HOL IN ISABELLE

Higher-Order Logic (HOL)

In HOL, we represent sets and predicates by $\boxed{\textbf{functions}}$, often denoted by **lambda abstractions**.

Definition (Lambda Abstraction)

Lambda abstractions are **terms** that denote functions directly by the rules which define them, *e.g.* the square function is $\lambda x.\ x * x$.

This is a way of defining a function without giving it a name:

$$f(x) \equiv x * x \qquad \text{vs} \qquad f \equiv \lambda x.\ x * x$$

We can use lambda abstractions exactly as we use ordinary function symbols. E.g. $\underbrace{(\lambda x.\ x * x)\,3} = 3 * 3 = 9$.

See β-reduction later in the lecture.

Higher-Order Functions

Using λ-notation, we can think about functions as $\boxed{\text{individual objects.}}$

E.g., we can define functions which map from and to other functions.

Example

The K-combinator maps some x to a function which sends any y to x.

$$\underbrace{\lambda x.\ \lambda y.\ x}_{\text{takes 2 arguments and returns } x} \qquad \text{thus, e.g.} \qquad (\lambda x.\ \lambda y.\ x)\,3 = \lambda y.\ 3$$

Example

The composition function maps two **functions** to their composition:

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

Representation of Logic in HOL I

- Types *bool*, *ind* (individuals) and $\alpha \Rightarrow \beta$ primitive. All others defined from these.
- Two primitive (families of) functions:

$$\begin{array}{ll} \checkmark\text{equality} & (=_\alpha) : \ \alpha \Rightarrow \alpha \Rightarrow bool \\ \checkmark\text{implication} & (\to) : bool \Rightarrow bool \Rightarrow bool \end{array}$$

  All other functions defined using this, lambda abstraction and application.
- Distinction between formulas and terms is dispensed with: formulas are just terms of type *bool*.
- Predicates are represented by functions $\alpha \Rightarrow bool$. Sets are represented as predicates.

Representation of Logic in HOL II

False can be defined as:

$$\bot \equiv \forall P.\ P$$

- $\boxed{\text{True}}$ is defined as:

$$\top \equiv \underbrace{(\lambda x.x) = (\lambda x.x)}$$

- $\boxed{\text{Universal quantification}}$ as function equality:

$$\forall x.\ \phi \equiv \underbrace{(\lambda x.\ \phi) = (\lambda x.\top)}.$$

  This works for x of any type: *bool*, $ind \Rightarrow bool$, ...
- (+ of HOL) Therefore, we can $\boxed{\textbf{quantify}}$ **over functions, predicates and sets.**
- $\boxed{\text{Conjunction}}$ and $\boxed{\text{disjunction}}$ are defined:

$$\begin{array}{lll} \checkmark P \wedge Q & \equiv & \forall R.(P \to Q \to R) \to R \\ \checkmark P \vee Q & \equiv & \forall R.(P \to R) \to (Q \to R) \to R \end{array}$$

- Define natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), rationals ($\mathbb{Q}$), reals ($\mathbb{R}$), complex numbers ($\mathbb{C}$), vector spaces, manifolds, ...

Isabelle/HOL

Higher-Order Logic is the underlying logic of Isabelle/HOL, the theorem prover we are using.

The axiomatisation is slightly different to the one described on the previous slides, and a bit more powerful (but we won't be delving into this).

We are interested in Isabelle/HOL from a functional programming and logic standpoint.

HOL = Functional Programming + Logic

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

τ	::=	(τ)	
	|	*bool* | *nat* | *int* | ...	base types
	|	*'a* | *'b* | ...	type variables
	|	$\tau \Rightarrow \tau$	functions
	|	$\tau \times \tau$	pairs (ascii: *)
	|	τ *list*	lists
	|	τ *set*	sets
	|	...	user-defined types

Convention: $\boxed{\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)}$ Right associative
A formula is simply a term of type *bool*.

