Church's Type Theory

Orders of a logical system

- Predicates that speak about domain objects are of 1-st order.
- Predicates that speak about objects of at most i-th order, are by themselves of $(i+1)$-th order.
- Functions that take and return domain objects are of 1-st order.
- Functions that take and return objects of at most i-th order, are by themselves of $(i+1)$-th order.

Examples

Let Real be the type of real numbers. The functions sin, cos are of type Real $\rightarrow$ Real, and are first-order.

The differentiation operator, which has type
(Real $\rightarrow$ Real) $\rightarrow$ (Real $\rightarrow$ Real) is second order. (Because it operates on first-order functions.

Let Nat be the type of natural numbers.

The formula $\forall x, y{:}\,\mathrm{Nat}\ x + y = y + x$ is first-order.

The induction principle $\forall N{:}\,\mathrm{Nat} \rightarrow \mathrm{Prop}\ \ P(0) \rightarrow (\ \forall n{:}\,\mathrm{Nat} P(n) \rightarrow P(n+1)\ ) \rightarrow \forall n{:}\,\mathrm{Nat}\ P(n)$ is second order. (Because it contains a quantifier over first-order predicates P.)

Binary Application Operator

The usual form of function/predicate application

$$f(t_1, \ldots, t_n)$$

is not suitable for HOL. In HOL, it must be possible to quantify over f, and to instantiate f with some functional expression. Therefore, f must become a term, in the same way as $t_1, \ldots, t_n$ are.

Definition: The application operator, written as $\cdot$ applies functions to arguments. The meaning of $f \cdot t$ is $f(t)$.

Currying

Functions with more than 1 arguments can be handled as follows:

$$f(t_1, \ldots, t_n)$$

can be replaced by

$$((f \cdot t_1) \cdot t_2) \ldots \cdot t_n,$$

an iterated unary function application.

Notation: We assume that $\cdot$ groups to the left. That means that $f \cdot t_1 \cdot t_2$ should be read as $(f \cdot t_1) \cdot t_2$.

Example $+ \cdot 1 \cdot 1$ equals 2.

$+ \cdot 5$ is a function that adds 5 to its argument.

$/ \cdot 1$ is the reciproke function.

More Notation

The $\cdot$ is usually omitted. Instead only parentheses are written:

The following three expressions represent $f(t_1, t_2, t_3, t_4)$:

$$(((f \cdot t_1) \cdot t_2) \cdot t_3) \cdot t_4,$$

$$f \cdot t_1 \cdot t_2 \cdot t_3 \cdot t_4,$$

$$(f\ t_1\ t_2\ t_3\ t_4).$$

The last notation is used whenever possible. Occassionaly you need to remember that the real meaning is the first notation.

λ-Notation

In the usual mathematical notation, there is no good way to construct functions. One usually writes things like:
Let $f(x)$ the function, s.t.

$$\forall x{:}X \quad f(x) = F(x).$$

Here F is some formula that defines f.

With the λ-notation, one can write

$$\lambda x{:}X \quad F(x).$$

Examples

$$\lambda n{:}\mathrm{Nat}\ (+\ n\ n), \qquad \lambda n{:}\mathrm{Nat}\ (+\ n\ 1).$$

$$\lambda n{:}\mathrm{Nat}\ 0.$$

λ-Notation (2)

If one want to apply a function of form $\lambda x{:}X\ F$ to some argument t, this can be done by substituting t for x in F.

So, we have

$$(\lambda x{:}X\ F) \cdot t = F[x := t].$$

For example,

$$(\lambda n{:}\mathrm{Nat}\ (+\ n\ n)) \cdot 3 = (+\ n\ n)[n := 3] = (+\ 3\ 3).$$

λ-Notation (3)

In HOL, λ abstraction is used for representing universal and existential quantification:

$$\forall x\ (P\ x) \Rightarrow \forall\ \lambda x\ (P\ x).$$

$$\forall x \exists y\ (P\ x\ y) \Rightarrow \forall\ (\lambda x\ \exists\ (\lambda y\ (P\ x\ y)\ )).$$

Types Constructed with $\rightarrow$

A lot of nonsense can be written down, for example

$$(+ \cdot +), \text{ or } (4 \ + \ 2).$$

Types can exclude at least some of the possible nonsense.

The operator $\rightarrow$ is used for constructing function types. $X \rightarrow Y$ is the type of functions from X to Y.

The $\rightarrow$ groups to the right. That means that $X_1 \rightarrow X_2 \rightarrow X_3$ should be read as $X_1 \rightarrow (X_2 \rightarrow X_3)$.

Examples for $\rightarrow$

The function $+$ has type $\mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat}$.
The function $\lambda n{:}\mathrm{Nat}\ (+\ n\ 1)$ has type $\mathrm{Nat} \rightarrow \mathrm{Nat}$. The differentiation operator has type $(\mathrm{Real} \rightarrow \mathrm{Real}) \rightarrow (\mathrm{Real} \rightarrow \mathrm{Real})$.

One can also have operators of type $\mathrm{Type} \rightarrow \mathrm{Type}$. An example of such an operator is the List operator.

