

# Extending Classical Higher-Order Logic with Second-Order Polymorphism - A Taster -



---

Norbert Völker  
CSEE Logic Seminar 17 May 2011



# Overview

---

- Simply-typed lambda-calculus
- Classical-higher order logic
- Beyond simply-typed polymorphism
  - Type-operator variables
  - Type abstraction
- Extending HOL with second-order polymorphism
  - Problem
  - Solution
  - Logic
- Related and further work



# Simply-Typed Polymorphic Lambda Calculus

---

- Terms
  - Types
  - Currying
  - Conversion rules
    - normal form
  - Type inference
- 
- HOL = "logic built on simply typed lambda calculus"



# HOL Terms and Types

---

$t ::= v : T$                       variable  
|  $c : T$                       constant  
|  $t t$                       application  
|  $\lambda (v : T). t$               abstraction

$T ::= \alpha$                       type variable  
|  $(T_1, \dots, T_n) \tau$         type constructor application ( $n \geq 0$ )

- HOL logical core
  - type constructors:  $bool$ ,  $(\rightarrow)$
  - term constants:  $(=)$ ,  $eps$
- Other types and terms can be *defined*.
- *Exercise*: what is the type of  $(=)$  and  $eps$ ?
- *Exercise*: name two constants of type  $(\alpha \rightarrow bool) \rightarrow bool$

# Classical Higher Order Logic (HOL)

- Core axioms:
  - equality is reflexive
  - equality is preserved by lambda calculus conversions
  - ...
- Definition principles for introducing new types and terms
- Classical:  $P \vee \neg P$
- Extensional:  $\forall x. f x = g x \Rightarrow f = g$
- Can define new types and then prove induction *theorems*
- Example:
$$P [] \wedge (\forall x xs. P xs \Rightarrow P (cons(x,xs)) \Rightarrow P (xs:(\alpha)list)$$
- Predicates can be seen as typed sets:
$$(\alpha) set \equiv (\alpha \rightarrow bool)$$
$$(x:\alpha) \in (xs: \alpha set) \equiv xs x$$



# How HOL Avoids Russel's Paradox

---

- Can we define "a set containing all those sets that do not contain themselves"?
- No - we can not define a predicate

$$\lambda (x:\alpha). \neg(x \in x)$$

- In fact, HOL has a set-theoretic semantics.



# Higher Order Unification and Matching

---

- HOL is cool
  - but more complicated than 1<sup>st</sup> order logic.
- Consider the basic unification problem
  - Given two terms  $t$  and  $u$ , is there a substitution  $\theta$  such that  $t\theta$  and  $u\theta$  have the same normal form?
  - This problem is undecidable.
- Higher-order matching is an instance where  $u$  is a closed term, so we are simply asking if  $t$  can be pattern-matched to  $u$ .
  - This has recently been proved decidable [Stirling 2007].  
The author says the proof is very complicated.
- There are special cases of higher-order matching and unification that can be solved efficiently.
  - Google comes up with 200K+ matches for the phrase "higher order unification".



# Type Extension Motivation 1: Category Theory

---

- HOL theory developments often make use of:
  - total functions:  $\alpha \rightarrow \beta$
  - partial functions:  $\alpha \rightarrow \beta$  *option*
  - relations:  $\alpha \sim \beta$
- Category theory has helped to unify mathematics - can it also unify HOL theory developments?
  - We would like to abstract over the three type constructors above.
- Solution: introduce *n*-ary *type operator variables* that can be instantiated with *n*-ary type constructors.
- Then we can form types like  $(\alpha, \beta)\phi$  where
  - $\phi$  is a 2-ary type operator variable
  - $\phi$  can be instantiated to *any* 2-ary type constructor.
- Exercise. Compare type operator instantiation with o-o inheritance.



## Motivation 2: Advanced Programming Types

---

- In Haskell, a *monad* consists of a type constructor  $M$  and two functions

$$\mathit{unit} : \alpha \rightarrow (\alpha)M$$

$$\mathit{bind} : \alpha M \rightarrow (\alpha \rightarrow \beta M) \rightarrow \beta M$$

subject to rules

$$\mathit{bind} f \ \mathit{unit} = f$$

$$\mathit{bind} \ \mathit{unit} = \mathit{id}$$

$$\mathit{bind} (\mathit{bind} g \ f) = \mathit{bind} g \ \mathit{bind} f$$

- Monads are everywhere.
- *Exercise*: define a list monad.
- Suppose we try to define a HOL predicate
$$\mathit{monad}(\mathit{unit}, \mathit{bind}) = \dots$$
- We need a 1-ary type operator variable  $\phi$  to vary over  $M$ .
  - but there is another problem.

# An Important Detail of Simple Types.

- Recall that the bound variable in a lambda abstraction is typed:

$$\lambda (v : T). t$$

- What does this mean?
  - In the abstraction body  $t$ , only occurrences of  $v$  with the (exact type)  $T$  refer to the bound variable.
  - To put it differently - the bound variable  $v$  cannot occur with different types in the abstraction body.
- This single-type restriction is a problem in the *monad* definition
  - variable *bind* occurs with different types in the axioms.
- Observation. The restriction does not apply to *constants*. Example:

$$(id: \alpha \rightarrow \alpha) \equiv \lambda (v : \alpha). v$$

We can form the term

$$id\ id = (id:(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) (id: \alpha \rightarrow \alpha)$$

For a variable  $f$ , it is impossible to form  $(ff)$ .

# Universal Types

- We can overcome the single-type restriction by introducing universal types that bind type variables ("pi-types")

$$\Pi \alpha_1 \dots \alpha_n. T$$

- Type abstraction and type application on terms:

$$\Lambda \alpha. t$$

$$t [ T ]$$

- Now bound variables can belong to a universal type.
  - We can apply them to different types in the body.
  - This allows us to define a predicate *monad* as above.
- The simply typed lambda calculus can be seen as a special case of universal types where all type variables are bound at "outermost level".
- But there is a problem.

