# Higher-Order Types and Lambda-Expressions



## 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 of constructs involving higher-order are:

induction: $\forall P\ P(0) \wedge \forall n{:}\mathrm{Nat}\ P(n) \rightarrow P(n+1) \rightarrow \forall n{:}\mathrm{Nat}\ P(n)$.

Differentiation: $(x.y)' = x' \cdot y + x \cdot y'$.

Statements involving functions: Every function that is 1-time differentiable in the complex plane is infinitely often differentiable.

Abstract Mathematical Structures Interpretations, number fields, lattices, groups.



## $\lambda$-notation

In usual mathematical notation, there is no good way for representing functions, as a consequence functions and formulas are confused.

The expression $x \cdot y^2$ can represent infinitely many functions of type Real $\rightarrow$ Real.

Differentation is a function of type (Real $\rightarrow$ Real) $\rightarrow$ (Real $\rightarrow$ Real).

Mathematicians, however speak of differentiating a *blue formula* after a variable.

Differentiation of $x \cdot y^2$ after $x$ results in $y^2$. Differentation of $x \cdot y^2$ after $y$ results in $2xy$. Differentation of $x \cdot y^2$ after $z$ results in 0.

Identifying functions and formulas causes problems. Does for example, the rule $(x \cdot y)' = x' \cdot y + x \cdot y'$ also holds for functions that are not represented by formulas? What if you change notation?

As a consequence, some notation for functions is needed: For this purpose, the $\lambda$-notation is used. $\lambda x{:}\, X\, t[x]$ is the function that for every $n \in X$, has the value $t[n]$. If $t[x]$ has type $Y$ on the assumption that $x$ has type $X$, then $\lambda x{:}\, X\, t[x]$ has type $X \to Y$.



Using $\lambda$-notation, it can be seen that there is only one differentiation operator of type $(\mathrm{Real} \to \mathrm{Real}) \to (\mathrm{Real} \to \mathrm{Real})$. The different derivatives are obtained by different $\lambda$-abstractions:

$$(\lambda x{:}\, \mathrm{Real}\ x \cdot y^2)' = \lambda x{:}\, \mathrm{Real}\ y^2,$$

$$(\lambda y{:}\, \mathrm{Real}\ x \cdot y^2)' = \lambda y{:}\, \mathrm{Real}\ x,$$

$$(\lambda z{:}\, \mathrm{Real}\ x \cdot y^2)' = \lambda x{:}\, \mathrm{Real}\ 0.$$



The differentiation rule can now be formulated as

$$\forall x, y{:}\, \mathrm{Real} \to \mathrm{Real}\ (x \cdot y)' = x' \cdot y + x \cdot y'.$$

In this expression, the $+$ and the $\cdot$ are not the usual operators on numbers of type $[\mathrm{Real}, \mathrm{Real}] \to \mathrm{Real}$, instead they are of type $[\mathrm{Real} \to \mathrm{Real}, \mathrm{Real} \to \mathrm{Real}] \to \mathrm{Real}$.

For example, $+$ is defined as

$$\lambda f_1, f_2{:}\, \mathrm{Real} \to \mathrm{Real}\ \lambda x{:}\, \mathrm{Real}\ f_1(x) + f_2(x).$$



## $\beta$-Conversion 1

By definition, $\lambda x{:}\, X\, t[x]$ is the function that, for every $n \in X$, assumes the value $t[n]$. Therefore

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

This equivalence is called $\beta$-equivalence. Applying the equality from left to right is called $\beta$-reduction.

## $\beta$-Conversion 2

Nothing goes without $\beta$-reduction. If one instantiates

$$\forall x, y\colon \text{Real} \to \text{Real } (x \cdot y)' = x' \cdot y + x \cdot y'$$

by

$$\lambda x\colon \text{Real } x, \text{ and } \lambda x\colon \text{Real } x^2,$$

the result equals

$$((\lambda x\colon \text{Real } x) \cdot (\lambda x\colon \text{Real } x^2))' =$$

$$(\lambda x\colon \text{Real } x)' \cdot (\lambda x\colon \text{Real } x^2) + (\lambda x\colon \text{Real } x) \cdot (\lambda x\colon \text{Real } x^2)'.$$

$\beta$-normalization (and applying the definitions of $+$ and $\cdot$) results in

$$(\lambda x\colon \text{Real } x^3)' = (\lambda x\colon \text{Real } x)' \cdot (\lambda x\colon \text{Real } x^2) + (\lambda x\colon \text{Real } x) \cdot (\lambda x\colon \text{Real } x^2)'.$$



## $\beta$-Conversion 3

Similarly, $\beta$-reduction is needed when applying induction.
If one instantiates in the induction axiom
$\forall P\ P(0) \wedge \forall n\colon \text{Nat } P(n) \to P(n+1) \to \forall n\colon \text{Nat } P(n)$,
the formula
$\lambda n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real } (n+1).x^n)$,
the result equals
$(\lambda n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real } (n+1).x^n))(0) \wedge$
$\forall n\colon \text{Nat } \lambda n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real } (n+1).x^n)(n) \to$
$\lambda n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real } (n+1).x^n)(n+1)$
$\to$
$\forall n\colon \text{Nat} \lambda n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real } (n+1).x^n)(n)$.