Isabelle/HOL Terms

Terms can be formed as follows:

- $\boxed{\textit{Function application:}}$ $f\,t$
  is the call of function f with argument t.
  If f has more arguments: $f\,t_1\,t_2 \ldots$
  Examples: $sin\ \pi$, $plus\ x\ y$
- $\boxed{\textit{Function abstraction:}}$ $\lambda x.\ t$
  is the function with parameter x and result t,
  i.e. "$x \mapsto t$".
  Example: $\lambda x.\ plus\ x\ x$

Note: $\underbrace{\lambda x_1.\lambda x_2.\ldots.\lambda x_n.\ t}$ is usually denoted by $\underbrace{\lambda x_1\ x_2 \ldots x_n.\ t}$

Isabelle/HOL Terms

Basic syntax:

t	::=	(t)	
	|	a	constant or variable (identifier)
	|	$t\, t$	function application
	|	$\lambda x.\ t$	function abstraction
	|	...	lots of syntactic sugar

Examples: $f\,(g\,x)\,y$
$h\,(\lambda x.\ f\,(g\,x))$

Convention: $f\, t_1\, t_2\, t_3 \equiv ((f\, t_1)\, t_2)\, t_3$

This language of terms is known as the $\boxed{\lambda\text{-calculus.}}$

β-reduction

The computation rule of the λ-calculus is the replacement of formal by actual parameters:

$$\boxed{(\lambda x.\ t)\, u = t[u/x]}$$

where $t[u/x]$ is "t with u substituted for x".

Example: $(\lambda x.\ x + 5)\, 3 = 3 + 5$

- The step from $(\lambda x.\ t)\, u$ to $t[u/x]$ is called $\boxed{\beta\text{-reduction.}}$
- Isabelle performs β-reduction automatically.

e.g. $(\lambda x.\ (\lambda w.\ x\ w))\ f\ j = (\lambda w.\ f\ w)\, j$
$= \boxed{f\ j}$

Well-typed Terms

$\boxed{\text{Terms must be well-typed}}$

(the argument of every function call must be of the right type)

The type info. is needed bc. numbers and arithmetic operations are overloaded in Isabelle, and w/o it, Isabelle will assume tht. the type of 0 is 'a rather than nat

Notation:
$t :: \tau$ means "t is a well-typed term of type τ".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\, u :: \tau_2}$$

NOT a FOL formula

e.g. $\boxed{\forall P.\ P(0 :: nat) \wedge (\forall n.\ P\, n \rightarrow P(n+1)) \rightarrow (\forall n.\ P\, n)}$

→ This formula is equiv. to: $[\![?P\ 0;\ \forall n.\ ?P\, n \rightarrow ?P\ (Suc\ n)]\!]$
$\Rightarrow ?P\ ?n$

Type inference

→ Isabelle automatically computes the type of each variable in a term. This is called $\boxed{\text{type inference.}}$

→ In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

User can help with $\boxed{\text{type annotations}}$ inside the term.

Examples $f\,(x{::}nat)$
$g\,(A{::}real\ set)$

e.g. $(\lambda x :: nat \Rightarrow real.\ (\lambda w :: nat.\ x\ w))\ (f\ (j :: complex))$
What is type of 'f'?
↳ complex ⇒ nat ⇒ real

Currying

Process of transforming a function that takes multiple arguments into:

- one that takes just a single argument, and
- returns another *function* if any arguments are still needed.

Typing:

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Advantage:

Currying allows *partial application*
$f\, a_1 :: \tau_2 \Rightarrow \tau$ where $a_1 :: \tau_1$

So, e.g. if *plus* :: nat ⇒ nat ⇒ nat then *plus 10* :: nat ⇒ nat

Predefined syntactic sugar

- *Infix:* +, -, *, #, @, ...
- *Mixfix:* *if _ then _ else _*, *case _ of*, ...

Prefix binds more strongly than infix:

! $f\,x + y \equiv (f\,x) + y \not\equiv f(x + y)$ **!**

Enclose *if* and *case* in parentheses:

! *(if _ then _ else _)* **!**

Example: Type *bool*

datatype *bool* = *True* | *False*

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots :: bool \Rightarrow bool \Rightarrow bool$

A *formula* is a term of type *bool*

if-and-only-if: =

Example: Type *nat*

datatype *nat* = *0* | *Suc nat*

Values of type *nat*: *0*, $\underbrace{Suc\ 0}_{}$, $\underbrace{Suc(Suc\ 0)}_{}$, . . .

Predefined functions: +, *, ... :: *nat* ⇒ *nat* ⇒ *nat*

! Numbers and arithmetic operations are overloaded:
0, 1, 2, ... :: 'a, + :: *'a ⇒ 'a ⇒ 'a*

You need type annotations: *1 :: nat, x + (y::nat)*
unless the context is unambiguous: *Suc z*