$(+ \cdot +)$ is not well-typed, because $+$ has type $\mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat}$, which can only be applied to objects of type Nat.

Ordered Pairs

The HOL system has an additional operator for constructing ordered pairs, which is called ,:

It is curried by itself, so that (a, b) is represented by

$$(\ ,\cdot\ a)\cdot b.$$

For the pairing operator, a new type constructor has to be introduced:

If $a{:}\,A$, and $b{:}\,B$, then a, b has type $A \times B$.

Polymorphism/Type Variables

HOL has built-in type Bool. All other types are user defined, or defined in libraries.

In addition, there are type variables, which enable polymorphism.

For example, equality has type $\alpha \to \alpha \to \text{Bool}$.

In concrete applications of $=$, α can be instantiated to a concrete type.

For example, in $1 + 1 = 2$, it will be instantiated to Int.

The pair constructor has type $\alpha \to \beta \to (\alpha \times \beta)$. In the concrete case of $(4, \mathbf{T})$, it has type $\text{Int} \to \text{Bool} \to (\text{Int} \times \text{Bool})$.

Hol Terms

We give a formal definition of the terms of HOL. Note that formula and terms are not distinguished. A formula is simply a term that has type Bool.

Definition: The terms of HOL are recursively defined as follows:

- A variable is a term.
- If f and t are terms, then $f \cdot t$ is a term.
- If x is a variable, and t is a term, then $\lambda x\ t$ is a term.
- If t_1, t_2 are terms, then $\ t_1, t_2\ $ is a term.

HOL Types

Definition: The types of HOL are recursively defined as follows:

- A type variable is a type.
- If A and B are types, then $A \to B$ is a type.
- If A and B are types, then $A \times B$ is a type.

Free and Bound Variables

In a HOL term of form

$$\lambda x\ t,$$

the occurrences of x inside t are bound by λx.

In the term

$$(\lambda x\ y\ (f\ x\ y))(g\ x\ y),$$

the first occurrences of x and y, in $(f\ x\ y)$ are bound occurrences. The second occurrences, in $(g\ x\ y)$ are free occurrences. The occurrences of f and g are free.

α-Equivalence

The notion of α-equivalence is the same as for first-order formulas.

The formulas $\lambda x\ (+\ x\ x)$ and $\lambda n\ (+\ n\ n)$ are α-equivalent.

The formulas $\lambda xy\ (f\ x\ y)$ and $\lambda yx\ (f\ x\ y)$ are not α-equivalent, but $\lambda xy\ (f\ x\ y)$ and $\lambda yx\ (f\ y\ x)$ are.

Substitution

Substitution is defined in essentially the same way as for first-order logic.

$t[x := u]$ is the term that one obtains when all free occurrences of x are replaced by u.

If capture occurs, then t has to be replaced by another α-variant before replacing x by u.

Examples of Substitution

$$\lambda n\ (+\ n\ m)[m := 1] \text{ equals } \lambda n\ (+\ n\ 1),$$

$$\lambda n\ (+\ n\ m)[m := (+\ m\ 1)] \text{ equals } \lambda n\ (+\ n\ (+\ m\ 1)),$$

$$\lambda n\ (+\ n\ m)[m := (+\ n\ 1)] \text{ equals } \lambda z\ (+\ z\ (+\ n\ 1)).$$

If one would not α-rename n, the result would be

$$\lambda n\ (+\ n\ (+\ n\ 1)),$$

which is a completely different function.

β-Equivalence, β-Reduction

Let u be a λ-term which contains a subterm of form

$$(\lambda x\ f) \cdot t$$

Then $u[\ (\ \lambda x\ f) \cdot t\ ]$ and $u[\ f[x := t]\ ]$ are called β-equivalent.

Applying this equivalence from left-to-right is called β-reduction.

$$(\lambda n\ (+\ n\ n)) \cdot 2 \equiv_\beta (+\ 2\ 2).$$

$$(+\ 1\ ((\lambda n\ (+\ n\ n))\ 7)) \equiv_\beta (+\ 1\ (+\ 7\ 7)).$$

Sequents

The HOL logic is based on sequents.

Definition: A sequent is an object of form $A_1, \ldots, A_p \vdash B$.

The meaning is: $A_1, \ldots, A_p$ imply B.

Type Checking of Sequents

In a sequent, all $A_1, \ldots, A_p$, and B must have type Bool.

Typechecking is done automatically, and the algorithm uses unification to instantiate type variables, when necessary.

1. Look up the types for identifiers that have a definition. If a defined identifier is polymorphic (has variables in its type), then each occurrence receives a fresh set of variables.
2. Assign to every undeclared identifier a fresh type variable α_i. When an undeclared variable occurs more than once, the different occurrences receive the same α_i.
3. Typecheck the sequent and unify type variables if necessary.

When finished, the algorithm either has constructed a unifying substitution Θ, or it has failed.