After $\beta$-reduction, this monster equals

$$(\lambda x\colon \text{Real } x)' = (\lambda x\colon \text{Real } 1) \wedge$$

$$\forall n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real}(n+1) \cdot x^n) \to$$
$$(\lambda x\colon \text{Real } x^{n+2})' = (\lambda x\colon \text{Real}(n+2) \cdot x^{n+2}) \to$$

$$\forall n\colon \text{Nat } (\lambda x\colon \text{Real } x^{n+1})' = (\lambda x\colon \text{Real}(n+1) \cdot x^n$$

One can guess why the $\lambda$-notation never got popular. Fortunately, computers are good at making such replacements.



## Free and Bound Variables

In the $\lambda$-term

$$\lambda x\colon X\ t,$$

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

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

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

## $\alpha$-Variants

Informally, two terms $t_1$ and $t_2$ are $\alpha$-variants, if they have the same meaning. Two $\lambda$-terms $t_1$ and $t_2$ are $\alpha$-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 $\lambda$, then at the corresponding position in $t_2$, there is also a subterm built by $\lambda$.
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 $\lambda$'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 $\lambda$'s inside $t$.

THEREFORE First replace $t$ by an $\alpha$-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 $\lambda$-terms, a couple of equivalence relations can be defined. They are usually called $\alpha, \beta, \delta,$ and $\eta$-equivalence. (Nobody seems to worry about the missing $\gamma$-equivalence)

One of them we defined already, namely $\alpha$-equivalence:

$$t_1 \equiv_\alpha t_2$$

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

## $\beta$ and $\eta$-equivalence

Another type of equivalence is $\beta$-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

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

$\eta$-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 $f$.



## $\delta$-equivalence

The last equivalence is $\delta$-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 $\mathrm{Form}, \mathrm{Type}, \mathrm{Kind}$ are called sorts. We write $S$ for the set of sorts, that is $S = \{\mathrm{Form}, \mathrm{Type}, \mathrm{Kind}\}$.

We now need the following:

1. Rules for determining the type of a $\lambda$-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 $\lambda$-term

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

**SORT:** For every context $C$, we have
$C \vdash \mathrm{Type}{:}\mathrm{Kind}$ and
$C \vdash \mathrm{Form}{:}\mathrm{Kind}$.

**AXIOM:** $C, \ x{:}X \vdash x{:}X$.
$C, \ x := y{:}Y \vdash x{:}Y$.

**WEAKENING:** If $C \vdash x{:}X$, then $C, D \vdash x{:}X$, 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 $\rightarrow$: Form $\Rightarrow$ Form
  $\forall$: $\Pi X$: Type ($X \Rightarrow$ Form) $\Rightarrow$ Form
- Each element of $\Gamma, \Delta$ has type Form in $C$.

Formulae can be freely replaced by formulae that are $\alpha, \beta, \delta, \eta$-equivalent.



Axioms:

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

on the condition that $C \vdash A$: 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 $\rightarrow$:

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

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



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. As was the case with sequent calculus, formulae can be freely replaced by formulae that are $\alpha, \beta, \delta, \eta$-equivalent.

$\rightarrow$-introduction:

$A$

$\ldots$

$B$

$A \rightarrow B$



$\rightarrow$-elimination:

$A$

$\ldots$

$A \rightarrow B$

$\ldots$

$B$



$\forall$-introduction:

$y{:}\,X$

$\ldots$

$F[x := y]$

$\forall x{:}\,X\ F.$

$\forall$-elimination:

$\forall x{:}\,X\ F$

$\ldots$

$F[x := t]$

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



## 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.

- Sequent Calculus defines classical Higher Order Logic.

  Intuitionistic Higher Order Logic can be obtained by forbidding sequents with more than one formula on the right.

## Definition of Logical Operators in HOL

Both calculi contain only rules for $\forall$ and $\rightarrow$. Other logical operators can be defined or declared when they are needed.

The following distinction is often made:

**Meta Logic** The logic used for checking the proofs. Nearly always HOL.

**Object Logic** The logic that the user is really interested in for doing his proofs, expressing his theorems.

So usually, the object logic is defined in HOL, after that the theory is defined in the object logic.



