Programming and Proving in

Tobias Nipkow

Fakultät für Informatik
TU München
Notation

Implication associates to the right:

\[ A \implies B \implies C \quad \text{means} \quad A \implies (B \implies C) \]

Similarly for other arrows: \( \Rightarrow, \xrightarrow{\quad} \)

\[
\frac{A_1 \ldots A_n}{B} \quad \text{means} \quad A_1 \implies \ldots \implies A_n \implies B
\]
1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction and Simplification

4. Logic and Proof beyond “=”

5. Isar: A Language for Structured Proofs
HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:
- For the moment: only $\text{term} = \text{term}$,
  e.g. $1 + 2 = 4$
- Later: $\land$, $\lor$, $\rightarrow$, $\forall$, ...
Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types bool, nat and list

Summary
Types

Basic type syntax:

\[ \tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \ldots \text{ base types} \]
\[ \mid 'a \mid 'b \mid \ldots \text{ type variables} \]
\[ \mid \tau \Rightarrow \tau \text{ functions} \]
\[ \mid \tau \times \tau \text{ pairs (ascii: *)} \]
\[ \mid \tau \text{ list} \text{ lists} \]
\[ \mid \tau \text{ set} \text{ sets} \]
\[ \mid \ldots \text{ user-defined types} \]

Convention: \[ \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3) \]
Terms

Terms can be formed as follows:

- **Function application:**
  \[ f \ t \]
  is the call of function \( f \) with argument \( t \).
  If \( f \) has more arguments: \( f \ t_1 \ t_2 \ldots \)
  Examples: \( \sin \pi, \ plus \ x \ y \)

- **Function abstraction:**
  \[ \lambda x. \ t \]
  is the function with parameter \( x \) and result \( t \),
  i.e. \( x \mapsto t \).
  Example: \( \lambda x. \ plus \ x \ x \)
Basic term syntax:

\[ t ::= (t) \]

| \[ a \] constant or variable (identifier) |
| \[ tt \] function application |
| \[ \lambda x. t \] function abstraction |
| \[ \ldots \] lots of syntactic sugar |

Examples: \[ f (g x) y \]

\[ h (\lambda x. f (g x)) \]

Convention: \[ f t_1 t_2 t_3 \equiv ((f t_1) t_2) t_3 \]

This language of terms is known as the \( \lambda \)-calculus.
The computation rule of the $\lambda$-calculus is the replacement of formal by actual parameters:

$$(\lambda x. t) \ u \ = \ t[u/x]$$

where $t[u/x]$ is “$t$ with $u$ substituted for $x$”.

Example: $(\lambda x. x + 5) \ 3 \ = \ 3 + 5$

- The step from $(\lambda x. t) \ u$ to $t[u/x]$ is called $\beta$-reduction.
- Isabelle performs $\beta$-reduction automatically.
Terms must be well-typed
(the argument of every function call must be of the right type)

Notation:
\[ t :: \tau \] means “\( t \) is a well-typed term of type \( \tau \).”

\[
\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2}
\]
Isabelle automatically computes the type of each variable in a term. This is called type inference.

In the presence of overloaded functions (functions with multiple types) this is not always possible.

User can help with type annotations inside the term. Example: \( f (x::nat) \)
Currying

Thou shalt Curry your functions

- Curried: \( f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau \)
- Tupled: \( f' :: \tau_1 \times \tau_2 \Rightarrow \tau \)

Advantage:

Currying allows *partial application*

\[ f \ a_1 \quad \text{where} \quad a_1 :: \tau_1 \]
Predefined syntactic sugar

- **Infix**: +, −, *, #, @, ...
- **Mixfix**: if _ then _ else _, case _ of, ...

Prefix binds more strongly than infix:

\[ f x + y \equiv (f x) + y \not\equiv f (x + y) \]

Enclose if and case in parentheses:

\[ (if _ then _ else _) \]
Isabelle text = Theory = Module

Syntax:  
theory \textit{MyTh}  
imports \textit{ImpTh_1} \ldots \textit{ImpTh_n}  
begin  
(definitions, theorems, proofs, \ldots)^*  
end  

\textit{MyTh}: name of theory. Must live in file \textit{MyTh.thy}  
\textit{ImpTh_i}: name of \textit{imported} theories. Import transitive.  

