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

Norbert Völker

University of Essex, England

TPHOLs 2007

Starting Point: HOL

- Classical higher order logic
- Simply typed λ-calculus without overloading
- Familiar type and term system
- Implemented in HOL System, HOL-LIGHT, ...
- In this talk: "HOL" means by default "HOL as in HOL-LIGHT"

HOL Limitation: Shallow Polymorphism

- HOL types:

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

- No binding of type variables.
- All occurrences of one (term) variable must have the same type.

$$\begin{array}{lcl} \textit{double} & :: & (\alpha \to \alpha) \to \alpha \to \alpha \\ \textit{double} & = & \lambda\,(f : \alpha \to \alpha).\ (f : \alpha \to \alpha) \circ (f : \alpha \to \alpha) \end{array}$$

- A constant can have differently typed occurrences, as in $(\textit{id}\ \textit{id})$. For a variable f it is impossible to form $(f\ f)$.

Challenge: *functor* Predicate in HOL

- Consider function:

$$\begin{array}{lccccc} \mathit{map} : & (\alpha \to \beta) & \to & \alpha\ \mathit{list} & \to & \beta\ \mathit{list} \\ \mathit{map} & f & & [x_1, \ldots, x_n] & = & [f\ x_1, \ldots, f\ x_n] \end{array}$$

  It fulfills the functor rules:

$$\begin{array}{lcl} \mathit{map}\ \mathit{id} & = & \mathit{id} \\ \mathit{map}\ (f \circ g) & = & \mathit{map}\ f \circ \mathit{map}\ g \end{array}$$

- Challenge: express these rules in a predicate, abstracting over *map*.

$$\mathit{functor}\ \phi \quad = \quad ((\phi\ \mathit{id} = \mathit{id}) \wedge \forall f\ g.\ \phi\ (f \circ g) = \phi\ f \circ \phi\ g)$$

- What is the type of ϕ?

Challenge: *functor* Predicate in HOL

- Consider function:

$$\begin{array}{lccccc} \mathit{map} : & (\alpha \to \beta) & \to & \alpha\ \mathit{list} & \to & \beta\ \mathit{list} \\ \mathit{map} & f & & [x_1, \ldots, x_n] & = & [f\ x_1, \ldots, f\ x_n] \end{array}$$

  It fulfills the functor rules:

$$\begin{array}{lcl} \mathit{map}\ \mathit{id} & = & \mathit{id} \\ \mathit{map}\ (f \circ g) & = & \mathit{map}\ f \circ \mathit{map}\ g \end{array}$$

- Challenge: express these rules in a predicate, abstracting over *map*.

$$\mathit{functor}\ \phi \quad = \quad ((\phi\ \mathit{id} = \mathit{id}) \wedge \forall f\ g.\ \phi\ (f \circ g) = \phi\ f \circ \phi\ g)$$

- What is the type of ϕ? Answer: $\phi : (\alpha \to \alpha) \to \beta \to \beta$
  This is noto a good abstraction over *map*.
- There is no satisfactory definition of *functor* in HOL.

HOL2P Types

$$
\begin{array}{rcll}
T & ::= & \alpha & \text{unconstrained (“large”) type variable} \\
 & | & {'\alpha} & \text{small type variable} \\
 & | & (T_1, \ldots, T_n)\ \tau & \text{type constructor application} \\
 & | & (T_1, \ldots, T_n)\ \theta & \text{type operator variable application} \\
 & | & \Pi\ {'\alpha}.\ T & \text{universal type}
\end{array}
$$

- Some examples:

$$(num)\theta$$

$$\Pi\ {'\alpha}.{'\alpha} \to {'\alpha}$$

$$(\Pi {'\alpha}\ {'\beta}.\ ({'\alpha} \to {'\beta}) \to {'\alpha}\ \theta \to {'\beta}\ \theta) \to bool$$

- Universal types: see polymorphic λ-calculus (“System F”)
- Type operator variables are known from FP type systems.

Small Types

A HOL2P type T is called small if:

- It is a small type variable $'\alpha$ or
- it is a type constructor or type operation application, and all argument types are small.

A HOL2P type T is called large or unconstrained if it is not small.

- Contains a large type variable, or a universal type.

Small types form an embedding of normal HOL in HOL2P.

Exercise: small or large?

$$(num)\theta \qquad '\alpha \to \alpha \qquad (\Pi\, '\alpha.\, '\alpha \to '\alpha) \to bool$$

Small Types

A HOL2P type T is called small if:

- It is a small type variable $'\alpha$ or
- it is a type constructor or type operation application, and all argument types are small.

A HOL2P type T is called large or unconstrained if it is not small.

- Contains a large type variable, or a universal type.

Small types form an embedding of normal HOL in HOL2P.

Exercise: small or large?

$(num)\theta$	$'\alpha \to \alpha$	$(\Pi\, '\alpha.\, '\alpha \to '\alpha) \to bool$
small	large	large

Small Types

A HOL2P type T is called small if:

- It is a small type variable $'\alpha$ or
- it is a type constructor or type operation application, and all argument types are small.