Examples of Type Checking

We check the sequent $a = b \vdash b = c$. First write it in Curried form:

$$(=\ \ a\ b) \vdash (=\ \ b\ c).$$

Look up the types of the symbols:

$$(\ (=\!:\alpha \to \alpha \to \text{Bool})\ (a\!:\beta)\ (b\!:\gamma)\ ) \vdash (\ (=\!:\delta \to \delta \to \text{Bool})\ (b\!:\gamma)\ (c\!:\zeta)\ ).$$

Then unify $\alpha \Rightarrow \beta,\ \gamma \Rightarrow \beta,\ \delta \Rightarrow \beta,\ \zeta \Rightarrow \beta$.

Examples of Type Checking (2)

We check the sequent $\vdash f(a) = f(4)$. First write it in the form

$$\vdash (= \ (f\ a)\ (f\ 4)).$$

Look up the types of the symbols:

$$\vdash (\ (=:\alpha \to \alpha \to \text{Bool})\ (\ (f{:}\beta)\ (a{:}\gamma)\ )\ (\ (f{:}\beta)\ (4{:}\text{Int})\ )\ ).$$

The identifier $f{:}\beta$ is applied on $a{:}\gamma$. This means that β must have form $\beta_1 \to \beta_2$ and that $\beta_1 = \gamma$. Because $=$ has type $\alpha \to \alpha \to \text{Bool}$, we see that $\alpha = \beta_2$.

The result is

$$\vdash (\ (=:\alpha \to \alpha \to \text{Bool})\ (\ (f{:}\gamma \to \alpha)\ (a{:}\gamma)\ )\ (\ (f{:}\gamma \to \alpha)\ (4{:}\text{Int})\ )\ ).$$

Looking at the second term, we see that $\gamma = \text{Int}$. The sequent is accepted, and the variable α remains open.

Theorems/Definitions

HOL has some data structure that stores proven theorems.

There is no distinction between definitions and theorems. If c is an unused variable, and t is a term without free (term or type) variables, then it is possible to add the theorem

$$\vdash c = t.$$

Type Definitions

Primitive Inference Rules for Sequents

Basic Equality Rules:

$$\text{REFL} \quad \frac{}{\vdash t = t}$$

$$\text{TRANS} \quad \frac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}$$

$$\text{MK\_COMB} \quad \frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash (s \cdot u) = (t \cdot v)}$$

Rules involving λ

If x is not free in an assumption in Γ, then

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

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

Note that this is a very restricted form of β-equivalence. We will later see that all other forms are derivable.

Logical Rules

$$\text{ASSUME} \quad \frac{}{p \vdash p}$$

$$\text{EQ\_MP} \quad \frac{\Gamma \vdash p = q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

$$\text{DEDUCT\_ANTISYM\_RULE} \quad \frac{\Gamma \vdash p \qquad \Delta \vdash q}{(\Gamma \backslash \{q\}) \cup (\Delta \backslash \{p\}) \vdash p = q}$$

Instantiation Rules

If x is an identifier that does not occur in a theorem, then

$$\text{INST} \quad \frac{\Gamma \vdash A}{\Gamma[x := t] \vdash A[x := t]}$$

If α is a type variable, then

$$\text{INST\_TYPE} \quad \frac{\Gamma \vdash A}{\Gamma[\alpha := T] \vdash A[\alpha := T]}$$

Definitions of Logical Operators

$\top = (\lambda x.\ x) = (\lambda x.\ x)$,

$\wedge = \lambda p.\ \lambda q.\ \lambda f.\ (f\ p\ q) = \lambda f.\ (f\ \top\ \top)$,

$\rightarrow$ (the logical operator) $= \lambda p.\ \lambda q.\ (p \wedge q) = p$,

$\forall = \lambda P.\ P = \lambda P.\ \top$,

$\exists = \lambda P.\ \forall Q.\ (\forall x.\ (P\ x)) \rightarrow Q) \rightarrow Q$,

$\vee = \lambda P.\ \lambda Q.\ \forall R.\ (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R$,

$\bot = \forall P.\ P$,

$\neg = \lambda P.\ (P \rightarrow \bot)$,

$\exists! = \lambda P.\ (\exists\ P) \wedge \forall x.y.\ (P\ x) \wedge (P\ y) \rightarrow (x = y)$,

Axioms of HOL

Extensionality:

$$\vdash (\lambda x.\ (t\ x)) = t.$$

Epsilon: (Hilbert choice operator)

$$\vdash \forall x.\ ((P\ x) \rightarrow (P\ (\epsilon\ (\lambda x.\ (P\ x))))\ ).$$

There is one declared type Ind.

There are infinitely many individuals:

$$\vdash \exists f{:}\mathrm{Ind} \rightarrow \mathrm{Ind}.\ (\forall x1.x2.\ (f\ x_1) = (f\ x_2) \rightarrow x_1 = x_2) \wedge \neg(\forall y.\exists x.y = (f\ x)).$$

Type Constructors

Higher-order logic is very expressive. Most of mathematics can be expressed in HOL, when appropriate axioms are used.

On the other hand, it is fairly easy to implement a checker for it, because there are not many logical rules.

Because of these features, it is used in systems in which the correctness of programs, or mathematical proofs can be checked. Such systems are called proof assistents.