
Overview of Isabelle/HOL

System Architecture

<i>Isabelle</i>	generic theorem prover

System Architecture

<i>Isabelle</i>	generic theorem prover
<i>Standard ML</i>	implementation language

System Architecture

<i>Isabelle/HOL</i>	Isabelle instance for HOL
<i>Isabelle</i>	generic theorem prover
<i>Standard ML</i>	implementation language

System Architecture

<i>ProofGeneral</i>	(X)Emacs based interface
<i>Isabelle/HOL</i>	Isabelle instance for HOL
<i>Isabelle</i>	generic theorem prover
<i>Standard ML</i>	implementation language

HOL

HOL = Higher-Order Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators (\wedge , \longrightarrow , \forall , \exists , \dots)

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators (\wedge , \longrightarrow , \forall , \exists , ...)

HOL is a programming language!

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators (\wedge , \longrightarrow , \forall , \exists , ...)

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} form ::= (form) \quad | \quad term = term \quad | \quad \neg form \\ \quad | \quad form \wedge form \quad | \quad form \vee form \quad | \quad form \longrightarrow form \\ \quad | \quad \forall x. form \quad | \quad \exists x. form \end{array}$$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} form ::= (form) \quad | \quad term = term \quad | \quad \neg form \\ \quad | \quad form \wedge form \quad | \quad form \vee form \quad | \quad form \longrightarrow form \\ \quad | \quad \forall x. form \quad | \quad \exists x. form \end{array}$$

Scope of quantifiers: as far to the right as possible

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} \textit{form} ::= (\textit{form}) \quad | \quad \textit{term} = \textit{term} \quad | \quad \neg \textit{form} \\ \quad | \quad \textit{form} \wedge \textit{form} \quad | \quad \textit{form} \vee \textit{form} \quad | \quad \textit{form} \longrightarrow \textit{form} \\ \quad | \quad \forall x. \textit{form} \quad | \quad \exists x. \textit{form} \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} form ::= (form) \quad | \quad term = term \quad | \quad \neg form \\ \quad | \quad form \wedge form \quad | \quad form \vee form \quad | \quad form \longrightarrow form \\ \quad | \quad \forall x. form \quad | \quad \exists x. form \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} \text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \quad | \quad \neg \text{form} \\ \quad \quad | \quad \text{form} \wedge \text{form} \quad | \quad \text{form} \vee \text{form} \quad | \quad \text{form} \longrightarrow \text{form} \\ \quad \quad | \quad \forall x. \text{form} \quad \quad | \quad \exists x. \text{form} \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x. P x \wedge Q x \equiv \forall x. (P x \wedge Q x)$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{l} \text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \quad | \quad \neg \text{form} \\ \quad | \quad \text{form} \wedge \text{form} \quad | \quad \text{form} \vee \text{form} \quad | \quad \text{form} \longrightarrow \text{form} \\ \quad | \quad \forall x. \text{form} \quad | \quad \exists x. \text{form} \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x. P x \wedge Q x \equiv \forall x. (P x \wedge Q x)$
- $\forall x. \exists y. P x y \wedge Q x \equiv \forall x. (\exists y. (P x y \wedge Q x))$

Formulae

Abbreviation: $\forall x y. P x y \equiv \forall x. \forall y. P x y$

Formulae

Abbreviation: $\forall x y. P x y \equiv \forall x. \forall y. P x y$ ($\forall, \exists, \lambda, \dots$)

Formulae

Abbreviation: $\forall x y. P x y \equiv \forall x. \forall y. P x y$ ($\forall, \exists, \lambda, \dots$)

Hiding and renaming:

$\forall x y. (\forall x. P x y) \wedge Q x y \equiv \forall x_0 y. (\forall x_1. P x_1 y) \wedge G x_0 y$

Formulae

Abbreviation: $\forall x y. P x y \equiv \forall x. \forall y. P x y$ ($\forall, \exists, \lambda, \dots$)

Hiding and renaming:

$\forall x y. (\forall x. P x y) \wedge Q x y \equiv \forall x_0 y. (\forall x_1. P x_1 y) \wedge G x_0 y$

Parentheses:

- \wedge, \vee and \longrightarrow associate to the right:

$$A \wedge B \wedge C \equiv A \wedge (B \wedge C)$$

Formulae

Abbreviation: $\forall x y. P x y \equiv \forall x. \forall y. P x y$ ($\forall, \exists, \lambda, \dots$)

Hiding and renaming:

$\forall x y. (\forall x. P x y) \wedge Q x y \equiv \forall x_0 y. (\forall x_1. P x_1 y) \wedge G x_0 y$

Parentheses:

- \wedge, \vee and \longrightarrow associate to the right:

$$A \wedge B \wedge C \equiv A \wedge (B \wedge C)$$

- $A \longrightarrow B \longrightarrow C \equiv A \longrightarrow (B \longrightarrow C) \not\equiv (A \longrightarrow B) \longrightarrow C$!

