Automated Reasoning

Lecture 6: Introduction to Higher Order Logic in Isabelle

Jacques Fleuriot
jdf@inf.ed.ac.uk

Acknowledgement: Tobias Nipkow kindly provided some of the slides for this lecture

Higher-Order Logic (HOL)

In HOL, we represent sets and predicates by **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. $(\lambda x.\ x * x)\,3 = 3 * 3 = 9$.

See β-reduction later in the lecture.

Higher-Order Functions

Using λ-notation, we can think about functions as 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.

$$\lambda x.\ \lambda y.\ x \qquad \text{thus, e.g.} \qquad (\lambda x.\ \lambda y.\ x)\, 3 = \lambda y.\ 3$$

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:

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

  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 $\alpha \Rightarrow \mathit{bool}$.
  Sets are represented as predicates.

Representation of Logic in HOL II

- True is defined as:

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

- Universal quantification as function equality:

$$\forall x.\ \phi \equiv (\lambda x.\ \phi) = (\lambda x.\top)\,.$$

  This works for x of any type: $bool$, $ind \Rightarrow bool$, ...

- Therefore, we can **quantify over functions, predicates and sets**.
- Conjunction and disjunction are defined:

$$\begin{array}{lcl} P \wedge Q & \equiv & \forall R.(P \rightarrow Q \rightarrow R) \rightarrow R \\ P \vee Q & \equiv & \forall R.(P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \end{array}$$

- 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 = Higher-Order Logic

HOL = Functional Programming + Logic

HOL = Functional Programming + Logic

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

HOL has

- datatypes
- recursive functions
- logical operators

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

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):

$$\tau \quad ::=$$

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

$$\tau \quad ::= \quad (\tau)$$

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

$$\tau \quad ::= \quad (\tau)$$
$$\quad | \quad \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots \qquad \text{base types}$$

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid$ *int* $\mid$ ...	base types
	$\mid$	*'a* $\mid$ *'b* $\mid$ ...	type variables

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

τ	$::=$	(τ)	
	$\mid$	$\mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots$	base types
	$\mid$	$\mathit{'a} \mid \mathit{'b} \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

τ	$::=$	(τ)	
	$\mid$	$\mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid$ *int* $\mid$ ...	base types
	$\mid$	*'a* $\mid$ *'b* $\mid$ ...	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs
	$\mid$	τ *list*	lists

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

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

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

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

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
	|	τ *list*	lists
	|	τ *set*	sets
	|	...	user-defined types

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \ \equiv \ \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

Isabelle/HOL Types

Basic syntax (as a BNF grammar):

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

Convention: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \;\equiv\; \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$

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

Isabelle/HOL Terms

Terms can be formed as follows:

Isabelle/HOL Terms

Terms can be formed as follows:

- *Function application:* $f\,t$

Isabelle/HOL Terms

Terms can be formed as follows:

- *Function application:* $f\,t$
  is the call of function f with argument t.

Isabelle/HOL Terms

Terms can be formed as follows:

- *Function application:* $f\,t$
  is the call of function f with argument t.
  If f has more arguments: $f\,t_1\;t_2\;\ldots$

Isabelle/HOL Terms

Terms can be formed as follows:

- *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: $sin\,\pi$, $plus\,x\,y$

Isabelle/HOL Terms

Terms can be formed as follows:

- *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: $sin\,\pi$, $plus\,x\,y$
- *Function abstraction:* $\lambda x.\;t$

Isabelle/HOL Terms

Terms can be formed as follows:

- *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: $sin\,\pi$, $plus\,x\,y$
- *Function abstraction:* $\lambda x.\ t$
  is the function with parameter x and result t

Isabelle/HOL Terms

Terms can be formed as follows:

- *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: $sin\,\pi$, $plus\,x\,y$
- *Function abstraction:* $\lambda x.\;t$
  is the function with parameter x and result t,
  i.e. "$x \mapsto t$".

