HOL2P type interpretations. Firstly, they have to respect smallness of type variables, e.g. small type variables must be mapped to elements of \mathcal{U}_0 :

$$\rho(\alpha :: \text{small}) \in \mathcal{U}_0$$

Secondly, because type operator variable applications preserve the smallness of types, the valuation $\rho \theta$ of a type operator variable must be a function that preserves the \mathcal{U}_0 subset of \mathcal{U}_1 :

$$u_1 \in \mathcal{U}_0 \wedge \dots \wedge u_n \in \mathcal{U}_0 \Rightarrow (\rho \theta)(u_1, \dots, u_n) \in \mathcal{U}_0$$

The definition of $\llbracket T \rrbracket_\rho$ is by primitive recursion over the structure of T :

$$\begin{aligned} \llbracket \text{bool} \rrbracket_\rho &= \text{boolset} \\ \llbracket \text{ind} \rrbracket_\rho &= \text{indset} \\ \llbracket A \rightarrow B \rrbracket_\rho &= \llbracket A \rrbracket_\rho \rightarrow_{\text{set}} \llbracket B \rrbracket_\rho \\ \llbracket \alpha \rrbracket_\rho &= \rho \alpha \\ \llbracket (T_1, \dots, T_n) \theta \rrbracket_\rho &= (\rho \theta)(\llbracket T_1 \rrbracket_\rho, \dots, \llbracket T_n \rrbracket_\rho) \\ \llbracket \Pi \alpha. T \rrbracket_\rho &= \Pi_{u \in \mathcal{U}_0} \llbracket T \rrbracket_{\rho[\alpha \mapsto u]} \end{aligned}$$

The first four equations above define the semantics of small HOL2P types without type operator variables. This definition agrees with the normal HOL type semantics.

4.4 The Semantics of HOL2P Terms

For variable, constant, application and abstraction terms in HOL2P, the meaning $\llbracket t \rrbracket_{\sigma, \rho}$ of t with respect to valuations σ and ρ can be defined in the same way as in HOL [10]. For the two HOL2P specific term operations, we have:

$$\begin{aligned} \llbracket \Lambda \alpha. t \rrbracket_{\sigma, \rho} &= \lambda_{\text{set}}(u \in \mathcal{U}_0). \llbracket t \rrbracket_{\sigma, \rho[\alpha \mapsto u]} \\ \llbracket t [T] \rrbracket_{\sigma, \rho} &= \llbracket t \rrbracket_{\sigma, \rho} \llbracket T \rrbracket_\rho \end{aligned}$$

By induction over the term structure, it can be shown that the meaning of HOL2P terms is consistent with that of HOL2P types, e.g.:

$$\llbracket t : T \rrbracket_{\sigma, \rho} \in \llbracket T \rrbracket_\rho$$

The reader might wonder where the construction of a semantics would break down if we allowed terms ranging over the full polymorphic λ -calculus in our

system. The crucial property is that in case of HOL2P, the model of a universal type is a product indexed by the fixed set \mathcal{U}_0 . This allows the iterative construction of a universe \mathcal{U}_1 with the required properties in a countable number of steps.

Lastly, the three inference rules specific to HOL2P express congruence properties of type abstractions and applications as well as β -reduction. The validity of these rules in the model follows directly from the corresponding properties of set theoretic functions and their applications.

5 Application Examples

Type Quantification

As a first application, we show that HOL2P can express type quantification terms [14]. For a predicate p and a type variable α , we define $\forall \alpha. p$ and $\exists \alpha. p$ to be abbreviations for the following *boolean* HOL2P terms:

$$\forall \alpha. p \equiv ((\Lambda \alpha. p) = (\Lambda \alpha. True)) \quad (4)$$

$$\exists \alpha. p \equiv ((\Lambda \alpha. p) \neq (\Lambda \alpha. False)) \quad (5)$$

As usual, a multiple universal quantification can be written in a shorter form:

$$\forall \alpha_1 \dots \alpha_n. p \equiv \forall \alpha_1. \dots \forall \alpha_n. p$$

The introduction rule for universal quantification over types follows directly from its definition and the HOL2P TYABS rule. It is subject to the usual side condition that α is not free in the assumption list Γ :

$$\frac{\Gamma \vdash p}{\Gamma \vdash \forall \alpha. p} \quad (\text{TYALL-I})$$

