Automated Reasoning

Lecture 7: 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
Recap

- Last time: Representing mathematical concepts
- This time: Higher-Order Logic (in Isabelle)
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 $\lambda x. x \times x$.

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

$$f(x) \equiv x \times x \quad \text{vs} \quad f \equiv \lambda x. x \times x$$

We can use lambda abstractions exactly as we use ordinary function symbols. E.g. $(\lambda x. x \times x) 3 = 9$. 
Higher-Order Functions

Using \( \lambda \)-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 \( x \).

\[
\lambda x. \lambda y. x.
\]

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

\[
\lambda f. \lambda g. \lambda x. f(g \ x).
\]
Types `bool`, `ind` (individuals) and \( \alpha \Rightarrow \beta \) primitive. All others defined from these.

Two primitive (families of) functions:

- equality \( (=_{\alpha}) : \alpha \Rightarrow \alpha \Rightarrow bool \)
- implication \( (\rightarrow) : bool \Rightarrow bool \Rightarrow bool \)

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.
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: \textit{bool}, \textit{ind} \Rightarrow \textit{bool}, ...

Therefore, we can quantify over functions, predicates and sets.

Conjunction and disjunction are defined:

\[ P \land Q \equiv \forall R. (P \to Q \to R) \to R \]
\[ P \lor Q \equiv \forall R. (P \to R) \to (Q \to R) \to R \]

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

\[ \tau \ ::= \]

Convention:
Basic syntax:

\[ \tau ::= (\tau) \]
Isabelle/HOL Types

Basic syntax:

\[ \tau ::= (\tau) \quad \mid \quad \text{bool} \quad \mid \quad \text{nat} \quad \mid \quad \text{int} \quad \mid \ldots \quad \text{base types} \]
Isabelle/HOL Types

Basic syntax:

\[ \tau ::= (\tau) \]
\[ \quad | \quad \text{bool} \quad | \quad \text{nat} \quad | \quad \text{int} \quad | \quad \ldots \quad \text{base types} \]
\[ \quad | \quad \text{'a} \quad | \quad \text{'b} \quad | \quad \ldots \quad \text{type variables} \]
Isabelle/HOL Types

Basic syntax:

\[ \tau \quad ::= \quad (\tau) \]
\[ \quad | \quad \text{bool} \quad | \quad \text{nat} \quad | \quad \text{int} \quad | \quad \ldots \quad \text{base types} \]
\[ \quad | \quad 'a \quad | \quad 'b \quad | \quad \ldots \quad \text{type variables} \]
\[ \quad | \quad \tau \Rightarrow \tau \quad \text{functions} \]
Isabelle/HOL Types

Basic syntax:

\[ \tau ::= (\tau) \]
\[ \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \quad \text{base types} \]
\[ \mid \text{'}a \mid \text{'}b \mid \ldots \quad \text{type variables} \]
\[ \mid \tau \Rightarrow \tau \quad \text{functions} \]
\[ \mid \tau \times \tau \quad \text{pairs (ascii: \ast)} \]
Isabelle/HOL Types

Basic syntax:

\[
\tau ::= (\tau) \\
| \text{bool} | \text{nat} | \text{int} | \ldots \quad \text{base types} \\
| 'a | 'b | \ldots \quad \text{type variables} \\
| \tau \Rightarrow \tau \quad \text{functions} \\
| \tau \times \tau \quad \text{pairs (ascii: *)} \\
| \tau \text{ list} \quad \text{lists}
\]
Isabelle/HOL Types

Basic syntax:

\[ \tau ::= \ (\tau) \ \\
| \ \text{bool} \ | \ \text{nat} \ | \ \text{int} \ | \ ... \quad \text{base types} \\
| \ 'a \ | \ 'b \ | \ ... \quad \text{type variables} \\
| \ \tau \Rightarrow \tau \quad \text{functions} \\
| \ \tau \times \tau \quad \text{pairs (ascii: \ *)} \\
| \ \tau \ list \quad \text{lists} \\
| \ \tau \ set \quad \text{sets} \]
Basic syntax:

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

<table>
<thead>
<tr>
<th>$\tau \quad ::= \quad$</th>
<th>bool</th>
<th>nat</th>
<th>int</th>
<th>...</th>
<th>base types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau \Rightarrow \tau$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>functions</td>
</tr>
<tr>
<td>$\tau \times \tau$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pairs</td>
</tr>
<tr>
<td>$\tau\ list$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lists</td>
</tr>
<tr>
<td>$\tau\ set$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sets</td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>user-defined types</td>
</tr>
</tbody>
</table>
Isabelle/HOL Types

