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.

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.)

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,

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

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 to $\rightarrow$

The function $+$ has type $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$.

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

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

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

Polymorphic Types constructed by Π

The $\rightarrow$ cannot express all types that one wants 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 $\mathrm{cons}_{\mathrm{Real}}$, which have type (List Real) and Real $\rightarrow$ (List Real).

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

Using polymorphism, one can define a single nil and cons-operator that can be applied on every type. The first argument of nil or cons is the type over which the list is being constructed.

In order to be able give a type to nil and cons, we introduce a new type constructor Π. Then

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

$$\mathrm{cons}\colon (\Pi \mathrm{X}\colon \mathrm{Type}\ \ \mathrm{X} \rightarrow (\mathrm{List}\ \mathrm{X}) \rightarrow (\mathrm{List}\ \mathrm{X})\ ).$$

Relation between $\rightarrow$ and Π.

$f\colon X \rightarrow Y$ means that f expects a term t of type X and returns an object of type Y.

$f\colon \Pi x\colon X\ Y$ means that f expects a term t of type X and that, for each t, it returns an object of type $Y[x := t]$. Because the actual type $Y[x := t]$ may depend on t, such a type is called dependent type.

In case that x does not occur in Y, $\quad \Pi x\colon X\ Y$ and $X \rightarrow Y$ are the same type.

Because of this, $X \rightarrow Y$ can be seen as syntactic sugar for $\Pi x\colon X\ Y$ for the case where x is not free in Y.

Examples

The type of (nil Nat) equals (List X)[X := Nat] = (List Nat).

The type of (cons Real) equals

$$(X \to (\text{List } X) \to (\text{List } X))[X := \text{Real}] =$$

$$\text{Real} \to (\text{List Real}) \to (\text{List Real}).$$

The following types are the same:

$$\text{Nat} \to \text{Nat} \to \text{Nat},$$

$$\Pi n{:}\,\text{Nat } (\text{Nat} \to \text{Nat}),$$

$$\text{Nat} \to \Pi m{:}\,\text{Nat Nat},$$

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

λ-terms

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

- A 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.

Free and Bound Variables

In a λ-term of form

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

the occurrences of x in t are bound by λx.

If there are occurrences of x in X, these are not bound by this λx.

α-Equivalence

The notion of α-equivalence is the same as for first-order formulas. One only has to be aware of the fact that $\lambda x{:}X\ t$ does not bind the occurrences of x in X.

$\lambda x{:}\mathrm{Nat}(+\ x\ x)$ and $\lambda n{:}\mathrm{Nat}(+\ n\ n)$ are α-equivalent.

The formulas $\lambda x{:}(\mathrm{List}\ x)\ (f\ x)$ and $\lambda y{:}(\mathrm{List}\ y)\ (f\ y)$ are not α-equivalent, but $\lambda x{:}(\mathrm{List}\ x)\ (f\ x)$ and $\lambda y{:}(\mathrm{List}\ x)\ (f\ y)$ 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

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

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

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

If one would not replace n, the result would be

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

which is a completely different function.

β-Equivalence, β-Reduction

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

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

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

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

$$(\lambda n{:}\mathrm{Nat}\ (+\ n\ n)) \cdot 2 \equiv_\beta (+\ 2\ 2).$$

$$(+\ 1\ ((\lambda n{:}\mathrm{Nat}\ (+\ n\ n))\ 7)) \equiv_\beta (+\ 1\ (+\ 7\ 7)).$$

δ-Equivalence, δ-Reduction

Suppose that some identifier x is defined as a term t.

Let u be some other term. Then u and $[x := t]$ are called δ-equivalent. Applying this equivalence from left-to-right is called δ-reduction.

So, δ-reduction is the unpacking of definitions.

If double is defined as $\lambda n{:}\,\mathrm{Nat}\ (+\ n\ n)$, then

$$(\mathrm{double}\ 3) \equiv_\delta ((\lambda n{:}\,\mathrm{Nat}\ (+\ n\ n))\ 3) \equiv_\beta (+\ 3\ 3).$$

α, β, δ-Equivalence

We write

$$t_1 \equiv_{\alpha,\beta,\delta} t_2$$

if t_1 can be obtained from t_2 by finitely often applying α, β or δ-equivalence.

It can be shown that α, β, δ-equivalence can be tested as follows:

1. Reduce t_1, t_2 using β, δ-reduction. If t_1, t_2 are well-typed, reduction is guaranteed to terminate.
2. Check if the results are α-equivalent.

Contexts

Our aim is to define natural deduction for higher-order logic.

In natural deduction for first-order logic, one starts with couple of assumptions, and then applies forward reasoning, starting from the assumptions. Sometimes an assumption is dropped, in order to introduce a $\neg$, or $\rightarrow$.