Usually: \textit{imports} Main
Chapter 1: Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types \texttt{bool}, \texttt{nat} and \texttt{list}

Summary
Proof General

An Isabelle Interface

by David Aspinall
Customized version of \texttt{(x)emacs}:

- all of Emacs
- Isabelle aware (when editing \texttt{.thy} files)
- mathematical symbols ("x-symbols")
X-Symbols

Input of funny symbols

• via abbreviation: =>, ===>, \&, \|, \ldots
• via ascii encoding (similar to \texttt{\LaTeX}): \texttt{\&and\&}, \ldots
• via menu ("X-Symbol")
isabelle jedit

Similar to ProofGeneral but

- based on jedit
- \(\Rightarrow\) easier to install
- \(\Rightarrow\) may be more familiar
- Warning: still experimental \ldots
Concrete syntax

In .thy files:
Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides
Overview_Demo.thy
1 Overview of Isabelle/HOL

Types and terms
Interfaces
By example: types \textit{bool}, \textit{nat} and \textit{list}
Summary
Type `bool`

**datatype** \( \text{bool} = True \mid False \)

Predefined functions:
\( \land, \lor, \rightarrow, \ldots \) :: \( \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \)

A logical formula is a term of type `bool`

if-and-only-if: \( = \)
Type \textit{nat}

\textbf{datatype} \quad \textit{nat} = 0 \mid \textit{Suc} \textit{nat}

Values of type \textit{nat}: 0, \textit{Suc} 0, \textit{Suc}(\textit{Suc} 0), \ldots

Predefined functions: +, \ast, \ldots :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}

\textbf{!} Numbers and arithmetic operations are overloaded:

0, 1, 2, \ldots :: 'a, \quad + :: 'a \Rightarrow 'a \Rightarrow 'a

You need type annotations: 1 :: \textit{nat}, x + (y::\textit{nat})

\ldots unless the context is unambiguous: \textit{Suc} z
Nat_Demo.thy
Type 'a list

Lists of elements of type 'a
datatype 'a list = Nil | Cons 'a ('a list)

Syntactic sugar:
- [] = Nil: empty list
- x # xs = Cons x xs: list with first element x ("head") and rest xs ("tail")
- [x_1, ..., x_n] = x_1 # ... # x_n # []
Structural Induction for lists

To prove that $P(xs)$ for all lists $xs$, prove

- $P([])$ and
- for arbitrary $x$ and $xs$, $P(xs)$ implies $P(x\#xs)$.

\[
\begin{align*}
P([]) & \quad \land x \; xs. \; P(xs) \quad \Longrightarrow \quad P(x\#xs) \\
\hline
P(xs) & \quad P(xs)
\end{align*}
\]
List_Demo.thy
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!

Predefined: \( xs \odot ys \) (append), \textit{length}, and \textit{map}:

\[
\text{map } f \[x_1, \ldots, x_n\] = [f x_1, \ldots, f x_n]
\]

\begin{verbatim}
fun map :: ('a => 'b) => 'a list => 'b list where
  map f [] = [] |
  map f (x#xs) = f x # map f xs
\end{verbatim}

Note: \textit{map} takes \textit{function} as argument.
1 Overview of Isabelle/HOL

Types and terms
Interfaces
By example: types \textit{bool}, \textit{nat} and \textit{list}

Summary
• **datatype** defines (possibly) recursive data types.

• **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.
Proof methods

- \textit{induct} performs structural induction on some variable (if the type of the variable is a datatype).

- \textit{auto} solves as many subgoals as it can, mainly by simplification (symbolic evaluation):
  
  \[=\] is used only from left to right!
Proofs

General schema:

\textbf{lemma} \textit{name}: "..."
apply (...)
apply (...)
:
done

If the lemma is suitable as a simplification rule:

\textbf{lemma} \textit{name}[\texttt{simp}]: "..."
Top down proofs

Command

\texttt{sorry}

“completes” any proof.