The elimination rule for universal quantification over types follows directly from the HOL2P TYAPP rule and the “type- β ” conversion for type application/type abstraction combinations (e.g. the general case of TYBETA):

$$\frac{\Gamma \vdash \forall \alpha. p}{\Gamma \vdash p[T \setminus \alpha]} \quad (\text{TYALL-E})$$

There are similar rules for existential quantification over types which can be derived either from its definition above or by rewriting it in form of a negated universal type quantification:

$$\exists \alpha. p \equiv \neg(\forall \alpha. \neg p) \quad (6)$$

As an example for a derivation, consider the statement that there exists a HOL2P type which is a terminal object in the category of HOL2P functions:

$$\exists \alpha. \forall f g. f = (g : \beta \rightarrow \alpha)$$

The proof of this statement is by contradiction using the fact that the unit type 1 is a witness, e.g. it is such a terminal object:

$$\begin{aligned}
& \exists \alpha. \forall f g. f = (g : \beta \rightarrow \alpha) \\
& \Leftrightarrow \{ \text{Equation (6), negation rules} \} \\
& \forall \alpha. \neg(\forall f g. f = (g : \beta \rightarrow \alpha)) \vdash \text{False} \\
& \Leftarrow \{ \text{Forward reasoning from assumption by TYALL-E with } T = 1 \} \\
& \neg(\forall f g. f = (g : \beta \rightarrow 1)) \vdash \text{False} \\
& \Leftrightarrow \{ \text{property of unit type 1} \} \\
& \text{True}
\end{aligned}$$

Notions from Category Theory

The predicate *functor* discussed in Section 2 applies to universally type-quantified functions ϕ :

$$\phi : \Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \theta \rightarrow \beta \theta) \quad (7)$$

It is defined in HOL2P as follows (compare with equations (1, 2) above):

$$\begin{aligned}
\text{functor } \phi = & (\forall \alpha. \phi [\alpha] [\alpha] = (\text{id} : \alpha \theta \rightarrow \alpha \theta)) \\
& \wedge \forall \alpha \beta \gamma. \forall (f : \alpha \rightarrow \beta)(g : \beta \rightarrow \gamma). \phi (g \circ f) = \phi g \circ \phi f
\end{aligned}$$

For example, the fact that *map_list* is a functor is expressed by the proposition:

$$\text{functor } (\Lambda \alpha \beta. (\text{map_list} : (\alpha \rightarrow \beta) \rightarrow (\alpha \text{ list} \rightarrow \beta \text{ list})))$$

This shows one of the costs of working in HOL2P - the definition of the *map_list* constant is given in a form that has free type variables α and β , and these type variables have to be bound if we want to use *map_list* as an argument to *functor* or similar functions that operate on polymorphic arguments.

The definition of predicate *functor* is parameterised with a free one-ary type operator variable θ . This is instantiated whenever the constant *functor* is applied to a function ϕ , e.g. we have $\theta = \text{list}$ in case of $\phi = \text{map_list}$.

Natural transformations are a key concept of category theory [13]. Given two functors ϕ and ψ between categories C and D , a natural transformation associates every object X in C with a “morphism” $\eta_X : \phi X \rightarrow \psi X$ subject to a commutativity rule:

$$\psi(f) \circ \eta_X = \eta_Y \circ \phi(f)$$

In our application of category theory to polymorphic HOL functions embedded within HOL2P, natural transformations are simply polymorphic functions which are compatible with two “functor” functions ϕ and ψ of a type as in (7). This leads to the definition of a three-argument predicate *ntrf* as follows:

$$\begin{aligned}
\text{ntrf} & :: (\Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \phi \rightarrow \beta \phi)) \\
& \rightarrow (\Pi \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \psi \rightarrow \beta \psi)) \\
& \rightarrow (\Pi \alpha. \alpha \phi \rightarrow \alpha \psi) \rightarrow \text{bool} \\
\text{ntrf } \phi \psi \eta = & \forall \alpha \beta. \forall (f : \alpha \rightarrow \beta). \psi f \circ \eta = \eta \circ \phi f
\end{aligned}$$

Since HOL2P functions are total, the predicate *ntrf* can be applied to any functions ϕ and ψ that are of the right type, not just functors.

