Isabelle's meta-logic

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Basic constructs

Implication $\Longrightarrow$ (**==>**)

For separating premises and conclusion of theorems

Equality $\equiv$ (**==**)

For definitions

Basic constructs

Implication $\Longrightarrow$ (`==>`)

For separating premises and conclusion of theorems

Equality $\equiv$ (`==`)

For definitions

Universal quantifier $\bigwedge$ (`!!`)

Rarely needed

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Equality $\equiv$ (==)

For definitions

Universal quantifier $\bigwedge$ (!!)

Rarely needed

Do not use *inside* HOL formulae

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ “and”

The proof state

1. $\bigwedge x_1 \ldots x_p. \llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow B$

$x_1 \ldots x_p$	Local constants
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Type and function definition in Isabelle/HOL

Type definition in Isabelle/HOL

Introducing new types

Keywords:

- **typedecl**: pure declaration
- **types**: abbreviation
- **datatype**: recursive datatype

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

Example:

typedecl *addr*   *-- "An abstract type of addresses"*

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types

name = string

('a,'b)foo = 'a list $\times$ *'b list*

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types

name = string

('a,'b)foo = 'a list $\times$ *'b list*

Type abbreviations are expanded immediately after parsing
Not present in internal representation and Isabelle output

datatype

The example

datatype *'a list = Nil | Cons 'a "'a list"*

Properties:

- Types: *Nil* :: *'a list*

  Cons :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
- Distinctness: *Nil* $\neq$ *Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y* $\wedge$ *xs = ys)*

The general case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \quad = \quad C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$$
$$\quad | \quad \ldots$$
$$\quad | \quad C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The general case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \;|\; \ldots \;|\; C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$\Longrightarrow 0 = 1$

! All functions in HOL must be total **!**

Function definition schemas in Isabelle/HOL

- Non-recursive with **defs/constdefs**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **recdef**
  User must (help to) prove termination
  ($\leadsto$ later)

defs/constdefs

Definition (non-recursive) by example

Declaration:

consts

$sq :: nat \Rightarrow nat$

Definition:

defs

$sq\_def: sq\ n \ \equiv\ n*n$

Definition (non-recursive) by example

Declaration:

consts

sq :: nat $\Rightarrow$ *nat*

Definition:

defs

sq_def: sq n $\equiv$ *n*n*

Declaration + definition:

constdefs

sq :: nat $\Rightarrow$ *nat*

sq n $\equiv$ *n*n*

Definitions: pitfalls

constdefs

$prime :: nat \Rightarrow bool$

$prime\ p \ \equiv\ 1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p)$

Definitions: pitfalls

constdefs

$prime :: nat \Rightarrow bool$

$prime\ p \ \equiv\ 1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p)$

Not a definition: free *m* not on left-hand side

Definitions: pitfalls

constdefs

$prime :: nat \Rightarrow bool$

$prime\ p \equiv 1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p)$

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

Definitions: pitfalls

constdefs

$prime :: nat \Rightarrow bool$

$prime\ p \equiv 1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p)$

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

$prime\ p \equiv 1 < p \wedge (\forall m.\ m\ dvd\ p \longrightarrow m = 1 \vee m = p)$

Using definitions

Definitions are not used automatically

Using definitions

Definitions are not used automatically

Unfolding the definition of *sq*:

apply*(unfold sq_def)*

primrec

The example

primrec

$$
\begin{array}{lll}
app\ Nil & ys & = & ys \\
app\ (Cons\ x\ xs) & ys & = & Cons\ x\ (app\ xs\ ys)
\end{array}
$$

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then
$f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p \quad = \quad r_1$$

$$\vdots$$

$$f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p \quad = \quad r_k$$

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$
\begin{aligned}
f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p &= r_1 \\
&\vdots \\
f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p &= r_k
\end{aligned}
$$

The recursive calls in r_i must be *structurally smaller*, i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_p$

nat is a datatype

datatype $nat = 0 \mid Suc\ nat$

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec
f 0 = ...
f(Suc n) = ... f n ...

More predefined types and functions

Type option

datatype *’a option = None | Some ’a*

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\begin{array}{rcl} \ldots \Rightarrow \textit{'a option} & \approx & \text{partial function:} \\ \textit{None} & \approx & \text{no result} \\ \textit{Some a} & \approx & \text{result } \textit{a} \end{array}$$

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\begin{array}{rcl} \ldots \Rightarrow \textit{'a option} & \approx & \text{partial function:} \\ \textit{None} & \approx & \text{no result} \\ \textit{Some a} & \approx & \text{result } a \end{array}$$

Example:

consts *lookup* $:: \textit{'k} \Rightarrow (\textit{'k} \times \textit{'v})\ \textit{list} \Rightarrow \textit{'v option}$

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\begin{array}{rcl} \ldots \Rightarrow \textit{'a option} & \approx & \text{partial function:} \\ \textit{None} & \approx & \text{no result} \\ \textit{Some a} & \approx & \text{result } a \end{array}$$

Example:

consts *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option*

primrec

lookup k [] = None

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \quad \approx \quad \text{partial function:}$$

$$\begin{aligned} \textit{None} \quad &\approx \quad \text{no result} \\ \textit{Some a} \quad &\approx \quad \text{result } a \end{aligned}$$

Example:

consts *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option*

primrec

lookup k [] = None

lookup k (x#xs) =

(if fst x = k then Some(snd x) else lookup k xs)

case

Every datatype introduces a *case* construct, e.g.

$$(case\ xs\ of\ [] \Rightarrow \ldots \mid y\#ys \Rightarrow ...\ y\ ...\ ys\ ...)$$

case

Every datatype introduces a *case* construct, e.g.

$$(\textit{case } xs \textit{ of } [] \Rightarrow \ldots \mid y\#ys \Rightarrow \ldots\ y \ldots\ ys \ldots)$$

In general: one case per constructor

case

Every datatype introduces a *case* construct, e.g.

(case xs of [] $\Rightarrow \ldots$ | y#ys $\Rightarrow$ ... y ... ys ...)

In general: one case per constructor

Same order of cases as in datatype

case

Every datatype introduces a *case* construct, e.g.

$$(\mathit{case}\ xs\ \mathit{of}\ []\ \Rightarrow \ldots \mid y\#ys \Rightarrow \ldots\ y \ldots\ ys \ldots)$$

In general: one case per constructor

Same order of cases as in datatype

No nested cases

case

Every datatype introduces a *case* construct, e.g.

$$(\textit{case } xs \textit{ of } [] \Rightarrow \ldots \mid y\#ys \Rightarrow \ldots\ y \ldots\ ys \ldots)$$

In general: one case per constructor

Same order of cases as in datatype

No nested cases

Needs *( )* in context

Case distinctions

apply(*case_tac* t)

creates k subgoals

$$t = C_i \; x_1 \ldots x_p \Longrightarrow \ldots$$

one for each constructor C_i.

Demo: trees