Allows top down development:

\begin{quote}
\textit{Assume lemma first, prove it later.}
\end{quote}
The proof state

1. $\bigwedge x_1 \ldots x_p. \ A \Rightarrow B$

$x_1 \ldots x_p$ fixed local variables

$A$ local assumption(s)

$B$ actual (sub)goal
Multiple assumptions

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

abbreviates

\[ A_1 \implies \ldots \implies A_n \implies B \]

; \quad \approx \quad "\text{and}"
1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction and Simplification

4. Logic and Proof beyond “=”

5. Isar: A Language for Structured Proofs
2 Type and function definitions

Type definitions

Function definitions
Type synonyms

def type_synonym name = \tau

Introduces an synonym name for type \tau

Examples:

def type_synonym string = char list

def type_synonym ('a,'b)foo = 'a list \times 'b list

Type synonyms are expanded after parsing and are not present in internal representation and output
\textbf{datatype — the general case}

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

- \textbf{Types:} \(C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau\)

- \textbf{Distinctness:} \(C_i \ldots \neq C_j \ldots\) if \(i \neq j\)

- \textbf{Injectivity:} \((C_i \ x_1 \ldots x_{n_i} = C_i \ y_1 \ldots y_{n_i}) = (x_1 = y_1 \land \cdots \land x_{n_i} = y_{n_i})\)

Distinctness and injectivity are applied automatically
Induction must be applied explicitly
Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards:  

\[
(case \ m \ of \ 0 \ \Rightarrow \ Suc \ 0 \ \mid \ Suc \ _ \ \Rightarrow \ 0)
\]

Nested patterns:

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

Complicated patterns mean complicated proofs!

Need ( ) in context
Tree_Demo.thy
Type and function definitions

Type definitions

Function definitions
Non-recursive definitions

Example:

\texttt{definition} \quad \texttt{sq :: nat \Rightarrow nat where} \quad \texttt{sq n = n*n}

No pattern matching, just \quad f \quad x_1 \ldots \quad x_n \quad = \ldots
Nontermination can kill

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

! All functions in HOL must be total!
Key features of `fun`

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema
Example: separation

```
fun sep :: 'a ⇒ 'a list ⇒ 'a list where
sep a (x#y#zs) = x # a # sep a (y#zs) |
sep a xs = xs
```
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)

Terminates because the arguments decrease lexicographically with each recursive call:
• (Suc m, 0) > (m, Suc 0)
• (Suc m, Suc n) > (Suc m, n)
• (Suc m, Suc n) > (m, _)
1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction and Simplification
4. Logic and Proof beyond “=”
5. Isar: A Language for Structured Proofs
3 Induction and Simplification

Induction

Simplification
Basic induction heuristics

Theorems about recursive functions are proved by induction

Induction on argument number $i$ of $f$ if $f$ is defined by recursion on argument number $i$
A tail recursive reverse

Our initial reverse:

fun rev :: 'a list ⇒ 'a list where
  rev [] = [] |
  rev (x#xs) = rev xs @ [x]

A tail recursive version:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
  itrev [] ys = ys |
  itrev (x#xs) ys =

lemma itrev xs [] = rev xs
Induct_Demo.thy

Generalisation
Generalisation

- Replace constants by variables
- Generalize free variables
  - by $\forall$ in formula
  - by arbitrary in induction proof
So far, all proofs were by structural induction because all functions were primitive recursive.
In each induction step, 1 constructor is added. In each recursive call, 1 constructor is removed.
Now: induction for complex recursion patterns.
fun div2 :: nat ⇒ nat where

\(\text{div2}\ 0 = 0\)  \\
\(\text{div2}\ (\text{Suc}\ 0) = 0\)  \\
\(\text{div2}(\text{Suc}(\text{Suc}\ n)) = \text{Suc}(\text{div2}\ n)\)

\(\leadsto\) induction rule div2.induct:

\[
\begin{array}{lll}
P(0) & P(\text{Suc}\ 0) & P(n) \\
\hline
\implies & & P(\text{Suc}(\text{Suc}\ n)) \\
\implies & & P(m)
\end{array}
\]
Computation Induction

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

\[ \text{for each defining equation} \]

\[ f(e) = \ldots f(r_1) \ldots f(r_k) \ldots \]

\( P(e) \) assuming \( P(r_1), \ldots, P(r_k) \).