Many polymorphic functions are natural transformations between “type functors”. In fact, this statement is just a rephrasal of the “theorem for free” [20] one can infer (under certain provisos) from the type of a function. As a simple example, consider the function *rev* which reverses the order of elements in a list:

$$\begin{aligned} \text{rev } [] &= [] \\ \text{rev } (\text{cons } h \ t) &= \text{append } (\text{rev } t) \ [h] \end{aligned}$$

It is a “natural transformation from lists to lists”, or to be more precise, from the *map_list* functor to the *map_list* functor:

$$\begin{aligned} \text{ntrf } &(\Lambda \alpha \ \beta. (\text{map_list} : (\alpha \rightarrow \beta) \rightarrow (\alpha \ \text{list} \rightarrow \beta \ \text{list}))) \\ &(\Lambda \alpha \ \beta. (\text{map_list} : (\alpha \rightarrow \beta) \rightarrow (\alpha \ \text{list} \rightarrow \beta \ \text{list}))) \\ &(\Lambda \alpha. (\text{rev} : \alpha \ \text{list} \rightarrow \alpha \ \text{list})) \end{aligned}$$

After unfolding definitions, it becomes apparent that this statement is equivalent to the following better known proposition:

$$\text{rev } (\text{map_list } f \ x) = \text{map_list } f \ (\text{rev } x)$$

Category theory defines a “horizontal” and a “vertical” composition of natural transformations, as well as compositions with functors. It is straightforward to translate these operations so that they apply to the HOL2P natural transformations defined above, and to show that the results of the operations are again natural transformations. Formal proofs can be found in the HOL2P example files. These also contain some further derivations related to homomorphisms, initial algebras and catamorphisms.

6 Implementation

It would be possible to implement a HOL2P proof assistant in a green-field software development project, starting from scratch. However, because the logic is an extension of HOL, and given the availability of several high quality HOL implementations, it makes more sense to built such a system by trying to reuse at least parts of existing provers.

For the prototype HOL2P implementation described in this paper, HOL-Light was chosen as the starting point. This decision was made because of its compactness and clean design which helps to keep the implementation of an extension more manageable. Having identified the base system, a fundamental question about the new HOL2P prover arises: should it aim to stay compatible with the existing HOL-Light prover and its associated libraries or should its design reflect the extended logic as much as possible?

In order to simplify the reuse of the existing HOL-Light libraries, it was decided to try and keep the number of modifications to a minimum. This was

achieved by taking an iterative development approach. Starting off with the existing HOL-Light system, one by one, modules were replaced with a new HOL2P version. Whenever a new module was completed, a test was carried out to ensure that the whole system was still a functioning theorem proving environment. In particular, interfaces were only ever extended, and not modified in any other way. Similarly, all functions kept their original signature, although the functionality they provide now applies to the more general case of HOL2P types and terms.

A typical example of this conservative development philosophy is the basic *hol_type* datatype that represents HOL/HOL2P types. In HOL-Light, this type is defined as:

$$\mathit{hol_type} = \mathit{Tyvar\ of\ string} \mid \mathit{Tyapp\ of\ string} * \mathit{hol_type\ list}$$

In principle, there are several possible options for the design of a corresponding HOL2P *hol_type* datatype. Having both type variables and type operator variables in the logic suggests a generalisation of *Tyvar* so that it takes a second arity argument, as well as introducing a type application operator similar to System F. Disregarding universal types for the moment, this might lead to a type declaration such as the following:

$$\begin{aligned} \mathit{hol_type} = & \mathit{Tycon\ of\ string} \mid \mathit{Tyvar\ of\ string} * \mathit{rank} * \mathit{int} \\ & \mid \mathit{Tyapp\ of\ hol_type} * \mathit{hol_type\ list} \mid \dots \end{aligned}$$

While perfectly acceptable for a new design, this definition was rejected. The rationale was that some HOL-Light libraries make use of the visibility of *hol_type* in order to define functions by pattern matching. Changing the existing constructors would thus have implied changes to such libraries. Instead, the definition of *hol_type* in HOL2P keeps the existing constructors and simply adds a new constructor for universal types ¹

$$\begin{aligned} \mathit{hol_type} = & \mathit{Tyvar\ of\ string} \mid \mathit{Tyapp\ of\ string} * \mathit{hol_type\ list} \\ & \mid \mathit{Utype\ of\ hol_type} * \mathit{hol_type} \end{aligned}$$

This declaration raises the question how small and large type variables are distinguished. This is done on a purely syntactic basis. Small type variables are recognised by starting with an apostrophe character. Similarly, type operator variables are distinguished from type constructors by starting with an underscore character.