A HOL2P type T is called large or unconstrained if it is not small.

- Contains a large type variable, or a universal type.

Small types form an embedding of normal HOL in HOL2P.

Exercise: small or large?

$(num)\theta$	$'\alpha \to \alpha$	$(\Pi\, '\alpha.\, '\alpha \to '\alpha) \to bool$
small	large	large

Problem: HOL + System F $\rightsquigarrow$ inconsistent logic [T. Coquand]

- HOL2P idea: restrict universal types to abstraction over small types.

Consequences of Universal Types

- A universal type $\Pi' \alpha_1 \ldots {}' \alpha_n . T$ binds ${}' \alpha_1 , \ldots , {}' \alpha_n$ in T.
- Need to distinguish free and bound type variables occurrences.
- Bound type variables are always small.
- Type operator variables can not be bound.
- Types are α-equivalent if they are the same up to a renaming of bound type variables.

  α-equivalent types are identified for most purposes.
- There is no β-conversion of types.

  Types can be seen as implicitly type-β reduced.

Type Substitution

- Type substitution only replaces free type (operator) variables.
- Type substitution has to respect smallness.
- An n-ary type operator variable θ can be instantiated with an n ary type constructor τ, or with a type operator written in universal type syntax:

$$S[(\Pi' \alpha_1 \ldots ' \alpha_n .\, T) \backslash \theta]$$

Examples:

$$\begin{aligned}
(num\ \theta)[list \backslash \theta] &= num\ list \\
(num\ \theta)[(\Pi' \alpha .\, ' \alpha\ list\ list) \backslash \theta] &= num\ list\ list \\
(\Pi' \alpha .\, ' \alpha\ \theta \to \beta)[list \backslash \theta][num \backslash ' \alpha][num \backslash \beta] &= (\Pi' \alpha .\, ' \alpha\ list \to num)
\end{aligned}$$

Terms

$$
\begin{array}{rcll}
t & ::= & (v : T) & \text{variable} \\
 & | & (c : T) & \text{constant} \\
 & | & t\ t & \text{application} \\
 & | & \lambda\,(v : T).\, t & \text{abstraction} \\
 & | & \Lambda\,'\alpha.\, t & \text{type abstraction} \\
 & | & t\,[T] & \text{type application}
\end{array}
$$

Side conditions:

1. The type of a free variable v must not contain bound type variables.
2. In a type application $t\,[T]$, the argument type T must be small.

Typing rules:

$$
\begin{array}{lcll}
t : T & \vdash & (\Lambda\alpha.\, t) & : \Pi\alpha.\, T \\
t : \Pi\,'\alpha.\, S & \vdash & t\,[T] & : S[T\backslash\,'\alpha]
\end{array}
$$

Sample term:

$$(\Lambda\,'\alpha.\, id : {}'\alpha \to {}'\alpha)\,[num] \quad : \quad num \to num$$

Type Quantification and Terminal Type Example

Type quantification can be defined as an abbreviation in HOL2P:

$$\begin{aligned} \forall \alpha.\, p &\equiv ((\Lambda \alpha.\, p) = (\Lambda \alpha.\, \mathit{True})) \\ \exists \alpha.\, p &\equiv ((\Lambda \alpha.\, p) \neq (\Lambda \alpha.\, \mathit{False})) \end{aligned}$$

Existence of a "terminal type":

$$\exists\, '\alpha.\, \forall\, '\beta.\, \forall f\ g.\, f = (g : {}'\beta \to {}'\alpha)$$

Functor Example

$$
\begin{aligned}
\mathit{functor} \quad &:: \quad (\Pi\, 'a\ 'b.\ ('a \to 'b) \to 'a\ \theta \to 'b\ \theta) \to \mathit{bool} \\
\mathit{functor}\ \phi \quad &= \quad (\forall\, 'a.\ \phi\ ['a]\ ['a]\ \mathit{id} = \mathit{id}) \\
&\quad \wedge \forall\, 'a\ 'b\ 'c.\ \forall (f :: 'a \to 'b)(g :: 'b \to 'c). \\
&\qquad \phi\,['a]['c]\ (g \circ f) = \phi\,['b]['c]\ g \circ \phi\,['a]['b]\ f
\end{aligned}
$$

HOL2P Inference Rules

Type abstraction congruence rule ($'\alpha$ not free in Γ):

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\Lambda\, '\alpha.\, s) = (\Lambda\, '\alpha.\, t)} \quad \text{(TYABS)}$$

Type application congruence rule: (S and T must be α-equivalent):

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

β-conversion for type abstraction/application terms:

$$\frac{}{\Gamma \vdash (\Lambda\, '\alpha.\, t)\,['\alpha] = t} \quad \text{(TYBETA)}$$

- Term α-conversion allows both type and term variable renaming.
- HOL2P restriction on type constructor definitions: they must not involve universal types.