Induction follows course of (terminating!) computation

Motto: properties of \( f \) are best proved by rule \texttt{f.induct}
How to apply \textit{f.induct}

If $f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau'$:

$$(\text{induct } a_1 \ldots a_n \text{ rule: } f.\text{induct})$$

Heuristic:

- there should be a call $f a_1 \ldots a_n$ in your goal
- ideally the $a_i$ should be variables.
Induct_Demo.thy

Computation Induction
Induction and Simplification

Induction

Simplification
Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ simplification rule

Simplification $\equiv$ (Term) Rewriting
An example

Equations:

\[ 0 + n = n \]  \hspace{0.5cm} (1)

\[ (\text{Suc } m) + n = \text{Suc } (m + n) \]  \hspace{0.5cm} (2)

\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \]  \hspace{0.5cm} (3)

\[ (0 \leq m) = True \]  \hspace{0.5cm} (4)

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{0.5cm} (1) \equiv

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{0.5cm} (2) \equiv

\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \]  \hspace{0.5cm} (3) \equiv

\[ 0 \leq 0 + x \]  \hspace{0.5cm} (4) \equiv

True
Conditional rewriting

Simplification rules can be conditional:

\[
\begin{bmatrix}
P_1; \ldots; P_k
\end{bmatrix} \implies l = r
\]

is applicable only if all \(P_i\) can be proved first, again by simplification.

Example:

\[
p(0) = \text{True} \\
p(x) \implies f(x) = g(x)
\]

We can simplify \(f(0)\) to \(g(0)\) but we cannot simplify \(f(1)\) because \(p(1)\) is not provable.
Termination

Simplification may not terminate. Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), \ g(x) = f(x) \)

\[
[\ P_1; \ldots; \ P_k \ ] \implies l = r
\]

is suitable as a simp-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[
n < m \implies (n < \text{Suc } m) = \text{True} \quad \text{YES}
\]
\[
\text{Suc } n < m \implies (n < m) = \text{True} \quad \text{NO}
\]
Proof method $\text{simp}$

Goal: 1. $[P_1; \ldots; P_m] \Rightarrow C$

apply($\text{simp add: eq}_1 \ldots \text{eq}_n$)

Simplify $P_1 \ldots P_m$ and $C$ using

- lemmas with attribute $\text{simp}$
- rules from $\text{fun}$ and $\text{datatype}$
- additional lemmas $\text{eq}_1 \ldots \text{eq}_n$
- assumptions $P_1 \ldots P_m$

Variations:

- $(\text{simp} \ldots \text{del:} \ldots)$ removes $\text{simp}$-lemmas
- $\text{add}$ and $\text{del}$ are optional
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:
  
  (auto simp add: ... simp del: ...)

- auto acts on all subgoals
- simp acts only on subgoal 1
- auto applies simp and more
- auto can also be modified:
  
  (auto simp add: ... simp del: ...)

66
Rewriting with definitions

Definitions **(definition)** must be used *explicitly*:

\[(\texttt{simp add: } f\_\texttt{def} \ldots )\]

\(f\) is the function whose definition is to be unfolded.
Case splitting with *simp*

Automatic:

\[
P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))
\]

By hand:

\[
P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \ e = \text{Suc } n \rightarrow P(b))
\]

Proof method: *(simp split: nat.split)*

Or *auto*. Similar for any datatype *t*: *t.split*
Simp_Demo.thy
1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction and Simplification
4. Logic and Proof beyond “=”
5. Isar: A Language for Structured Proofs
4 Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
Syntax (in decreasing precedence):

\[
\text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \quad | \quad \neg \text{form} \\
\quad | \quad \text{form} \land \text{form} \quad | \quad \text{form} \lor \text{form} \quad | \quad \text{form} \rightarrow \text{form} \\
\quad | \quad \forall x. \text{form} \quad | \quad \exists x. \text{form}
\]

Examples:

\[
\neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \\
s = t \land C \equiv (s = t) \land C \\
A \land B = B \land A \equiv A \land (B = B) \land A \\
\forall x. \ P \ x \land Q \ x \equiv \forall x. (P \ x \land Q \ x)
\]

Input syntax: \(\longleftrightarrow\) (same precedence as \(\rightarrow\))
Warning

Quantifiers have low precedence and need to be parenthesized (if in some context)

\[ P \land \forall x. Q x \not\sim P \land (\forall x. Q x) \]
X-Symbols

... and their ascii representations:

\( \forall \) \( \langle \text{forall} \rangle \) ALL
\( \exists \) \( \langle \text{exists} \rangle \) EX
\( \lambda \) \( \langle \text{lambda} \rangle \) %
\( \rightarrow \) -->
\( \leftrightarrow \) <--->
\( \wedge \) \&
\( \lor \) |
\( \neg \) ~
\( \neq \) ~=
Sets over type `'a'

'\text{\texttt{a set}} = 'a \Rightarrow \text{bool}

- \{\}, \{e_1, \ldots, e_n\}, \{x. P x\}
- e \in A, \ A \subseteq B
- A \cup B, \ A \cap B, \ A - B, \ - A
- ...

