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 $\bigwedge$ (!!)

For binding local variables

Basic constructs

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

For separating premises and conclusion of theorems

Equality $\equiv$ (`==`)

For definitions

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

For binding local variables

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

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

$$; \quad \approx \quad \text{“and”}$$

The proof state

1. $\bigwedge x_1 \ldots x_p.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$

$x_1 \ldots x_p$	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

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

- *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

datatype $(\alpha_1, \ldots, \alpha_n)\tau$ = $C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$

$\quad\quad\quad\quad\quad\quad\quad\quad | \quad \ldots$

$\quad\quad\quad\quad\quad\quad\quad\quad | \quad C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *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:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

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 **definition**
  No problem

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof
- Well-founded recursion with **function**
  User-supplied termination proof

definition

Definition (non-recursive) by example

definition $sq :: nat \Rightarrow nat$ **where** $sq\ n = n * n$

Definitions: pitfalls

definition $prime :: nat \Rightarrow bool$ **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

Definitions: pitfalls

definition *prime :: nat* $\Rightarrow$ *bool* **where**
prime p = (1 < p $\wedge$ *(m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

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

Definitions: pitfalls

definition $prime :: nat \Rightarrow bool$ **where**
$prime\ p = (1 < p \wedge (m\ dvd\ p \longrightarrow m = 1 \vee m = p))$

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

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

Definitions: pitfalls

definition *prime* :: *nat* $\Rightarrow$ *bool* **where**
prime p = (*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 = (*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 *app :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**

app Nil ys = ys |

app (Cons x xs) ys = Cons x (app xs ys)

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*:

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

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 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* | *Suc nat*

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec *f :: nat* $\Rightarrow$ *...*
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:

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

Type option

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

Important application:

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

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

Example:

consts *lookup :: 'k* ⇒ *('k* × *'v) list* ⇒ *'v option*

Type option

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

Important application:

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

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

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:

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

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

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

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

case

Datatype values can be taken apart with *case* expressions:

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

Wildcards:

$$(case\ xs\ of\ [] \Rightarrow [] \mid y\#\_ \Rightarrow [y])$$

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

case

Datatype values can be taken apart with *case* expressions:

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

Wildcards:

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

Nested patterns:

$$(\textit{case } xs \textit{ of } [0] \Rightarrow 0 \mid [\textit{Suc } n] \Rightarrow n \mid \_ \Rightarrow 2)$$

Complicated patterns mean complicated proofs!

case

Datatype values can be taken apart with *case* expressions:

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

Wildcards:

$$(case\ xs\ of\ [] \Rightarrow [] \mid y\#\_ \Rightarrow [y])$$

Nested patterns:

$$(case\ xs\ of\ [0] \Rightarrow 0 \mid [Suc\ n] \Rightarrow n \mid \_ \Rightarrow 2)$$

Complicated patterns mean complicated proofs!

Needs *( )* in context

Proof by case distinction

If t :: τ and τ is a datatype

apply(*case_tac* t)

Proof by case distinction

If t :: τ and τ is a datatype

apply(*case_tac* t**)**

creates k subgoals

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

one for each constructor C_i of type τ.

Demo: trees

fun

From primitive recursion to arbitrary pattern matching

Example: Fibonacchi

fun $fib :: nat \Rightarrow nat$ **where**

fib 0 = 0 |
fib (Suc 0) = 1 |
fib (Suc(Suc n)) = fib (n+1) + fib n

Example: Separation

fun *sep :: 'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**

sep a [] = [] |
sep a [x] = [x] |
sep a (x#y#zs) = x # a # sep a (y#zs)

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$
\begin{aligned}
&ack\ 0\ n = Suc\ n\ | \\
&ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0)\ | \\
&ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)
\end{aligned}
$$

Key features of fun

- Arbitrary pattern matching

Key features of fun

- Arbitrary pattern matching
- Order of equations matters

Key features of fun

- Arbitrary pattern matching
- Order of equations matters
- Termination must be provable
  by lexicographic combination of size measures

Size

- $size(n{::}nat) = n$

Size

- *size(n::nat) = n*
- *size(xs) = length xs*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*
- *size* counts number of (non-nullary) constructors

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5, 3) > (4, 7) > (4, 6) > (4, 0) > (3, 42) > \cdots$$

Similar for tuples:

$$(5, 6, 3) > (4, 12, 5) > (4, 11, 9) > (4, 11, 8) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Theorem If each component ordering terminates, then their *lexicographic product* terminates, too.

Ackermann terminates

$ack\ 0\ n = Suc\ n$

$ack\ (Suc\ m)\ 0 = ack\ m\ (Suc\ 0)$

$ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)$

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Note: order of arguments not important for Isabelle!

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

for each equation $f(e) = t$,
prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

> for each equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Induction follows course of (terminating!) computation

Computation Induction: Example

fun *div2 :: nat* $\Rightarrow$ *nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

Computation Induction: Example

fun *div2 :: nat* $\Rightarrow$ *nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

$\leadsto$ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Demo: fun