# Girard's Paradox

- HOL extended with universal types is inconsistent [Coquand 1994].
  - Inconsistency proofs vary depending on precise logic definitions.
- [Girard, 1972] for "System U" employed "Burali-Forte Paradox" [1897]. Proof outline following [Coquand 1992]:
  - Define a mapping  $(i : \prod \alpha. \alpha \sim \alpha \rightarrow U_0)$  where
$$U_0 = ((\prod \alpha. \alpha \sim \alpha) \rightarrow bool) \rightarrow bool$$
$$i = \Lambda \alpha. \lambda (r : \alpha \sim \alpha) (p : (\prod \alpha. \alpha \sim \alpha) \rightarrow bool). (p[\alpha]) r$$
  - Define a strict well-ordering  $<_0$  on  $i$ -image of all well-orderings.
  - Consider  $\Omega = i(<_0)$
  - Deduce  $\Omega <_0 \Omega$ .
- Intuitively, the problem is caused by self-reference: In the type  $(\prod \alpha. T)$ , the type variable  $\alpha$  can again be instantiated with  $(\prod \alpha. T)$ .
  - This would be like forming a set-theoretic product over all sets. This is not allowed in set-theory.
- What to do?



# HOL2P Solution: Small Types

---

- Idea: restrict the power of universal types so that  $(\Pi\alpha. T)$  can not be applied to itself.
- Radical solution:
  - Introduce a notion of "small types" that excludes universal types
  - Restrict type abstraction and application to small types.
  - This ensures a set-theoretic semantics and consistency
- HOL2P = HOL + type operator variables  
+ abstraction over small types
- "2P" indicates "second order polymorphism"
  - Added one layer so we can abstract over normal HOL types.
- I suspect this construction is "logician folklore", but I am not aware it has been formalised before.

# HOL2P Types

$T ::= (\alpha :: \textit{small})$	small type variable
$(\alpha)$	unrestricted type variable
$(T_1, \dots, T_n)\tau$	type constructor application
$(T_1, \dots, T_n)\phi$	<b>type operator variable application</b>
$\Pi\alpha_1 \dots \alpha_n. T$	<b>universal type</b>

- *Small* types contain no universal types and no unrestricted ty vars.
  - These types correspond to normal HOL types
- Formation of universal types is restricted to small types  $\alpha_1, \dots, \alpha_n, T$
- Type substitution must respect smallness.
- Types are  *$\alpha$ -equivalent* if they are convertible by renaming of bound type variables.
- There is no "applying a universal type to a type" type
  - Types can be seen as implicitly type- $\beta$  reduced.

# HOL2P Terms

$t ::= v : T$	variable
$c : T$	constant
$t t$	application
$\lambda (v : T). t$	abstraction
$\Lambda \alpha. t$	<b>type abstraction (<math>\alpha</math> small)</b>
$\mathbf{t} [ T ]$	<b>type application (<math>T</math> small)</b>

*Formation rule:* the type of a free variable  $v$  must not contain bound type variables.

*Typing rules:*

$$t : T \vdash (\Lambda \alpha. t) : \Pi \alpha. T$$
$$(t : \Pi \alpha. S) \vdash \mathbf{t} [ T ] : S [ T \setminus \alpha ]$$

# HOL2P Rules

- All HOL inference rules apply also in HOL2P.
- Additional rules:

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

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

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

*Exercise:* What is the type of equality in HOL2P?

# Type Quantification

- Type quantification can be defined as an abbreviation in HOL2P:

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

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

- Example

$isFunctor : (\Pi \alpha \beta. . (\alpha \rightarrow \beta) \rightarrow \alpha \phi \rightarrow \beta \phi) \rightarrow bool$

$isFunctor (\varphi) = (\forall \alpha. \varphi[\alpha][\alpha] id = id )$

$\wedge (\forall \alpha \beta \gamma. \forall (g: \beta \rightarrow \gamma) (f: \alpha \rightarrow \beta ).$

$\varphi[\alpha][\gamma](g \circ f) = \varphi [\beta][\gamma] g \circ \varphi [\alpha][\beta] f )$



# HOL2P System [2007]

---

- Implementation of HOL2P theorem prover on top of existing HOL-Light system.
- Preserves compatibility with HOL-LIGHT as much as possible.
- Parsing will automatically try to insert certain type applications.
- Has been used for some relatively small applications

# Type Matching and Inference Problem

- Type operator variables make HOL2P type matching “higher order”:
- Matching problems like

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

have in general several solutions:

(1) $?F = \Lambda 'a. 'a$	$?x = (\text{nat list}) \text{ list}$
(2) $?F = \text{list}$	$?x = (\text{nat list})$
(3) $?F = \Lambda 'a. 'a \text{ list list}$	$?x = \text{nat}$
(4) $?F = \Lambda 'a. \text{nat list list}$	$?x = \text{“any type”}$

- Without guidance, the current HOL2P implementation will only find the second match.
- Users often need to add explicit type instantiations when parsing terms or applying rules that involve type operator variables.



# Related and Further Work

---

- Related work:
  - COQ based on constructive type theory
  - HOL-Omega extends HOL2P
- Further work
  - improve HOL2P tactics and type inference
  - study complexity of HOL2P algorithms
  - investigate combination with overloading/ type classes.
  - mechanically check the HOL2P semantics and the implementation of its logical core
  - run paradox proofs in an "unrestricted version of HOL2P"
  - applications
    - put category theory to use in HOL2P
    - derivation of generic programs as in "Algebra of Programming"