
Isabelle's meta-logic

Basic constructs

Implication \implies (\implies)

For separating premises and conclusion of theorems

Basic constructs

Implication \implies (\implies)

For separating premises and conclusion of theorems

Equality \equiv (\equiv)

For definitions

Basic constructs

Implication \implies (\implies)

For separating premises and conclusion of theorems

Equality \equiv (\equiv)

For definitions

Universal quantifier \wedge ($!!$)

For binding local variables

Basic constructs

Implication \implies (\implies)

For separating premises and conclusion of theorems

Equality \equiv (\equiv)

For definitions

Universal quantifier \wedge (\forall)

For binding local variables

Do not use *inside* HOL formulae

Notation

$$[A_1; \dots ; A_n] \implies B$$

abbreviates

$$A_1 \implies \dots \implies A_n \implies B$$

Notation

$$[A_1; \dots ; A_n] \implies B$$

abbreviates

$$A_1 \implies \dots \implies A_n \implies B$$

; \approx “and”

The proof state

$$1. \bigwedge x_1 \dots x_p. [A_1; \dots ; A_n] \implies B$$

$x_1 \dots x_p$ Local constants

$A_1 \dots A_n$ 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

typedefcl

typedefcl *name*

Introduces new “opaque” type *name* without definition

typedefcl

typedefcl *name*

Introduces new “opaque” type *name* without definition

Example:

typedefcl *addr* — An abstract type of addresses

types

types *name* = τ

Introduces an *abbreviation* *name* for type τ

types

types *name* = τ

Introduces an *abbreviation* *name* for type τ

Examples:

types

name = *string*

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

types

types *name* = τ

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 ⇒ 'a list ⇒ 'a list
- **Distinctness:** Nil ≠ Cons x xs
- **Injectivity:** (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)

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.

$$\implies 0 = 1$$

Why nontermination can be harmful

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

Subtract $f\ x$ on both sides.

$$\implies 0 = 1$$

! 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* = ($1 < p \wedge (m \text{ dvd } p \longrightarrow m = 1 \vee m = p)$)

Definitions: pitfalls

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

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

Definitions: pitfalls

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

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* = ($1 < p \wedge (m \text{ dvd } p \longrightarrow m = 1 \vee m = p)$)

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

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

prime *p* = ($1 < p \wedge (\forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p)$)

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, \dots, C_k) then $f :: \dots \Rightarrow \tau \Rightarrow \dots \Rightarrow \tau'$ can be defined by *primitive recursion*:

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

⋮

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

The general case

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

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

⋮

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

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

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:

... \Rightarrow 'a option \approx partial function:

None \approx no result

Some a \approx result a

Type option

datatype 'a option = None | Some 'a

Important application:

... \Rightarrow 'a option \approx partial function:

None \approx no result

Some a \approx result a

Example:

primrec lookup :: 'k \Rightarrow ('k \times 'v) list \Rightarrow 'v option where

Type option

datatype 'a option = None | Some 'a

Important application:

... \Rightarrow 'a option \approx partial function:

None \approx no result

Some a \approx result a

Example:

primrec lookup :: 'k \Rightarrow ('k \times 'v) list \Rightarrow 'v option where
lookup k [] = None

Type option

datatype 'a option = None | Some 'a

Important application:

... \Rightarrow 'a option \approx partial function:

None \approx no result

Some a \approx result a

Example:

primrec lookup :: 'k \Rightarrow ('k \times 'v) list \Rightarrow 'v option where

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 [] ⇒ ... | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [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:

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

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

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

Complicated patterns mean complicated proofs!

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)

Complicated patterns mean complicated proofs!

Needs () in context

Proof by case distinction

If $t :: \tau$ and τ is a datatype

`apply(case_tac t)`

Proof by case distinction

If $t :: \tau$ and τ is a datatype

`apply(case_tac t)`

creates k subgoals

$$t = C_i x_1 \dots x_p \implies \dots$$

one for each constructor C_i of type τ .

Demo: trees

fun

*From primitive recursion
to arbitrary pattern matching*

Example: Fibonacci

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 ⇒ 'a list ⇒ '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

ack 0 n = Suc n |
ack (Suc m) 0 = ack m (Suc 0) |
ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

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

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

Size

- $\text{size}(n::\text{nat}) = n$
- $\text{size}(xs) = \text{length } xs$

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:

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

Lexicographic ordering

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

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

Similar for tuples:

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

Lexicographic ordering

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

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

Similar for tuples:

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

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

$$\mathit{ack} \ 0 \ n = \mathit{Suc} \ n$$

$$\mathit{ack} \ (\mathit{Suc} \ m) \ 0 = \mathit{ack} \ m \ (\mathit{Suc} \ 0)$$

$$\mathit{ack} \ (\mathit{Suc} \ m) \ (\mathit{Suc} \ n) = \mathit{ack} \ m \ (\mathit{ack} \ (\mathit{Suc} \ m) \ n)$$

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

Ackermann terminates

$$\mathit{ack} \ 0 \ n = \mathit{Suc} \ n$$

$$\mathit{ack} \ (\mathit{Suc} \ m) \ 0 = \mathit{ack} \ m \ (\mathit{Suc} \ 0)$$

$$\mathit{ack} \ (\mathit{Suc} \ m) \ (\mathit{Suc} \ n) = \mathit{ack} \ m \ (\mathit{ack} \ (\mathit{Suc} \ m) \ n)$$

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)

\rightsquigarrow induction rule *div2*.induct:

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

Demo: fun