Isabelle’s meta-logic
Basic constructs

Implication $\implies (\implies)$
For separating premises and conclusion of theorems
Basic constructs

Implication $\implies (==>)$
  For separating premises and conclusion of theorems

Equality $\equiv (==)$
  For definitions
Basic constructs

Implication $\implies (\Rightarrow)$
For separating premises and conclusion of theorems

Equality $\equiv (\equiv)$
For definitions

Universal quantifier $\forall (\forall)$
For binding local variables
Basic constructs

Implication $\implies (\Rightarrow)$
For separating premises and conclusion of theorems

Equality $\equiv (\equiv)$
For definitions

Universal quantifier $\forall (\forall)$
For binding local variables

Do not use inside HOL formulae
Notation

\[
\begin{bmatrix} A_1; \ldots ; A_n \end{bmatrix} \iff B
\]

abbreviates

\[ A_1 \iff \ldots \iff A_n \iff B \]
Notation

\[
\begin{bmatrix}
A_1; \ldots ; A_n
\end{bmatrix} \Rightarrow B
\]

abbreviates

\[
A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B
\]

; \quad \approx \quad "and"
The proof state

1. $\wedge x_1 \ldots x_p. \left[ A_1; \ldots ; A_n \right] \Rightarrow B$

$x_1 \ldots x_p$ 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
**typedef**

**typedef** *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**

\[ \text{types } name = \tau \]

Introduces an *abbreviation* \( name \) for type \( \tau \)
Introduces an abbreviation \textit{name} for type $\tau$

Examples:

\begin{verbatim}
\texttt{types name = $\tau$}
\texttt{types}
\texttt{name = string}
\texttt{('a,'b)foo = 'a list \times 'b list}
\end{verbatim}
Introduces an abbreviation \textit{name} for type $\tau$

Examples:

\begin{verbatim}
  types
  \textit{name} = string
  (\textquote{\texttt{a}}, \textquote{\texttt{b}})foo = \textquote{\texttt{a list}} \times \textquote{\texttt{b list}}
\end{verbatim}

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)
The general case

datatype \((\alpha_1, \ldots , \alpha_n)\tau\) = 
\begin{align*}
C_1 \tau_{1,1} & \cdots \tau_{1,n_1} \\
\vdots
\end{align*}
\begin{align*}
C_k \tau_{k,1} & \cdots \tau_{k,n_k}
\end{align*}

- **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:**
\[
(C_i \ x_1 \ldots x_{n_i} = C_i \ y_1 \ldots y_{n_i}) = (x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_i})
\]
The general case

datatype \((\alpha_1, \ldots, \alpha_n)\) = \(C_1 \tau_{1,1} \cdots \tau_{1,n_1}
\| \cdots \| \| C_k \tau_{k,1} \cdots \tau_{k,n_k}

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

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.

\[ \Rightarrow 0 = 1 \]
Why nontermination can be harmful

How about \( f(x) = f(x) + 1 \)?

Subtract \( f(x) \) on both sides.

\[ \Rightarrow 0 = 1 \]

! 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 (non-recursive) by example

\textbf{definition} \textit{sq :: nat \Rightarrow nat} \textbf{where} \textit{sq n = n \ast n}
**Definitions: pitfalls**

**definition** \textit{prime} :: nat \to bool where \\
\textit{prime} \ p = (1 < p \land (m \text{ dvd} \ p \to m = 1 \lor m = p))
Definitions: pitfalls

**Definition** $\text{prime} :: \text{nat} \Rightarrow \text{bool}$ where

$$\text{prime } p = (1 < p \land (m \text{ dvd } p \rightarrow m = 1 \lor m = p))$$

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

\[\text{definition} \ prime :: \text{nat} \Rightarrow \text{bool} \ \text{where} \]
\[\prime p = (1 < p \land (m \text{ dvd } p \rightarrow m = 1 \lor m = p))\]

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

**Every free variable on the rhs must occur on the lhs**
**Definitions: pitfalls**

```haskell
definition prime :: nat ⇒ bool where
prime p = (1 < p ∧ (m dvd p → m = 1 ∨ m = p))
```

Not a definition: free \( m \) not on left-hand side

![ ]

Every free variable on the rhs must occur on the lhs

```haskell
prime p = (1 < p ∧ (∀ m. m dvd p → m = 1 ∨ m = p))
```
Using definitions

Definitions are not used automatically
Definitions are not used automatically

Unfolding the definition of \( sq \):

\[ \text{apply}(unfold \ sq\_def) \]
primrec
The example

primrec app :: 'a list ⇒ 'a list ⇒ 'a list where
app Nil ys = ys |
app (Cons x xs) ys = Cons x (app xs ys)
The general case

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

\[
\begin{align*}
  f \ x_1 \cdots (C_1 \ y_{1,1} \cdots y_{1,n_1}) \cdots x_p &= r_1 \mid \\
  \vdots \\
  f \ x_1 \cdots (C_k \ y_{k,1} \cdots y_{k,n_k}) \cdots x_p &= r_k
\end{align*}
\]
The general case

If $\tau$ 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*:

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

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 \texttt{nat} = 0 \mid \texttt{Suc nat}
nat is a datatype

datatype $nat = 0 \mid Suc\ n$

Functions on $nat$ definable by primrec!

