Isabelle’s meta-logic
Basic constructs

Implication $\implies (==>)$

For separating premises and conclusion of theorems
Basic constructs

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

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

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

Equality $\equiv (==)$
   For definitions

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

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

Equality $\equiv (=)$
For definitions

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

Do not use *inside* HOL formulae
Notation

\[
\left[ A_1 ; \ldots ; A_n \right] \Rightarrow B
\]

abbreviates

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

\[ \left[ A_1 ; \ldots ; A_n \right] \Rightarrow B \]

abbreviates

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

; \; \approx \; \text{“and”}
The proof state

1. \( \bigwedge x_1 \ldots x_p. \left[ A_1; \ldots ; A_n \right] \Rightarrow B \)

- \( x_1 \ldots x_p \) 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:

- `typedef`: pure declaration
- `types`: abbreviation
- `datatype`: recursive datatype
typedecl

Introduces new “opaque” type \textit{name} without definition
**typedecl**

**typedecl** *name*

Introduces new “opaque” type *name* without definition

Example:

**typedecl** *addr*  — An abstract type of addresses
types $name = \tau$

Introduces an abbreviation $name$ for type $\tau$
**types**

**types** $name = \tau$

Introduces an *abbreviation* $name$ for type $\tau$

Examples:

**types**

$\begin{align*}
  name &= \text{string} \\
  (\text{'a,'b})foo &= \text{'a list} \times \text{'b list}
\end{align*}$
**types**

**types** \( name = \tau \)

Introduces an *abbreviation* \( name \) for type \( \tau \)

Examples:

**types**

\[
name = \text{string} \\
('a,'b)foo = 'a \text{ list} \times 'b \text{ list}
\]

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:**  
  \[(Cons x xs = Cons y ys) = (x = y \land xs = ys)\]
The general case

**datatype** \((\alpha_1, \ldots, \alpha_n)\tau\) = \(C_1 \tau_{1,1} \cdots \tau_{1,n_1}\)  
\[\vdots\]  
\(\vdots\)  
\(C_k \tau_{k,1} \cdots \tau_{k,n_k}\)

- **Types:** \(C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau\)
- **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 \((\alpha_1, \ldots, \alpha_n)\tau = C_1 \tau_{1,1} \cdots \tau_{1,n_1} \quad | \quad \ldots \quad | \quad C_k \tau_{k,1} \cdots \tau_{k,n_k}\)

- **Types:** \(C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau\)
- **Distinctness:** \(C_i \ldots \neq C_j \ldots \text{ if } i \neq j\)
- **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.

$$\implies 0 = 1$$
**Why nontermination can be harmful**

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

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

\[
\Rightarrow 0 = 1
\]

\![ ] \text{All functions in HOL must be total} \!

– p.14
Non-recursive with definition
No problem
Non-recursive with `definition`  
No problem

Primitive-recursive with `primrec`  
Terminating by construction
Function definition schemas in Isabelle/HOL

- Non-recursive with \texttt{definition}
  No problem

- Primitive-recursive with \texttt{primrec}
  Terminating by construction

- Well-founded recursion with \texttt{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

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

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

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

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

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
The definition of the prime predicate is:

\[
\text{prime} \colon \text{nat} \to \text{bool} \quad \text{where} \quad \text{prime} \ p = (1 < p \land (m \ \text{dvd} \ p \to m = 1 \lor m = p))
\]

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

\[
\text{prime} \ p = (1 < p \land (\forall m. \ m \ \text{dvd} \ p \to m = 1 \lor m = p))
\]
Definitions are not used automatically
Using definitions

Definitions are not used automatically

Unfolding the definition of $sq$:

\texttt{apply}\((unfold\ sq\_def)\)
primrec
The example

primrec

\[\text{app \, Nil \, ys} = \, \text{ys}\]

\[\text{app \, (Cons \, x \, xs) \, ys} = \, \text{Cons \, x \, (app \, xs \, ys)}\]
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 primitive recursion:

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

$$\vdots$$

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

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

$$\vdots$$

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

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 \textit{nat} = 0 \mid \textit{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**

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

Important application:

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

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

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

Important application:

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

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

\[ \text{Some } a \cong \text{result } a \]

Example:

```consts lookup :: 'k \Rightarrow ('k × 'v) list \Rightarrow 'v option```
Type option

datatype ’a option = None | Some ’a

Important application:

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

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

Example:

consts lookup :: ’k ⇒ (’k × ’v) list ⇒ ’v option
primrec
lookup k [] = None
**Type option**

```plaintext
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:

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

\textit{case}
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:

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

Complicated patterns mean complicated proofs!

Needs ( ) in context
Case distinctions

If \( t :: \tau \) and \( \tau \) is a datatype

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

creates \( k \) subgoals

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

one for each constructor \( C_i \) of type \( \tau \).
Demo: trees
fun

*From primitive recursion to arbitrary pattern matching*
fun fib :: nat ⇒ nat where

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

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

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

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

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

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

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

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 \texttt{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 \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 \( t \).
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
fun div2 :: nat ⇒ nat where
  div2 0 = 0 |
  div2 (Suc 0) = 0 |
  div2(Suc(Suc n)) = Suc(div2 n)
fun div2 :: nat ⇒ nat where
  div2 0 = 0 |
  div2 (Suc 0) = 0 |
  div2(Suc(Suc n)) = Suc(div2 n)

⇒ induction rule div2.induct:

\[
\begin{array}{c}
P(0) \quad P(Suc 0) \quad P(n) \\
\hline
P(Suc(Suc n)) \quad P(m)
\end{array}
\]
Demo: fun