Higher-Order Logic

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.

Application Operator

The usual method of function/predicate application

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

is not good enough for HOL. The reason is that f cannot be the result of a function itself.

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

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

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

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

for this function.

Examples

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

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

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.

With $\Rightarrow$ one can construct 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 to $\Rightarrow$

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.

Polymorphic Types constructed by Π

The $\Rightarrow$ cannot express all types that one would like to express. Consider the type of lists. For every type X, one can define the type (List X), which is the type of (finite) lists over X.

Lists are constructed by nil and the cons-operator. Lists over the natural numbers can be constructed by the functions $\mathrm{nil}_{\mathrm{Nat}}$ of type (List Nat) and $\mathrm{cons}_{\mathrm{Nat}}$ of type Nat $\Rightarrow$ (List Nat) $\Rightarrow$ (List Nat).

Similarly, lists over real numbers can be constructed by the functions $\mathrm{nil}_{\mathrm{Real}}$ and $\mathrm{cons}_{\mathrm{Real}}$.

One does not want to construct a nil_X, and cons_X for every type X.

One would want to construct universal functions nil and cons, that, given a type X, produce the nil_X and cons_X for that type.

Both nil and cons are functions, with the property that the type of the result, depends on the type of the arguments. Therefore, their types must be dependent types.

The Π-operator is used for constructing dependent types, we have

$$\text{nil}\colon (\Pi \text{X}\colon \text{Type } (\text{List X}) \ ),$$

$$\text{cons}\colon (\Pi \text{X}\colon \text{Type } \ \text{X} \Rightarrow (\text{List X}) \Rightarrow (\text{List X}) \ ).$$

Relation between $\Rightarrow$ and Π.

$f{:}\, X \Rightarrow Y$ means that f wants to see an X and returns a Y.

$f{:}\, \Pi x{:}\, X\ Y$ means that f wants to see an x of type X, and that for each x of type X it returns an element of type Y. Because x can occur in Y, the return-type can be different for different x's.

In case that x does not occur in Y, $\Pi x{:}\, X\ Y$ and $X \Rightarrow Y$ are the same.

Because of this, $X \Rightarrow Y$ can be seen as syntactic sugar for $\Pi x{:}\, X\ Y$.

The following types are the same:

$$\mathrm{Nat} \Rightarrow \mathrm{Nat} \Rightarrow \mathrm{Nat},$$

$$\Pi n{:}\,\mathrm{Nat}\ (\mathrm{Nat} \Rightarrow \mathrm{Nat}),$$

$$\mathrm{Nat} \Rightarrow \Pi m{:}\,\mathrm{Nat}\ \mathrm{Nat},$$

$$\Pi n{:}\,\mathrm{Nat}\ (\Pi m{:}\,\mathrm{Nat}\ \mathrm{Nat}).$$

Declarations and Definitions

Before a variable can be used, it has to be declared or defined. A declaration specifies only the type of a variable. A definition specifies also the value.

A declaration is a statement of the form $x{:}\,X$. The meaning is x has type X. It must be the case that x is a variable, and X is a type.

Examples of declarations are

Nat: Type
0: Nat
+: Nat ⇒ Nat ⇒ Nat

List: Type ⇒ Type
nil: ΠX: Type (List X)
cons: ΠX: Type X ⇒ (List X) ⇒ (List X)

Definitions

A definition has form $x := y{:}Y$. It must be the case that x is a variable, and y is a term of type Y. It is not allowed that x occurs in y or Y.

Examples of definitions are:

$1 := (s\ 0){:}\mathrm{Nat}$,
$2 := (s\ (s\ 0)){:}\mathrm{Nat}$
$\bot := \forall F{:}\mathrm{Prop}\ \ F$
$\neg := \lambda F{:}\mathrm{Prop}\ (F \to \bot)$

Logical Operators

Higher Order Logic has only two logical operators, namely implication and universal quantification.

We use the standard notation for the logical operators: $F_1 \to F_2$ means F_1 implies F_2.

$\forall x{:}\, X\ F$ means: for all x of type X, the formula F.

It will turn out that the other logical operators can be defined.

λ-terms

Definition: We define by recursion what λ-terms are:

- An variable is a λ-term.
- If f and t are λ-terms, then $f \cdot t$ is a λ-term.
- If x is a variable, X is a λ-term, and y is a λ-term, then $\lambda x{:}\, X\ y$ is a λ-term.
- If x is a variable, X is a λ-term, and Y is a λ-term, then $\Pi x{:}\, X\ Y$ is a λ-term.

Note: Variables, atoms, names, identifiers are the same.