primrec $f :: nat \Rightarrow \ldots$

$f\ 0 = \ldots$

$f(Suc\ n) = \ldots f\ n \ldots$
More predefined types and functions
**Type option**

```ocaml
datatype 'a option = None | Some 'a
```
Type option

datatype 'a option = None | Some 'a

Important application:

\[ \ldots \Rightarrow \ 'a\ option \ \approx \ \text{partial function:} \]

\[ \text{None} \ \approx \ \text{no result} \]

\[ \text{Some} \ a \ \approx \ \text{result} \ a \]
**Type option**

datatype \( \texttt{'}a \text{ option} = \texttt{None} \mid \texttt{Some 'a} \)

Important application:

\[ \ldots \Rightarrow \texttt{'}a \text{ option} \approx \text{ partial function}: \]

\[
\begin{align*}
\texttt{None} & \approx \text{ no result} \\
\texttt{Some a} & \approx \text{ result } a
\end{align*}
\]

Example:

consts \texttt{lookup :: 'k} \Rightarrow (\texttt{'k} \times \texttt{'v}) \text{ list} \Rightarrow \texttt{'v option}
Type option

datatype ’a option = None | Some ’a

Important application:

\[ \ldots \Rightarrow \text{’a option} \approx \text{partial function:} \]

\[
\begin{align*}
None & \approx \text{no result} \\
\text{Some } a & \approx \text{result } a
\end{align*}
\]

Example:

consts lookup :: ’k ⇒ (’k × ’v) list ⇒ ’v option

primrec

lookup k [] = None
**Type option**

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

Important application:

```
... ⇒ 'a option ≈ partial function:

None ≈ no result
Some a ≈ result a
```

Example:

```
consts lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option
primrec
lookup k [] = None
lookup k (x#xs) =
  (if fst x = k then Some(snd x) else lookup k xs)
```
Datatype values can be taken apart with case expressions:

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

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

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

Wildcards:

\[(\text{case } xs \text{ of } [] \Rightarrow [] \mid y\#_\_ \Rightarrow [y])\]
Datatype values can be taken apart with case expressions:

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

Wildcards:

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

Nested patterns:

\[(\text{case } xs \text{ of } [0] \Rightarrow 0 \mid [\text{Suc } n] \Rightarrow n \mid _\Rightarrow 2)\]
Datatype values can be taken apart with case expressions:

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

Wildcards:

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

Nested patterns:

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

Complicated patterns mean complicated proofs!
Datatype values can be taken apart with case expressions:

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

Wildcards:

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Needs ( ) in context
Proof by case distinction

If $t :: \tau$ and $\tau$ is a datatype

apply($\text{case_tac } t$)
Proof by case distinction

If $t :: \tau$ and $\tau$ is a datatype

\[
\text{apply(} \text{case_tac } t \text{)}
\]

creates $k$ subgoals

\[
t = C_i \ x_1 \ldots x_p \implies \ldots
\]

one for each constructor $C_i$ of type $\tau$. 

- p.27
Demo: trees
fun

*From primitive recursion to arbitrary pattern matching*
Example: Fibonacci

fun fib :: nat ⇒ 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 ⇒ nat ⇒ 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

- $size(n::nat) = n$
Size

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

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

\begin{align*}
    \text{ack } 0 \ n &= \text{Suc } n \\
    \text{ack } (\text{Suc } m) \ 0 &= \text{ack } m \ (\text{Suc } 0) \\
    \text{ack } (\text{Suc } m) \ (\text{Suc } n) &= \text{ack } m \ (\text{ack } (\text{Suc } m) \ n)
\end{align*}
Ackermann terminates

\[ \text{ack} \ 0 \ n = \text{Succ} \ n \]

\[ \text{ack} \ (\text{Succ} \ m) \ 0 = \text{ack} \ m \ (\text{Succ} \ 0) \]

\[ \text{ack} \ (\text{Succ} \ m) \ (\text{Succ} \ n) = \text{ack} \ m \ (\text{ack} \ (\text{Succ} \ m) \ n) \]

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

\[ \text{ack} \ 0 \ n = \ Suc \ n \]
\[ \text{ack} \ (Suc \ m) \ 0 = \ text{ack} \ m \ (Suc \ 0) \]
\[ \text{ack} \ (Suc \ m) \ (Suc \ n) = \ ack \ m \ (ack \ (Suc \ m) \ n) \]

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

Note: order of arguments not important for Isabelle!
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 $t$. 
If $f :: \tau \Rightarrow \tau'$ is defined by \texttt{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

\[
\text{fun div2 :: nat } \Rightarrow \text{ nat where}
\]
\[
div2 \ 0 = 0 \ |
\]
\[
div2 \ (\text{Suc} \ 0) = 0 \ |
\]
\[
div2(\text{Suc}(\text{Suc} \ n)) = \text{Suc}(\text{div2} \ n)
\]
**Computation Induction: Example**

\[
\text{fun } \text{div2} :: \text{nat } \Rightarrow \text{nat} \text{ where } \\
\text{div2 } 0 = 0 \mid \\
\text{div2 } (}\text{Suc } 0) = 0 \mid \\
\text{div2} (}\text{Suc}(\text{Suc } n)) = \text{Suc}(\text{div2 } n)
\]

\[ \leadsto \text{induction rule } \text{div2} . \text{induct} : \\
P(0) \quad P(\text{Suc } 0) \quad P(n) \quad \Rightarrow \quad P(\text{Suc}(\text{Suc } n))
\]

\[ \quad \Rightarrow \quad P(m) \]
Demo: fun