The reward of this conservative development philosophy is a high degree of backward compatibility of the new HOL2P system with HOL-Light, and it is possible to run at least the main HOL-Light theory developments without any changes.

¹ The reader might be surprised that the first argument of the *Utype* constructor is not simply a *string*, e.g. the type variable name. It was done in order to ease reuse by keeping the representation in line with the HOL-Light treatment of term level abstractions. The latter employs a constructor *Abs* that takes a *term* as first argument, and not just a variable name.

A brief summary of some major implementation aspects is given below. The reader interested in details is referred to the system source files and accompanying documentation [18].

- The main challenge in the implementation of the logical core was the presence of type variable bindings in HOL2P. In particular, this makes the type and term substitution code more complex as well as requiring a notion of α -equivalence of types. In order to achieve a uniform treatment of instantiations, type operator variables, constant names and replacement type operators are all represented as *hol_type* elements.
- For the implementation of derived logical operations, the main difficulty is the matching of types in the presence of type operator variables. Similar to normal “higher order” term matching with function variables, it is possible to have more than one match. For example, trying to solve:

$$?F(?x) = (\text{nat list}) \text{ list}$$

there are several possible solutions, i.e.

- | | | |
|-----|---|---------------------------------------|
| (1) | $?F = \Lambda \alpha. \alpha$ | $?x = (\text{nat list}) \text{ list}$ |
| (2) | $?F = \text{list}$ | $?x = (\text{nat list})$ |
| (3) | $?F = \Lambda \alpha. \alpha \text{ list list}$ | $?x = \text{nat}$ |
| (4) | $?F = \Lambda \alpha. \text{nat list list}$ | $?x$ can be any HOL type |

In the current HOL2P implementation, a rather simple approach was chosen - the algorithm will only match type operator variables against a type constructor or another type operator variable. This means that only solution (2) is allowed. In order to return other solutions, an explicit instantiation of the type operator variable has to be performed by the user, e.g. by adding it in the statement of a goal or by instantiating a theorem before using it in a proof step. For example, after applying the instantiation $?F = \Lambda \alpha. \alpha$, the first solution will be returned, while setting $?F = \Lambda \alpha. (\alpha ?G) ?H$ will result in solution (3).

- In the implementation of type inference, the presence of type operator variables poses the same problem for the unification algorithm as for the matching algorithm above, and the same approach of restricting the number of solutions was chosen. In order to reduce the amount of explicit type annotations, user can explicitly declare the type of a variable. This information will then be used during the parsing of later goal statements and other terms input by the user. Unfortunately, despite this facility, a fair number of explicit type annotations and type instantiations can be necessary, see for example the category theory derivation source files. Additionally, the way type applications terms are handled in the parser makes it necessary to explicitly specify both the original type of a term as well as the type resulting from performing the operation.
- Boolean type quantification terms as described in Section 5 are treated simply as syntactic sugar, e.g. they are parsed into normal HOL2P terms according to the defining equations (4) and (5).

- Reasoning about terms containing type abstractions and type applications is supported by appropriate tactics. In particular there are tactics that perform the introduction and elimination of universal type quantification terms as described by rules (TYALL-I) and (TYALL-E) above. There is also a conversion that will perform a “type- β -reduction”, e.g. simplifying a type abstraction/type application combination by performing the corresponding type instantiation on the abstraction body term.

7 Conclusions and Further Work

HOL2P is a new logical system that extends classical higher order logic with type operator variables and universal types, albeit in a restricted form in order to ensure consistency. The paper shows that the logic is capable of expressing boolean type quantification terms, as well as allowing the application of basic category-theoretic notions on the level of polymorphic HOL functions.

HOL2P has been implemented as a modification of HOL-Light using a conservative approach which respects existing module interfaces and other visible module features. This has resulted in a proof assistant that is to a high degree backwards compatible with HOL-Light, and in particular it is possible to run the standard theories that come with that system.