In higher-order logic, one also has to consider declarations of primitive notions and definitions.

It is convenient to collect all these three things (assumptions, declarations, and definitions) in a single container that is called context.

So, at each point in the natural deduction proof, the list of assumptions, declarations and definitions that apply at this point, is called the context at this point.

Contexts (2)

The letter Γ will be used for denoting contexts.

We write $\Gamma \vdash t{:}T$ if, using the type checking rules, together with the declarations and definitions in Γ, it can be deduced that t has type T.

(We will give the rules for determining the type of a term later)

A context is a list of declarations, definitions and assumptions.

That the order is important can be seen from the following examples:

$$\mathrm{Nat}{:}\,\mathrm{Type},\ \ 0{:}\,\mathrm{Nat},$$

(It would be wrong to define 0 as Nat, before Nat is defined as type)

$$0{:}\,\mathrm{Nat},\ \ s{:}\,\mathrm{Nat} \to \mathrm{Nat},\ \ 1 := (s\ 0){:}\,\mathrm{Nat}.$$

(It would be wrong to define $1 := (s\ 0)$, before s and 0 are declared)

Declarations

A declaration is a statement of the form $x{:}X$. The meaning is x has type X.

A declaration $x{:}X$ can be appended to a context Γ if x is a variable, that does not have a declaration/definition in Γ.

It must be the case that

1. X has form $\Pi y_1{:}Y_1 \cdots y_n{:}Y_n$ Type, with $n \geq 0$. The context $\Gamma, y_1{:}Y_1, \ldots, y_n{:}Y_n$ must be well-formed.

2. X has form $\Pi y_1{:}Y_1 \cdots y_n{:}Y_n$ Prop, with $n \geq 0$. The context $\Gamma, y_1{:}Y_1, \ldots, y_n{:}Y_n$ must be well-formed.

3. X is a user defined type. In that case, it must be the case that $\Gamma \vdash X{:}\mathrm{Type}$.

Declarations (2)

Examples of declarations are

$\mathrm{Nat}\colon \mathrm{Type}$
$0\colon \mathrm{Nat}$
$+\colon \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$

$\mathrm{List}\colon \mathrm{Type} \to \mathrm{Type}$
$\mathrm{nil}\colon \Pi\mathrm{X}\colon \mathrm{Type}\ (\mathrm{List\ X})$
$\mathrm{cons}\colon \Pi\mathrm{X}\colon \mathrm{Type}\ \mathrm{X} \to (\mathrm{List\ X}) \to (\mathrm{List\ X})$

($\mathrm{Type} \to \mathrm{Type}$) is an abbreviation of $\Pi x\colon \mathrm{Type}\ \mathrm{Type}$)

Definitions

A definition has form $x := y{:}Y$. The meaning is: x is defined as y, which has type Y.

A definition can be appended to a context Γ if x is a variable that does not have a declaration/definition in Γ, and $\Gamma \vdash y{:}Y$.

Examples of definitions are:

$1 := (s\ 0){:}\,\mathrm{Nat}$,

$2 := (s\ (s\ 0)){:}\,\mathrm{Nat}$

$\bot := \forall F{:}\,\mathrm{Prop}\ \ F{:}\quad \mathrm{Prop}$,

$\neg := \lambda F{:}\,\mathrm{Prop}\ (F \rightarrow \bot){:}\quad \mathrm{Prop} \rightarrow \mathrm{Prop}$.

Assumptions

Higher Order Logic has two logical operators. These are implication and universal quantification.

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

(It will be clear from the context whether $\longrightarrow$ is the function type constructor, or the logical implication operator)

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

The other logical operators are definable.

An assumption F can be appended to a context if $\Gamma \vdash F{:}\mathrm{Prop}$.

Typing Rules

ASSUMPTION: If there is a declaration $x{:}X$ in Γ, then

$$\Gamma \vdash x{:}X.$$

DEFINITION: If there is a definition $x := y{:}Y$ in Γ, then

$$\Gamma \vdash x{:}Y.$$

EQUIVALENCE: If

$$\Gamma \vdash t_1{:}X,$$

and $t_1 \equiv_{\alpha,\beta,\delta} t_2$, using the definitions in Γ for the δ-equivalence, then

$$\Gamma \vdash t_2{:}X.$$

Typing Rules (2)

APPLICATION: If

$$\Gamma \vdash t{:}\, X,$$

and

$$\Gamma \vdash f{:}\, \Pi x{:}\, X\ \ Y,$$

then

$$\Gamma \vdash (f \cdot t){:}\, (Y[x := t]).$$

LAMBDA: If

$$\Gamma,\ x{:}\, X \vdash y{:}\, Y,$$

then

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

Typing Rules (3)