Warning

Quantifiers have low priority and need to be parenthesized:

$$! \quad P \wedge \forall x. Q x \quad \rightsquigarrow \quad P \wedge (\forall x. Q x) \quad !$$

Types and Terms

Types

Syntax:

$$\tau ::= (\tau)$$
$$| \textit{bool} \mid \textit{nat} \mid \dots \quad \text{base types}$$

Types

Syntax:

$\tau ::= (\tau)$
| *bool* | *nat* | ... base types
| *'a* | *'b* | ... type variables

Types

Syntax:

τ	$::=$	(τ)	
		$bool \mid nat \mid \dots$	base types
		$'a \mid 'b \mid \dots$	type variables
		$\tau \Rightarrow \tau$	total functions

Types

Syntax:

τ	$::=$	(τ)	
		$bool \mid nat \mid \dots$	base types
		$'a \mid 'b \mid \dots$	type variables
		$\tau \Rightarrow \tau$	total functions
		$\tau \times \tau$	pairs (ascii: *)

Types

Syntax:

τ	$::=$	(τ)	
		$bool \mid nat \mid \dots$	base types
		$'a \mid 'b \mid \dots$	type variables
		$\tau \Rightarrow \tau$	total functions
		$\tau \times \tau$	pairs (ascii: *)
		$\tau \textit{ list}$	lists

Types

Syntax:

τ	$::=$	(τ)	
		$bool \mid nat \mid \dots$	base types
		$'a \mid 'b \mid \dots$	type variables
		$\tau \Rightarrow \tau$	total functions
		$\tau \times \tau$	pairs (ascii: *)
		$\tau \textit{ list}$	lists
		\dots	user-defined types

Types

Syntax:

$\tau ::=$	(τ)	
	$bool \mid nat \mid \dots$	base types
	$'a \mid 'b \mid \dots$	type variables
	$\tau \Rightarrow \tau$	total functions
	$\tau \times \tau$	pairs (ascii: *)
	$\tau \textit{ list}$	lists
	\dots	user-defined types

Parentheses: $T1 \Rightarrow T2 \Rightarrow T3 \equiv T1 \Rightarrow (T2 \Rightarrow T3)$

Terms: Basic syntax

Syntax:

$term ::= (term)$	
a	constant or variable (identifier)
$term\ term$	function application
$\lambda x. term$	function “abstraction”

Terms: Basic syntax

Syntax:

$term ::= (term)$	
a	constant or variable (identifier)
$term\ term$	function application
$\lambda x. term$	function “abstraction”
\dots	lots of syntactic sugar

Terms: Basic syntax

Syntax:

$term ::= (term)$	
a	constant or variable (identifier)
$term\ term$	function application
$\lambda x. term$	function “abstraction”
\dots	lots of syntactic sugar

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

Terms: Basic syntax

Syntax:

$term ::= (term)$	
a	constant or variable (identifier)
$term\ term$	function application
$\lambda x. term$	function “abstraction”
\dots	lots of syntactic sugar

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

Parantheses: $f\ a_1\ a_2\ a_3 \equiv ((f\ a_1)\ a_2)\ a_3$

λ -calculus on one slide

Informal notation: $t[x]$

λ -calculus on one slide

Informal notation: $t[x]$

- *Function application:*

$f a$ is the call of function f with argument a

λ -calculus on one slide

Informal notation: $t[x]$

- **Function application:**
 $f a$ is the call of function f with argument a
- **Function abstraction:**
 $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.