Derived Type Quantifier Rules

Universal type quantifier introduction ($'\alpha$ not free in Γ):

$$\frac{\Gamma \vdash p}{\Gamma \vdash \forall\, '\alpha.\, p} \qquad (\texttt{TYALL-I})$$

Universal type quantifier elimination:

$$\frac{\Gamma \vdash \forall\, '\alpha.\, p}{\Gamma \vdash p[T \backslash\, '\alpha]} \qquad (\texttt{TYALL-E})$$

Existential type quantifier introduction:

$$\frac{\Gamma \vdash p[T \backslash\, '\alpha]}{\Gamma \vdash \exists\, '\alpha.\, p} \qquad (\texttt{TYEX-I})$$

Existential type quantifier elimination ($'\alpha$ not free in Γ):

$$\frac{\Gamma \vdash \exists\, '\alpha.\, p}{\Gamma \vdash p} \qquad (\texttt{TYEX-E})$$

Example Proof: Existence of Terminal Type

$$\exists\, 'a.\ \forall\, 'b.\ \forall f\ g.\ f = (g : 'b \to 'a)$$

$\Leftrightarrow$ { Existential type quantifier introduction with T=1 }

$$\forall\, 'b.\ \forall\, (f : 'b \to 1)\ g.\ f = g$$

$\Leftrightarrow$ { Universal type introduction }

$$\forall\, (f : 'b \to 1)\ g.\ f = g$$

$\Leftrightarrow$ { property of unit type 1 }

True

- Reasoning about type quantification terms is similar to reasoning with normal $\forall/\exists$ quantified terms.

Informal Outline of Semantics

- Small monomorphic HOL2P types correspond to monomorphic HOL types. They can be interpreted as sets in some universe $\mathscr{U}_0$.
- Introduce a new universe $\mathscr{U}_1$ that
  - extends $\mathscr{U}_0$
  - is closed with respect to function spaces
  - is closed with respect to $\mathscr{U}_0$-indexed products: $\Pi_{u \in \mathscr{U}_0} f\ u$
- Crucial step in then semantics definition: the interpretation of type abstraction and application terms:

$$\begin{aligned} [\![\Lambda' \alpha.\, t]\!]_{\sigma,\rho} &= \lambda_{set}\ (u \in \mathscr{U}_0).\, [\![t]\!]_{\sigma,\rho[\alpha \mapsto u]} \\ [\![t\,[T]]\!]_{\sigma,\rho} &= [\![t]\!]_{\sigma,\rho}\ [\![T]\!]_{\rho} \end{aligned}$$

- Construction of $\mathscr{U}_1$ relies on the smallness constraint for type abstractions permitting a restriction to $\mathscr{U}_0$-indexed products.

The Implementation Approach

- HOL2P was implemented as a modification of HOL-LIGHT.
- Goal: I tried to reuse HOL2P as much as possible.
- Incremental approach, replacing modules one by one.
  After each module replacement, it was checked that the system was still a functioning theorem prover.
- Some awkwardness caused by conservative design philosophy:

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

- But also strong benefits:
  - Existing HOL-code served as test driver.
  - High degree of backward compatibility.

General Implementation Comments

- HOL-LIGHT has high quality code written in OCaml.
- Most complex task in type/term/theorem modules: type/term substitution and instantiation

  Need to keep track of variable and type variable bindings.
- Type matching and inference lead to (restricted) higher order problems.
- Type inference in HOL2P is a serious challenge.
- Type applications are often redundant, and can be omitted in some cases when entering terms.
- Type quantification terms are treated as syntactic sugar.
- The implementation of HOL2P tactics was easier than expected.

Type Matching and Inference Problem

Type operator variables make HOL2P type matching "higher order":

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

Such matching problems have in general several solutions:

(1)	$?F$	$=$	$\Lambda'\alpha.\,'\alpha$	$?x$	$=$	$(\mathit{nat\ list})\ \mathit{list}$
(2)	$?F$	$=$	list	$?x$	$=$	$(\mathit{nat\ list})$
(3)	$?F$	$=$	$\Lambda'\alpha.\,'\alpha\ \mathit{list\ list}$	$?x$	$=$	nat
(4)	$?F$	$=$	$\Lambda'\alpha.\,\mathit{nat\ list\ list}$	$?x$	$=$	$\mathit{"any\ type"}$

The current HOL2P implementation will only find solution (2).

- Use *TYPE_INST* for explicit instantiation of theorems in other cases.

HOL2P type inference leads to (restricted) higher order unification.

- HOL2P includes a *TYINST* facility for explicit instantiation of type (operator) variables in terms.

Results

- HOL2P is a minimal HOL extension that supports generic "Algebra of Programming" (AoP) reasoning on polymorphic HOL functions.
- HOL2P only allows abstraction over small types that correspond to normal HOL types.
  This restriction makes a set-theoretic semantics possible.
- HOL2P was implemented by modifying HOL-LIGHT, and is highly backwards compatible.
- HOL2P type matching and type inference lead to challenging higher order problems.
- Most advanced HOL2P example is currently AoP "Banana-Split" theorem and its application.

Further Work

- Research into
  - HOL2P matching and type inference problems
  - combinations of HOL2P with overloading/ type classes.
  - related extensions of HOL
- Application of HOL2P to further examples (especially AoP).
- Mechanically check the HOL2P semantics and the implementation of its logical core.
- ...