\in \ \texttt{\langle in\rangle} : \texttt{\langle subseteq\rangle} \leq\texttt{\langle union\rangle} \texttt{\langle inter\rangle} \texttt{\langle subseteq\rangle} \texttt{\langle union\rangle} \texttt{\langle inter\rangle}
4 Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
**simp and auto**

**simp**: rewriting and a bit of arithmetic

**auto**: rewriting and a bit of arithmetic, logic and sets

- Show you where they got stuck
- Highly incomplete
- Extensible with new *simp*-rules

Exception: *auto* acts on all subgoals
fastsimp

- rewriting, logic, sets, relations and a bit of arithmetic.
- **incomplete** but better than *auto*.
- Succeeds or fails
- Extensible with new *simp*-rules
• A **complete** proof search procedure for FOL . . .
• . . . but (almost) **without** “=”
• Covers logic, sets and relations
• Succeeds or fails
• Extensible with new deduction rules
Automating arithmetic

arith:

• proves linear formulas (no “∗”)  
• complete for quantifier-free real arithmetic  
• complete for first-order theory of nat and int (Presburger arithmetic)
Sledgehammer
Architecture:

Isabelle

Formula
& filtered library

Proof
lemmas used

external

ATPs\textsuperscript{1}

Characteristics:

- Sometimes it works,
- Sometimes it doesn’t.

Do you feel lucky?

\textsuperscript{1}Automatic Theorem Provers
by \((proof\text{-}method)\)

\[\approx\]

apply \((proof\text{-}method)\)
done
Auto_Proof_Demo.thy
4 Logic and Proof beyond “=”

Logical Formulas
Proof Automation
Single Step Proofs
Inductive Definitions
Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.
What are these ?-variables ?

After you have finished a proof, Isabelle turns all free variables \( V \) in the theorem into \(?V\).

Example: theorem \( \text{conjI} \): \([\,?P\,;\,?Q\,] \imp ?P \land ?Q\)

These ?-variables can later be instantiated:

- **By hand:**
  \[
  \text{conjI[of "a=b" "False"] } \sim \\Rightarrow
  \] \[
  [a = b; False] \imp a = b \land False
  \]

- **By unification:**
  unifying \(?P \land ?Q\) with \( a=b \land False\)
  sets \(?P\) to \( a=b\) and \(?Q\) to \( False\).
Rule application

Example: rule: $[?P; ?Q] \Rightarrow ?P \land ?Q$

subgoal: 1. ... $\Rightarrow A \land B$

Result: 1. ... $\Rightarrow A$
2. ... $\Rightarrow B$

The general case: applying rule $[ A_1; \ldots ; A_n ] \Rightarrow A$
to subgoal ... $\Rightarrow C$:

- Unify $A$ and $C$
- Replace $C$ with $n$ new subgoals $A_1 \ldots A_n$

apply(*rule xyz*)
“Backchaining”
Typical backwards rules

\[
\frac{?P \quad ?Q}{?P \land ?Q} \quad \text{conjI}
\]

\[
\frac{?P \iff ?Q}{?P \rightarrow ?Q} \quad \text{impI} \quad \frac{\bigwedge x. ?P x}{\forall x. ?P x} \quad \text{allI}
\]

\[
\frac{?P \iff ?Q \quad ?Q \iff ?P}{?P = ?Q} \quad \text{iffI}
\]

They are known as introduction rules because they introduce a particular connective.
Teaching *blast* new intro rules

If $r$ is a theorem $[A_1; \ldots; A_n] \Rightarrow A$ then

$$(blast\ intro: r)$$

allows *blast* to backchain on $r$ during proof search.