Contexts

A context is a list of declarations and definitions. Formally a context C is a sequence of the form

$$D_1, \ldots, D_n, \quad n \geq 0,$$

where each D_i is either of form $x{:}\,X$ or $x := y{:}\,X$.

It is not allowed to redefine/redeclare a variable.

We write $C \vdash x{:}\,X$ if term x has type X in context C.

Example:

$$\mathrm{Nat}{:}\,\mathrm{Type},\ s{:}\,\mathrm{Nat} \Rightarrow \mathrm{Nat} \vdash \lambda n{:}\,\mathrm{Nat}\ (s\ (s\ n)){:}\,(\mathrm{Nat} \Rightarrow \mathrm{Nat}).$$

Free and Bound Variables

In the λ-term

$$\lambda x{:}X\ t,$$

we call x the bound variable, X the type, and t the body of the term.

The λ binds all the occurrences of x in its body. (Not in its type!)

A variable, occurring somewhere in a λ-term is bound if there is a λ that binds it. Otherwise it is free.

α-Variants

Informally, two terms t_1 and t_2 are α-variants, if they have the same meaning. Two λ-terms t_1 and t_2 are α-variants if the following are true:

1. If somewhere in t_1 there is a subterm built by $\cdot$, then at the corresponding position in t_2, there is also a subterm built by $\cdot$.
2. If somewhere in t_1 there is a subterm built by λ, then at the corresponding position in t_2, there is also a subterm built by λ.
3. If a variable occurring somewhere in t_1 is free in t_1, then at the corresponding position in t_2 occurs the same variable, and it is also free in t_2.
4. If a variable x occurring somewhere in t_1 is not free, then at the corresponding position in t_2, there is also a variable, which is also not free, and both are bound by λ's at the same positions.

Substitution

Substitution is the essentially the replacement of all occurrences of a free variable x of a term t by some term u.

BUT Free variable of u may get caught by λ's inside t.

THEREFORE First replace t by an α-variant, where this will not happen. Such a variant always exists, when you have infinitely many variables.

The notation is

$$t[x := u].$$

Examples of Substitution

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

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

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

The term

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

would be a totally different function.

Equivalences

Based on the intended meanings of the λ-terms, a couple of equivalence relations can be defined. They are usually called α, β, δ, and η-equivalence. (Nobody seems to worry about the missing γ-equivalence)

One of them we defined already, namely α-equivalence:

$$t_1 \equiv_\alpha t_2$$

if t_1 can be obtained from t_2 by renaming bound variables.

β and η-equivalence

Another type of equivalence is β-equivalence, which is defined as follows:

$$t_1 \equiv_\beta t_2,$$

if t_2 can be obtained from t_1 by replacing a subterm of the form

$$(\lambda x{:}X\ f) \cdot t$$

by

$$f[x := t].$$

η-equivalence is defined as follows:
$t_1 \equiv_\eta t_2$ if t_2 is obtained from t_1 by replacing a subterm of the form

$$(\lambda x{:}X\ (f\ x))$$

by f. It must be the case that x is not free in f.

δ-equivalence

The last equivalence is δ-equivalence. It is based on expansion of definitions. Because of this, contrary to the other equivalences, it depends on a context C.

If context C contains a definition $x := y{:}Y$, then

$$t_1 \equiv_\delta t_1[x := y].$$

If x is defined as y, then it is allowed to replace x by y.

$\alpha, \beta, \delta, \eta$-equivalence

We write

$$C \vdash t_1 \equiv_{\alpha,\beta,\delta,\eta} t_2$$

if t_1 can be obtained from t_2 by finitely often applying the $\alpha, \beta, \delta, \eta$ -replacement rules, in either direction, and on every subterm.

Typing Rules

We assume that the following types are given:

Form : The type of formulae.
Type : The type of types.
Kind : The type of Form and Type, needed for technical reasons.

The objects Form, Type, Kind are called sorts. We write S for the set of sorts, that is $S = \{\text{Form}, \text{Type}, \text{Kind}\}$.

We now need the following:

1. Rules for determining the type of a λ-term if there exists one.
2. Rules for determining whether or not a context is well-formed.

Rules for well-formedness of Contexts

The empty context is well-formed.

DECL: If C is well-formed, x is a variable, not occurring in C,
$C \vdash X{:}T$, where T is a sort, then

$$C, \quad x{:}X$$

is well-formed.

DEF: If C is well-formed,
$C \vdash y{:}Y$, and
x is a variable, not occurring in C, then

$$C, \quad x := y{:}Y$$

is well-formed.