PI: If

$$\Gamma,\ x{:}\,X \vdash T{:}\,\mathrm{Type},$$

then

$$\Gamma \vdash (\ \Pi x{:}\,X\ \ T\ ){:}\,\mathrm{Type}.$$

Typing Rules (4)

IMPLICATION: If

$$\Gamma \vdash A\colon \mathrm{Prop},$$

and

$$\Gamma \vdash B\colon \mathrm{Prop},$$

then

$$\Gamma \vdash A \to B\colon \mathrm{Prop}.$$

FORALL: If

$$\Gamma, x\colon X \vdash F\colon \mathrm{Prop},$$

then

$$\Gamma \vdash \forall x\colon X\ \ F\colon \mathrm{Prop}.$$

Natural Deduction for Higher-Order Logic

In a natural deduction proof, at each point in the proof, the context has to be well-formed.

$\rightarrow$-introduction:

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

$\rightarrow$-elimination:

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

$\forall$-introduction:

$$\begin{array}{|l} \begin{array}{|l} x{:}\,X \\ \hline \ldots \\ F \end{array} \\ \forall x{:}\,X\ \ F. \end{array}$$

$\forall$-elimination:

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

Term t must have type X in the context at the point where the rule is applied.

α, β, δ-equivalence:

F_1

$\ldots$

F_2

At the point where the rule is applied, it must be the case that Γ implies that $F_1 \equiv_{\alpha,\beta,\delta} F_2$.

Definitions of Logical Operators

We list the definitions of the standard operators:

$$\bot := \forall P{:}\,\mathrm{Prop}\;\; P{:}\,\mathrm{Prop}$$

$$\top := \forall P{:}\,\mathrm{Prop}\;\; P \to P{:}\,\mathrm{Prop}$$

$$\neg := \lambda P{:}\;\mathrm{Prop}\;(P \to \bot){:}\;\mathrm{Prop} \to \mathrm{Prop}$$

$$\vee := \lambda P, Q{:}\,\mathrm{Prop}$$
$$\forall R{:}\,\mathrm{Prop}\;(P \to R) \to (Q \to R) \to R{:}\,\mathrm{Prop} \to \mathrm{Prop} \to \mathrm{Prop}$$

$$\wedge := \lambda P, Q{:}\,\mathrm{Prop}$$
$$\forall R{:}\,\mathrm{Prop}\;(P \to Q \to R) \to R{:}\,\mathrm{Prop} \to \mathrm{Prop} \to \mathrm{Prop}$$

$$\leftrightarrow := \lambda P, Q{:}\,\mathrm{Prop}\;(P \to Q) \wedge (Q \to P){:}\,\mathrm{Prop} \to \mathrm{Prop} \to \mathrm{Prop}$$

$$\exists := \lambda D{:}\,\mathrm{Type}\ \lambda P{:}\,D \rightarrow \mathrm{Prop}$$
$$\forall Q{:}\,\mathrm{Prop}\ (\forall d{:}\,D\ (P\ d) \rightarrow Q) \rightarrow Q$$
$$:\Pi D{:}\,\mathrm{Type}\ \Pi P{:}\,D \rightarrow \mathrm{Prop}\ (P \rightarrow \mathrm{Prop}) \rightarrow \mathrm{Prop}$$

$$= := \lambda D{:}\,\mathrm{Type}\ \lambda d_1, d_2{:}\,D$$
$$\forall P{:}\,D \rightarrow \mathrm{Prop}\ (P\ d_1) \rightarrow (P\ d_2){:}\,\Pi D{:}\,\mathrm{Type}\ D \rightarrow D \rightarrow \mathrm{Prop}$$

An alternative definition of equality is the following:

$$= := \lambda D{:}\,\mathrm{Type}\ \lambda d_1, d_2{:}\,D$$
$$\forall P{:}\,D \rightarrow D \rightarrow \mathrm{Prop}\ (\ \forall d{:}\,D\ (P\ d\ d)\ ) \rightarrow (P\ d_1\ d_2){:}$$
$$\Pi D{:}\,\mathrm{Type} D \rightarrow D \rightarrow \mathrm{Prop}$$

Intuitionistic vs. Classical

The situation is the same as with first order logic:

Natural Deduction defines intuitionistic Higher Order Logic.

One can obtain classical Higher Order Logic from this by adding the law of excluded middle:

$$\forall F{:}\mathrm{Prop}\ F \vee (\neg F),$$

or double negation elimination:

$$\forall F{:}\mathrm{Prop}\ (\neg\neg F \to F).$$

Applications

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.