Overview of Isabelle/HOL

System Architecture

Isabelle	generic theorem prover

System Architecture

Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

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 ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, $\ldots$)

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

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

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):

$$
\begin{array}{lclclcl}
form & ::= & (form) & \mid & term = term & \mid & \neg form \\
 & \mid & form \wedge form & \mid & form \vee form & \mid & form \longrightarrow form \\
 & \mid & \forall x.\ form & \mid & \exists x.\ form & &
\end{array}
$$

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{rclll}
form & ::= & (form) & \mid \; term = term & \mid \; \neg form \\
& \mid & form \wedge form & \mid \; form \vee form & \mid \; form \longrightarrow form \\
& \mid & \forall x.\ form & \mid \; \exists x.\ form &
\end{array}
$$

Examples

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

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{llllll}
form & ::= & (form) & \mid\ term = term & \mid\ \neg form \\
 & \mid & form \wedge form & \mid\ form \vee form & \mid\ form \longrightarrow form \\
 & \mid & \forall x.\ form & \mid\ \exists x.\ form &
\end{array}
$$

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):

$$
\begin{array}{llllll}
form & ::= & (form) & \mid & term = term & \mid \; \neg form \\
& \mid & form \wedge form & \mid & form \vee form & \mid \; form \longrightarrow form \\
& \mid & \forall x.\ form & \mid & \exists x.\ form &
\end{array}
$$

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):

$$
\begin{array}{llllll}
form & ::= & (form) & \mid\ term = term & \mid\ \neg form \\
 & \mid & form \wedge form & \mid\ form \vee form & \mid\ form \longrightarrow form \\
 & \mid & \forall x.\ form & \mid\ \exists x.\ form &
\end{array}
$$

Examples

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

Scope of quantifiers: as far to the right as possible

Formulae

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

Formulae

Abbreviation: $\forall x\, y.\ P\ x\ y \;\equiv\; \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Formulae

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

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \ \equiv\ A \wedge (B \wedge C)$

Formulae

Abbreviation: $\forall x\, y.\ P\, x\, y \;\equiv\; \forall x.\ \forall y.\ P\, x\, y \qquad (\forall, \exists, \lambda, \ldots)$

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:

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

Types and Terms

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & \mid & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types} \\
& | & \textit{'a} \mid \textit{'b} \mid \ldots & \text{type variables}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types} \\
& | & \textit{'a} \mid \textit{'b} \mid \ldots & \text{type variables} \\
& | & \tau \Rightarrow \tau & \text{total functions}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
 & | & \textit{bool} \mid \textit{nat} \mid \dots & \text{base types} \\
 & | & \textit{'a} \mid \textit{'b} \mid \dots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } \texttt{*}\text{)}
\end{array}
$$

Types

Syntax:

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

Types

Syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types} \\
& | & \textit{'a} \mid \textit{'b} \mid \ldots & \text{type variables} \\
& | & \tau \Rightarrow \tau & \text{total functions} \\
& | & \tau \times \tau & \text{pairs (ascii: } \texttt{*}\text{)} \\
& | & \tau \; \textit{list} & \text{lists} \\
& | & \ldots & \text{user-defined types}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & \textit{bool} \mid \textit{nat} \mid \ldots & \text{base types} \\
& | & \textit{'a} \mid \textit{'b} \mid \ldots & \text{type variables} \\
& | & \tau \Rightarrow \tau & \text{total functions} \\
& | & \tau \times \tau & \text{pairs (ascii: } \texttt{*}\text{)} \\
& | & \tau \; \textit{list} & \text{lists} \\
& | & \ldots & \text{user-defined types}
\end{array}
$$

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

Terms: Basic syntax

Syntax:

$$\begin{array}{lcll} term & ::= & (term) & \\ & \mid & a & \text{constant or variable (identifier)} \\ & \mid & term\ term & \text{function application} \\ & \mid & \lambda x.\ term & \text{function “abstraction”} \end{array}$$

Terms: Basic syntax

Syntax:

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

Terms: Basic syntax

Syntax:

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

Examples: *f (g x) y* *h (*λ*x. f (g x))*

Terms: Basic syntax

Syntax:

$$
\begin{array}{rcll}
term & ::= & (term) & \\
& \mid & a & \text{constant or variable (identifier)} \\
& \mid & term\ term & \text{function application} \\
& \mid & \lambda x.\ term & \text{function “abstraction”} \\
& \mid & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

Examples: *f (g x) y* *h (*λ*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"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

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"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

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

$\longrightarrow_\beta$ ***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 `::` τ 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' ::* $\tau_1 \times \tau_2 \Rightarrow \tau$

Curying

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*$_1$ :: τ_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:

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

Terms: Syntactic sugar

Some 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 _)* !

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

Type bool

Formulae = terms of type *bool*

True :: *bool*
False :: *bool*
$\wedge$, $\vee$, $\ldots$ :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge$, $\vee$, $\ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

if-and-only-if: =

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

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

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

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, \ldots :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

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

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

. . . unless the context is unambiguous: $Suc\ z$

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*")
  and rest *xs* ("*tail*")
- Syntactic sugar: *[x*$_1$*,...,x*$_n$*]*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*")
  and rest *xs* ("*tail*")
- Syntactic sugar: *[*$x_1, \ldots, x_n$*]*

Large library:
hd, *tl*, *map*, *length*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, . . .

Don't reinvent, reuse!
$\leadsto$ `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax:

```
theory MyTh
imports ImpTh_1 ... ImpTh_n
begin
(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 $MyTh$
imports $ImpTh_1 \ldots ImpTh_n$
begin
(declarations, definitions, theorems, proofs, ...)*
end

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

Usually:

theory $MyTh$
imports `Main`
$\vdots$

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”)

X-Symbols

Input of funny symbols in Proof General

- via menu ("X-Symbol")
- via ascii encoding (similar to LaTeX): \<and>, \<or>, ...
- via abbreviation: /\, \/, -->, ...

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

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

Demo: terms and types