Rules for determining the type of a λ-term

In the rules, we implicitly assume that all contexts are well-formed.

SORT: For every context C, we have

$C \vdash \mathrm{Type}\colon \mathrm{Kind}$ and

$C \vdash \mathrm{Form}\colon \mathrm{Kind}$.

AXIOM: $C,\ x\colon X \vdash x\colon X$.

$C,\ x := y\colon Y \vdash x\colon Y$.

WEAKENING: If $C \vdash x\colon X$, then $C, D \vdash x\colon X$, for every definition or declaration D.

APPL: If $C \vdash t{:}\, X$, and

$C \vdash f{:}\, \Pi x{:}\, X\ Y$, then $C \vdash (f \cdot t){:}\, (Y[x := t])$.

LAMBDA: If $C,\ x{:}\, X \vdash y{:}\, Y$, then

$C \vdash (\lambda x{:}\, X\ y){:}\, (\Pi x{:}\, X\ Y)$.

Note that $C, x{:}\, X$ has to be well-formed.

PI: If $C, x{:}X \vdash Y{:}T$, where T is a sort, then $C \vdash (\Pi x{:}X\ Y){:}T$.
(Remember that $C, x{:}X$ must be well-formed)

EQUIV: If $C \vdash x{:}X_1$,
$C \vdash X_1 \equiv_{\alpha\beta\delta\eta} X_2$, and
$C \vdash X_2{:}T$, with T a sort, then $C \vdash x{:}X_2$.

Sequent Calculus for Higher-Order Logic

Sequents have form $\Gamma \vdash_C \Delta$, where C is a well-formed context, such that

- C contains declarations $\rightarrow : \mathrm{Form} \Rightarrow \mathrm{Form}$
  $\forall : \Pi X : \mathrm{Type}\ (X \Rightarrow \mathrm{Form}) \Rightarrow \mathrm{Form}$

- Each element of Γ, Δ has type Form in C.

Axioms:

$$\text{(axiom)}\ \frac{}{A \vdash_C A}$$

on the condition that $C \vdash A\colon \mathrm{Form}$.

Structural Rules:

$$\text{(weakening left)}\ \frac{\Gamma \vdash_C \Delta}{\Gamma, A \vdash_C \Delta} \qquad \text{(weakening right)}\ \frac{\Gamma \vdash_C \Delta}{\Gamma \vdash_C \Delta, A}$$

$$\text{(contraction left)}\ \frac{\Gamma, A, A \vdash_C \Delta}{\Gamma, A \vdash_C \Delta} \qquad \text{(contraction right)}\ \frac{\Gamma \vdash_C \Delta, A, A}{\Gamma \vdash_C \Delta, A}$$

Rules for $\rightarrow$:

$$(\rightarrow \text{-left }) \ \frac{\Gamma \vdash_C \Delta, A \quad \Gamma, B \vdash_C \Delta}{\Gamma, A \rightarrow B \vdash_C \Delta}$$

$$(\rightarrow \text{-right }) \ \frac{\Gamma, A \vdash_C \Delta, B}{\Gamma \vdash_C \Delta, A \rightarrow B}$$

Rules for $\forall$:

$$(\forall\text{-left})\ \frac{\Gamma,\ F[x := t] \vdash_C \Delta}{\Gamma,\ \forall x{:}X\ F \vdash_C \Delta}$$

It must be the case that $C \vdash t{:}X$.

$$(\forall\text{-right})\ \frac{\Gamma \vdash_{C'} \Delta,\ F[x := y]}{\Gamma \vdash_C \Delta,\ \forall x{:}X\ F}$$

The y must be a variable that is not free in $\Gamma, \Delta,\ \forall x{:}X\ F$.

C' has form $C,\ y{:}X$.

Natural Deduction for Higher-Order Logic

In natural deduction for HOL, the context and the assumptions are mixed.

$\rightarrow$-introduction:

$$\begin{array}{|l} \begin{array}{|l} A \\ \hline \vdots \\ B \end{array} \\ A \rightarrow B \end{array}$$

$\rightarrow$-elimination:

A

$\ldots$

$A \to B$

$\ldots$

B

$\forall$-introduction:

$$\begin{array}{|l}
\quad \begin{array}{|l} y{:}X \\ \hline \ldots \\ F[x := y] \end{array} \\
\forall x{:}X\ F.
\end{array}$$

$\forall$-elimination:

$$\begin{array}{|l}
\forall x{:}X\ F \\
\ldots \\
F[x := t]
\end{array}$$

Term t must have type X at the point where $F[x := t]$ is introduced.