λ -calculus on one slide

Informal notation: $t[x]$

- *Function application:*
 $f a$ is the call of function f with argument a
- *Function abstraction:*
 $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
Replace formal by actual parameter (“ β -reduction”):
 $(\lambda x.t[x]) a \longrightarrow_{\beta} t[a]$

λ -calculus on one slide

Informal notation: $t[x]$

- **Function application:**
 $f a$ is the call of function f with argument a
- **Function abstraction:**
 $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- **Computation:**
Replace formal by actual parameter (“ β -reduction”):
 $(\lambda x.t[x]) a \longrightarrow_{\beta} t[a]$

Example: $(\lambda x. x + 5) 3 \longrightarrow_{\beta} (3 + 5)$

\longrightarrow_{β} *in Isabelle: Don't worry, be happy*

Isabelle performs β -reduction automatically

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent

Terms and Types

Terms must be well-typed

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

Terms and Types

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 .

Type inference

Isabelle automatically computes (“*infers*”) the type of each variable in a term.

Type inference

Isabelle automatically computes (“*infers*”) the type of each variable in a term.

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

Type inference

Isabelle automatically computes (“*infers*”) the type of each variable in a term.

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

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

Example: $f (x::nat)$

Currying

Thou shalt curry your functions

Currying

Thou shalt curry your functions

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

Currying

Thou shalt curry your functions

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

Advantage: *partial application* $f a_1$ with $a_1 :: \tau_1$

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

$$! \quad f x + y \equiv (f x) + y \neq f (x + y) \quad !$$

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

Type bool

Formulae = terms of type *bool*

True :: *bool*

False :: *bool*

\wedge, \vee, \dots :: *bool* \Rightarrow *bool* \Rightarrow *bool*

⋮

Type *bool*

Formulae = terms of type *bool*

True :: *bool*

False :: *bool*

\wedge, \vee, \dots :: *bool* \Rightarrow *bool* \Rightarrow *bool*

⋮

if-and-only-if: =

Type nat

$0 :: \text{nat}$

$\text{Suc} :: \text{nat} \Rightarrow \text{nat}$

$+, *, \dots :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

\vdots

Type nat

$0 :: \text{nat}$

$\text{Suc} :: \text{nat} \Rightarrow \text{nat}$

$+, *, \dots :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

\vdots

! Numbers and arithmetic operations are overloaded:

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

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

Type nat

$0 :: \text{nat}$

$\text{Suc} :: \text{nat} \Rightarrow \text{nat}$

$+, *, \dots :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

\vdots

! Numbers and arithmetic operations are overloaded:

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

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

... unless the context is unambiguous: $\text{Suc } z$

Type list

- `[]`: empty list
- `x # xs`: list with first element `x` ("*head*") and rest `xs` ("*tail*")
- Syntactic sugar: `[x1, ..., xn]`

Type list

- `[]`: empty list
- `x # xs`: list with first element `x` ("*head*") and rest `xs` ("*tail*")
- Syntactic sugar: `[x1, ..., xn]`

Large library:

`hd`, `tl`, `map`, `size`, `filter`, `set`, `nth`, `take`, `drop`, `distinct`, ...

Don't reinvent, reuse!

~> `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax:

```
theory MyTh = ImpTh1 + ... + ImpThn :  
(declarations, definitions, theorems, proofs, ...)*  
end
```

- *MyTh*: name of theory. Must live in file *MyTh.thy*
- *ImpTh_i*: name of *imported* theories. Import transitive.

Theory = Module

Syntax:

```
theory MyTh = ImpTh1 + ... + ImpThn :  
(declarations, definitions, theorems, proofs, ...)*  
end
```

- *MyTh*: name of theory. Must live in file *MyTh.thy*
- *ImpTh_i*: name of *imported* theories. Import transitive.

Unless you need something special:

```
theory MyTh = Main:
```

Proof General



An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs (info: C-h i)
- Isabelle aware (when editing .thy files)
- mathematical symbols (“x-symbols”)

Interaction:

- via mouse
- or keyboard (key bindings see C-h m)

X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$): `\<and>`, `\<or>`, ...
- via abbreviation: `/\`, `\/`, `-->`, ...

x-symbol	\forall	\exists	λ	\neg	\wedge	\vee	\longrightarrow	\Rightarrow
ascii (1)	<code>\<forall></code>	<code>\<exists></code>	<code>\<lambda></code>	<code>\<not></code>	<code>/\</code>	<code>\/</code>	<code>--></code>	<code>=></code>
ascii (2)	ALL	EX	%	~	&			

(1) is converted to x-symbol, (2) stays ascii.

Demo: terms and types