Basic syntax:

\[ \tau ::= (\tau) \]

| bool | nat | int | ... base types |
| 'a | 'b | ... type variables |
| \( \tau \Rightarrow \tau \) functions |
| \( \tau \times \tau \) pairs (ascii: *) |
| \( \tau \text{ list} \) lists |
| \( \tau \text{ set} \) 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:

\[ \tau \ ::= \ (\tau) \]

| \tau \Rightarrow \tau \quad \text{functions} |
| \tau \times \tau \quad \text{pairs (ascii: \,*\)} |
| \tau \ list \quad \text{lists} |
| \tau \ set \quad \text{sets} |
| \ldots \quad \text{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.
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: $\sin$, $\text{plus } x y$

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

  - Example: $\lambda x. \text{plus } x x$
Isabelle/HOL Terms

Terms can be formed as follows:

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

  Examples: $\sin$, $\text{plus} \, x \, y$
Terms can be formed as follows:

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

- **Function abstraction**: $\lambda x. \, t$
  
is the function with parameter $x$ and result $t$, i.e. 
  
  $$x \mapsto t$$
  
Examples:

- $\sin$,
- $\text{plus} \, x \, y$
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: \( f t_1 t_2 \ldots \)
Terms can be formed as follows:

- **Function application:** \( ft \)
  is the call of function \( f \) with argument \( t \).
  If \( f \) has more arguments: \( ft_1 t_2 \ldots \)

Examples: \( \sin \pi, \ \text{plus} \ x \ y \)
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: $\sin \pi$, $\text{plus} \ x \ y$

- **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: \( \sin \pi \), \( \text{plus } x y \)

▶ **Function abstraction**: \( \lambda x. t \)
  is the function with parameter \( x \) and result \( t \),
i.e. \( "x \mapsto t" \).
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: \( \sin \pi, \quad \text{plus} \, x \, y \)

- **Function abstraction**: \( \lambda x. \, t \)
  
is the function with parameter \( x \) and result \( t \),
  
i.e. “\( x \mapsto t \)”.

Example: \( \lambda x. \, \text{plus} \, x \, x \)
Isabelle/HOL Terms

Basic syntax:

\[ t ::= \]

\[ (t) \]

\[ j \]

\[ a \]

constant or variable (identifier)

\[ t \]

function application

\[ x. t \]

function abstraction

\[ \ldots \]

lots of syntactic sugar

Examples:

\[ f (g x) y \]

\[ h (x. f (g x)) \]

Convention:

\[ f t_1 t_2 t_3 \]

\[ ((f t_1) t_2) t_3 \]

This language of terms is known as the \(-\)calculus.
Isabelle/HOL Terms

Basic syntax:

\[ t ::= (t) \]
Isabelle/HOL Terms

Basic syntax:

\[
t ::=(t) \\
\quad | \quad a \quad \text{constant or variable (identifier)}
\]
Isabelle/HOL Terms

Basic syntax:

\[
\begin{align*}
t & ::= (t) \\
   & | a \quad \text{constant or variable (identifier)} \\
   & | t \; t \quad \text{function application}
\end{align*}
\]
Isabelle/HOL Terms

Basic syntax:

\[
\begin{align*}
  t & ::= (t) \\
  & \mid a \quad \text{constant or variable (identifier)} \\
  & \mid t \, t \quad \text{function application} \\
  & \mid \lambda x. \, t \quad \text{function abstraction}
\end{align*}
\]
Basic syntax:

\[ t ::= (t) \]
\[ \mid a \quad \text{constant or variable (identifier)} \]
\[ \mid t t \quad \text{function application} \]
\[ \mid \lambda x. t \quad \text{function abstraction} \]
\[ \mid \ldots \quad \text{lots of syntactic sugar} \]
Isabelle/HOL Terms

Basic syntax:

\[ t ::= (t) \]
\[ | a \quad \text{constant or variable (identifier)} \]
\[ | t \ t \quad \text{function application} \]
\[ | \lambda x. \ t \quad \text{function abstraction} \]
\[ | ... \quad \text{lots of syntactic sugar} \]

Examples: \( f(g \ x) \ y \)
Isabelle/HOL Terms

Basic syntax:

\[ t ::= \begin{align*}
(t) \\
| a \quad \text{constant or variable (identifier)} \\
| tt \quad \text{function application} \\
| \lambda x. t \quad \text{function abstraction} \\
| \ldots \quad \text{lots of syntactic sugar}
\end{align*} \]

Examples:
\[ f (g \, x) \, y \]
\[ h (\lambda x. f (g \, x)) \]
Isabelle/HOL Terms

Basic syntax:

\[ t ::= (t) \]
\[ ~a~ \quad \text{constant or variable (identifier)} \]
\[ t \; t \; \text{function application} \]
\[ \lambda x. \; t \; \text{function abstraction} \]
\[ ~...~ \; \text{lots of syntactic sugar} \]

Examples:

\[ f \; (g \; x) \; y \]
\[ h \; (\lambda x. \; f \; (g \; x)) \]