Isabelle/HOL Terms

Terms can be formed as follows:

- *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: $sin\,\pi$, $plus\,x\,y$
- *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: $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.\,t$ is usually denoted by $\lambda x_1\;x_2\ldots x_n.\;t$

Isabelle/HOL Terms

Basic syntax:

$$t \quad ::=$$

Isabelle/HOL Terms

Basic syntax:

$$t \quad ::= \quad (t)$$

Isabelle/HOL Terms

Basic syntax:

$$
\begin{array}{lcll}
t & ::= & (t) & \\
  & | & a   & \text{constant or variable (identifier)}
\end{array}
$$

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	$\lambda x.\ t$	function abstraction

Isabelle/HOL Terms

Basic syntax:

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

Isabelle/HOL Terms

Basic syntax:

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

Examples: $f\,(g\ x)\ y$

Isabelle/HOL Terms

Basic syntax:

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

Examples: $f\ (g\ x)\ y$

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

Isabelle/HOL Terms

Basic syntax:

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

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

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

Isabelle/HOL Terms

Basic syntax:

t	$::=$	(t)	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$t\ t$	function application
	$\mid$	$\lambda x.\ t$	function abstraction
	$\mid$	...	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 *λ-calculus*.

β-reduction

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

$$(\lambda x.\ t)\ u \ = \ t[u/x]$$

β-reduction

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

$$(\lambda x.\, t)\, u \;=\; t[u/x]$$

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

β-reduction

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

$$(\lambda x.\ t)\ u \ = \ t[u/x]$$

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

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

β-reduction

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

$$(\lambda x.\, t)\, u \;=\; t[u/x]$$

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 *β-reduction*.

β-reduction

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

$$(\lambda x.\ t)\ u \ = \ t[u/x]$$

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 *β-reduction*.
- Isabelle performs β-reduction automatically.

Well-typed Terms

Terms must be well-typed

Well-typed Terms

Terms must be well-typed

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

Well-typed Terms

Terms must be well-typed

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

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

Well-typed Terms

Terms must be well-typed

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

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

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

Type inference

Isabelle automatically computes the type of each variable in a term.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

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

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

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

User can help with *type annotations* inside the term.

Examples $f\ (x{::}nat)$
$4{::}real$
$g\ (A{::}real\ set)$

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.

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: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

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*

$$f\, a_1 :: \tau_2 \Rightarrow \tau \quad \text{where} \quad a_1 :: \tau_1$$

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

Predefined syntactic sugar

- *Infix:* +, -, *, #, @, ...

Predefined syntactic sugar

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

Predefined syntactic sugar

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

Prefix binds more strongly than infix:

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

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 $bool = True \mid False$

Example: Type *bool*

datatype $bool = True \mid False$

Predefined functions:

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

Example: Type *bool*

datatype $bool = True \mid False$

Predefined functions:

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

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

Example: Type *bool*

datatype $bool = True \mid False$

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

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

if-and-only-if: $=$ or $\leftrightarrow$

Example: Type *nat*

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

Example: Type *nat*

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

Values of type *nat*: *0*, *Suc 0*, *Suc(Suc 0)*, . . .

Example: Type *nat*

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

Values of type *nat*: *0*, *Suc 0*, *Suc(Suc 0)*, . . .

Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

Example: Type *nat*

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

Values of type *nat*: *0*, *Suc 0*, *Suc(Suc 0)*, . . .

Predefined functions: $+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:

$0, 1, 2, \ldots :: 'a, \quad + :: \ 'a \Rightarrow 'a \Rightarrow 'a$

Example: Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Values of type *nat*: 0, $Suc\ 0$, $Suc(Suc\ 0)$, . . .

Predefined functions: $+, *, ... :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0, 1, 2, ... :: 'a, \quad + :: 'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1 :: nat$, $x + (y::nat)$

Example: Type *nat*

datatype $nat = 0 \mid Suc\ nat$

Values of type *nat*: 0, $Suc\ 0$, $Suc(Suc\ 0)$, . . .

Predefined functions: $+, *, ... :: nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0, 1, 2, ... :: 'a, \quad + :: 'a \Rightarrow 'a \Rightarrow 'a$

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

More on Isabelle/HOL

If you are really keen, look at the chapter "Higher-Order Logic" in the "logics" document in the Isabelle documentation.

Or the file `src/HOL/HOL.thy` in the Isabelle installation.

Exercise (only if you are interested!): why can't Russell's paradox happen in HOL?

Summary

- General introduction to Higher-Order Logic
- Types and Terms in Isabelle/HOL
- As usual, see recommended reading on AR Lecture Schedule page