## Embedding of FOL into HOL

First order logic can be defined in two ways:

**By declaration** One declares the logical operators and gives axioms that describe their behaviour.

**By definition** One defines the logical operators by terms that behave in the way the operators should.



## Declaration of Logical Operators

$\bot$: Form
$\top$: Form
$\neg$: Form $\Rightarrow$ Form
$\vee$: Form $\Rightarrow$ Form $\Rightarrow$ Form
$\wedge$: Form $\Rightarrow$ Form $\Rightarrow$ Form
$\leftrightarrow$: Form $\Rightarrow$ Form $\Rightarrow$ Form

$\exists : \Pi D{:}\,\text{Set}\ P{:}\,D \Rightarrow \text{Form}\ (P \Rightarrow \text{Form}) \Rightarrow \text{Form}$
$\approx : \Pi D{:}\,\text{Set}\ D \Rightarrow D \Rightarrow \text{Form}$



## Axioms for Logical Operators (1)

$\forall P{:}\,\text{Form}\ \bot \rightarrow P$

$\top$

$\forall P{:}\,\text{Form}\ (P \rightarrow \bot) \rightarrow \neg P$
$\forall P{:}\,\text{Form}\ P \rightarrow (\neg P) \rightarrow \bot$

## Axioms for Logical Operators (2)

$\forall P, Q\colon \text{Form } P \to Q \to P \wedge Q$
$\forall P, Q\colon \text{Form } P \wedge Q \to P$
$\forall P, Q\colon \text{Form } P \wedge Q \to Q$

$\forall P, Q\colon \text{Form } P \to P \vee Q$
$\forall P, Q\colon \text{Form } Q \to P \vee Q$
$\forall P, Q\colon \text{Form } P \vee Q \to \forall C\colon \text{Form } (P \to C) \to (Q \to C) \to C$

$\forall P, Q\colon \text{Form } (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q)$
$\forall P, Q\colon \text{Form } (P \leftrightarrow Q) \to (P \to Q)$
$\forall P, Q\colon \text{Form } (P \leftrightarrow Q) \to (Q \to P)$



## Axioms for Logical Operators (3)

$\forall D\colon \text{Type } \forall P\colon D \Rightarrow \text{Form } \forall d\colon D\ (P\ d) \to (\exists\ D\ P)$
$\forall D\colon \text{Type } \forall P\colon D \Rightarrow \text{Form}$
$\qquad (\exists\ D\ P) \to \forall C\colon \text{Form } (\forall d\colon D\ (P\ d) \to C) \to C$

$\forall D\colon \text{Type } (\approx\ D\ d\ d)$
$\forall D\colon \text{Type } \forall d_1, d_2\colon D \Rightarrow \text{Form}$
$\qquad (\approx\ D\ d_1\ d_2) \to \forall P\colon D \Rightarrow \text{Form } (P\ d_1) \to (P\ d_2)$



## Definitions of Logical Operators

Defining the Logical Operators has the advantage that the logical operators are invisible, because definitions can be expanded without additional proof steps. This makes the proofs usually shorter. Unfortunately proofs also become harder to understand.

$\bot := \Pi P\colon \text{Form}\colon \text{Form}$
$\neg := \lambda P\colon \text{ Form } (P \to \bot)\colon \text{ Form} \Rightarrow \text{Form}$
$\vee := \lambda P, Q\colon \text{Form}$
$\qquad \forall R\colon \text{Form } (P \to R) \to (Q \to R) \to R\colon \text{Form} \Rightarrow \text{Form} \Rightarrow \text{Form}$

$\wedge := \lambda P, Q\colon \text{Form}$
$\qquad \forall R\colon \text{Form } (P \to Q \to R) \to R\colon \text{Form} \Rightarrow \text{Form} \Rightarrow \text{Form}$

$\leftrightarrow := \lambda P, Q\colon \text{Form } (P \to Q) \wedge (Q \to P)\colon \text{Form} \Rightarrow \text{Form} \Rightarrow \text{Form}$



$\exists := \lambda D\colon \text{Type } \lambda P\colon D \to \text{Form}$
$\qquad \forall Q\colon \text{Form } (\forall d\colon D\ (P\ d) \to Q) \to Q$
$\qquad\qquad \colon \Pi D\colon \text{Type } \Pi P\colon D \Rightarrow \text{Form } (P \Rightarrow \text{Form}) \Rightarrow \text{Form}$

$\approx := \lambda D\colon \text{Type } \lambda d_1, d_2\colon D$
$\qquad \forall P\colon D \Rightarrow \text{Form } (P\ d_1) \to (P\ d_2)\colon \Pi D\colon \text{Type } D \Rightarrow D \Rightarrow \text{Form}$

An alternative definition is the following:
$\approx := \lambda D\colon \text{Type } \lambda d_1, d_2\colon D$
$\qquad \forall P\colon D \Rightarrow D \Rightarrow \text{Form } \forall d\colon D\ (P\ d\ d) \to (P\ d_1\ d_2)\colon$
$\qquad\qquad \Pi D\colon \text{Type} D \Rightarrow D \Rightarrow \text{Form}$