Convention:

\[ f \; t_1 \; t_2 \; t_3 \; \equiv \; ((f \; t_1) \; t_2) \; t_3 \]
Basic syntax:

\[ t ::= (t) \]

\[ | a \quad \text{constant or variable (identifier)} \]

\[ | t t \quad \text{function application} \]

\[ | \lambda x. t \quad \text{function abstraction} \]

\[ | \ldots \quad \text{lots of syntactic sugar} \]

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 \(\lambda\)-calculus.
The computation rule of the \( \lambda \)-calculus is the replacement of formal by actual parameters:

\[ (\lambda x. t) u = t[u/x] \]
The computation rule of the $\lambda$-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 $x$”.
The computation rule of the $\lambda$-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 computation rule of the \( \lambda \)-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 \( \beta \)-reduction.
The computation rule of the $\lambda$-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 $\beta$-reduction.
- Isabelle performs $\beta$-reduction automatically.
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)
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 \( \tau \)”.
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 \( \tau \)”.

\[
\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2}
\]
Isabelle automatically computes the type of each variable in a term.
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

\[
\begin{align*}
  f (x::nat) \\
  g (A::\text{real set})
\end{align*}
\]
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 \textit{function} if any arguments are still needed.

\textbf{Typing}:

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

\textbf{Advantage}:

Currying allows \textit{partial application}

\[ f \ a_1 :: \tau_2 \Rightarrow \tau \text{ where } a_1 :: \tau_1 \]
Predefined syntactic sugar

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

**Prefix binds more strongly than infix:**

\[ f \times (x + y) \neq f (x + y) \]

Enclose \texttt{if} and \texttt{case} in parentheses:

\[ (\texttt{if } x > 0 \texttt{ then } y \texttt{ else } z) \]
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

```
!f x + y ̸= f (x + y)
```

Enclose if and case in parentheses:

```
!(if _ then _ else _)
```
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[
\begin{align*}
\text{!} & \quad fx + y \equiv (fx) + y \neq f(x + y) \\
\end{align*}
\]
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[ f x + y \equiv (f x) + y \neq f (x + y) \]

Enclose *if* and *case* in parentheses:

\[ (if _ then _ else _) \]
Example: Type `bool`

```plaintext
datatype bool = True | False
```
Example: Type bool

datatype bool = True | False

Predefined functions:
\(\land, \lor, \rightarrow, \ldots\) :: bool \Rightarrow bool \Rightarrow bool
Example: Type $bool$

```plaintext
datatype bool = True | False

Predefined functions:
\(\land, \lor, \rightarrow, \ldots\) :: bool \Rightarrow bool \Rightarrow bool
```

A formula is a term of type $bool$
Example: Type \textit{bool}

\textbf{datatype} \hspace{1em} \textit{bool} = True \mid False

Predefined functions:
$\land, \lor, \rightarrow, \ldots \hspace{1em} :: \textit{bool} \Rightarrow \textit{bool} \Rightarrow \textit{bool}$

A \textit{formula} is a term of type \textit{bool}

if-and-only-if: $=$
Example: Type \textit{nat}

\textbf{datatype} \hspace{1em} \textit{nat} = 0 \mid \text{Suc} \textit{nat}
Example: Type \textit{nat}

\begin{verbatim}

datatype nat = 0 | Suc nat

Values of type \textit{nat}: 0, Suc 0, Suc(Suc 0), \ldots
\end{verbatim}
Example: Type \textit{nat}

\textbf{datatype} \hspace{1em} \textit{nat} = 0 \mid \text{Suc} \textit{nat}

Values of type \textit{nat}: 0, \text{Suc} 0, \text{Suc}(\text{Suc} 0), \ldots

Predefined functions: +, *, \ldots :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}
Example: Type $\text{nat}$

```plaintext
datatype $\text{nat} = 0 \mid \text{Suc}\ \text{nat}$
```

Values of type $\text{nat}$: $0, \text{Suc}\ 0, \text{Suc(Suc}\ 0), \ldots$

Predefined functions: $+, \ast, \ldots :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

! Numbers and arithmetic operations are overloaded:

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

\[
\text{datatype } \quad \textit{nat} = \textit{0} \mid \textit{Suc } \textit{nat}
\]

Values of type $\textit{nat}$: $0$, $\textit{Suc }0$, $\textit{Suc} (\textit{Suc }0)$, \ldots

Predefined functions: $+$, $\times$, \ldots :: $\textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}$

! Numbers and arithmetic operations are overloaded:

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

You need type annotations: $1 :: \textit{nat}$, $x + (y :: \textit{nat})$
Example: Type \( \textit{nat} \)

\begin{verbatim}

datatype nat = 0 | Suc nat

Values of type \( \textit{nat} \): 0, Suc 0, Suc(Suc 0), \ldots

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

\end{verbatim}

\begin{verbatim}
!

Numbers and arithmetic operations are overloaded:

0, 1, 2, \ldots :: ’a, + :: ’a \Rightarrow ’a \Rightarrow ’a

\end{verbatim}

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

unless the context is unambiguous: Suc z
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