The implementation of the logical core of the HOL2P was relatively straightforward by transferring techniques for dealing with bound term variables to the type level. The presence of type operator variables poses a major challenge when solving type equations occurring during matching of terms or when unification is required by type inference. In the current HOL2P implementation, a simple approach was taken which restricts the number of automatically found solutions to at most one. In situations where another solution is required, users need to give explicit type instantiations or annotations, e.g. when stating goals or when specifying theorems that are to be applied during a proof.

Further work is necessary in order to investigate the practical value of HOL2P. For example, one could imagine more category-theory inspired datatype definition packages. These could make use of the extra expressiveness of HOL2P in order to derive theorems by instantiation from their generic counterparts, with the aim of reducing the amount of work done at the ML programming level. Similarly, one could consider recasting recursive function definition packages so that they operate on hylomorphisms [11] with the aim of simplifying further reasoning by making use of fusion properties. The use of monads [21] and the application of generic programming concepts might lead to more structure in the definition of functions and more calculational proofs. Other potential applications include model-theoretic reasoning where it can be useful to have a predicate that expresses the existence of a type.

HOL2P was conceived as a minimal extension of HOL that supports reasoning about HOL types and polymorphic HOL functions, with free type operator variables for genericity. It should be possible to extend the approach to arbitrary higher-rank polymorphism, provided one adds appropriate restrictions in

the type/term formation rules so as to avoid the impredicativity inherent in full System F. Another plausible extension would be the introduction of universal types that bind type operator variables.

With regards to the HOL2P implementation, further work should revisit the algorithms used for type matching, unification and type inference so as to reduce the amount of explicit type annotations required from the user. This work might benefit from recent advances in type inference research for higher-rank types [12,19]. Because of the complexity of HOL2P, it would be appropriate to mechanically check the semantics given in this paper, and ideally also some of the algorithms used in the implementation of the logical core, similar to the verification work done for HOL-Light [10].

Acknowledgements

The author would like to thank Ray Turner who suggested the idea of extending HOL with a second type layer, and who provided moral support. The anonymous TPHOLs reviewers deserve credit for their helpful comments.

References

1. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming - an introduction. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, Springer, Heidelberg (1999)
2. Bird, R.S., de Moor, O.: Algebra of Programming. Prentice-Hall, Englewood Cliffs (1997)
3. Coquand, T.: A new paradox in type theory. In: Prawitz, D., Skyrms, B., Westerstahl, D. (eds.) Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science, pp. 555–570. North-Holland, Amsterdam (1994)
4. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, Cambridge (1989)
5. Glimming, J.: Logic and Automation for Algebra of Programming. PhD thesis, MSc Thesis, University of Oxford (2001)
6. Gordon, M., Wadsworth, C.P., Milner, R.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
7. Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge University Press, Cambridge (1993)
8. Gordon, M.J.C., Pitts, A.: The HOL Logic and System. In: Bowen, J. (ed.) Towards Verified Systems. Real-Time Safety Critical Systems Series, vol. 2, Elsevier, Amsterdam (1994)
9. Harrison, J.: The HOL Light System Reference – Version 2.20 (2006), <http://www.cl.cam.ac.uk/~jrh13/hol-light>
10. Harrison, J.: Towards self-verification in HOL-Light. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, Springer, Heidelberg (2006)
11. Hu, Z., Iwasaki, H., Takeichi, M.: Deriving structural hyломorphisms from recursive definitions. In: ICFP, pp. 73–82 (1996)
12. Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming 17(1) (2007)

13. Mac Lane, S.: *Categories for the Working Mathematician*. Springer, Heidelberg (1971)
14. Melham, T.F.: The HOL logic extended with quantification over type variables. *Formal Methods in System Design* 3(1–2), 7–24 (1994)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: *Isabelle/HOL*. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Pierce, B.C.: *Types and Programming Languages*. MIT Press, Cambridge (2002)
17. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0* (2004), <http://coq.inria.fr>
18. Völker, N.: *HOL2P Prototype Implementation* (2007), <http://cswww.essex.ac.uk/staff/norbert/hol2p>
19. Vytiniotis, D., Weirich, S., Jones, S.P.: Boxy types: inference for higher-rank types and impredicativity. *ACM SIGPLAN Notices* 41(9), 251–262 (2006)
20. Wadler, P.: Theorems for free? In: *Functional Programming Languages and Computer Architecture*, Springer, Heidelberg (1989)
21. Wadler, P.: The essence of functional programming. In: *POPL*, pp. 1–14 (1992)