Example:

**theorem** trans: $[?x \leq ?y; ?y \leq ?z] \Rightarrow ?x \leq ?z$

**goal** 1. $[a \leq b; b \leq c; c \leq d] \Rightarrow a \leq d$

**proof** apply($blast\ intro:\ trans$)

Can greatly increase the search space!
Forward proof: OF

If \( r \) is a theorem \([ A_1; \ldots; A_n ] \implies A\) and \( r_1, \ldots, r_m \) \((m \leq n)\) are theorems then

\[
r[\text{OF } r_1 \ldots r_m]\]

is the theorem obtained by proving \( A_1 \ldots A_m \) with \( r_1 \ldots r_m \).

Example: theorem refl: \(?t = ?t\)

\[
\text{conjI}[\text{OF refl[of "a"] refl[of "b"]}] \quad \leadsto \quad a = a \land b = b
\]
From now on:❓ mostly suppressed on slides
Single_Step_Demo.thy
is part of the Isabelle framework. It structures theorems and proof states: \[ \[ A_1; \ldots; A_n \] \implies A \]

is part of HOL and can occur inside the logical formulas \( A_i \) and \( A \).

Phrase theorems like this
\[ \[ A_1; \ldots; A_n \] \implies A \]
not like this
\[ A_1 \land \ldots \land A_n \implies A \]
4 Logic and Proof beyond “=”

- Logical Formulas
- Proof Automation
- Single Step Proofs
- Inductive Definitions
Example: even numbers

Informally:

- 0 is even
- If \( n \) is even, so is \( n + 2 \)
- These are the only even numbers

In Isabelle/HOL:

\[
\text{inductive } ev :: \text{nat } \Rightarrow \text{bool} \\
\text{where} \\
\quad ev \ 0 \\
\quad ev \ n \Rightarrow ev \ (n + 2)
\]
Easy proof:  \( ev \ 4 \)

\[

ev \ 0 \implies ev \ 2 \implies ev \ 4
\]

Trickier proof:  \( ev \ m \implies ev \ (m+m) \)

Idea: induction on the length of the proof of \( ev \ m \)

Better: induction on the structure of the proof

Two cases: \( ev \ m \) is proved by

- **rule** \( ev \ 0 \)

  \[
  \implies m = 0 \implies ev \ (0+0)
  \]

- **rule** \( ev \ n \implies ev \ (n+2) \)

  \[
  \implies m = n+2 \text{ and } ev \ (n+n) \text{ (ind. hyp.)}
  \]

  \[
  \implies m+m = (n+2)+(n+2) = ((n+n)+2)+2
  \]

  \[
  \implies ev \ (m+m)
  \]
Rule induction for \( ev \)

To prove

\[ ev \ n \quad \rightarrow \quad P \ n \]

by *rule induction* on \( ev \ n \) we must prove

- \( P \ 0 \)
- \( P \ n \quad \rightarrow \quad P(n+2) \)

Rule \( ev\text{-}\text{induct} \):

\[
\begin{array}{c}
\text{ev} \ n \\
P \ 0 \\
\land \ n. \\ P \ n \\
\Rightarrow \\ P(n+2)
\end{array}
\]

\[ P \ n \]
Format of inductive definitions

**inductive** $I :: 	au \Rightarrow bool$ **where**

\[
\begin{array}{l}
\left[ I \ a_1 ; \ldots ; I \ a_n \right] \Rightarrow I \ a
\end{array}
\]

**: **

Note:

- $I$ may have multiple arguments.
- Each rule may also contain *side conditions* not involving $I$. 
Rule induction in general

To prove

\[ I \, x \implies P \, x \]

by *rule induction* on \( I \, x \)

we must prove for every rule

\[ \left[ I \, a_1; \ldots; I \, a_n \right] \implies I \, a \]

that \( P \) is preserved:

\[ \left[ P \, a_1; \ldots; P \, a_n \right] \implies P \, a \]
Inductive_Demo.thy
1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction and Simplification

4. Logic and Proof beyond “=”

5. Isar: A Language for Structured Proofs
Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!
Apply scripts versus Isar proofs

Apply script = assembly language program
Isar proof = structured program with comments

But: apply still useful for proof exploration
A typical Isar proof

proof
  assume \( \text{formula}_0 \)
  have \( \text{formula}_1 \) by simp
  
  have \( \text{formula}_n \) by blast
  show \( \text{formula}_{n+1} \) by 

qed

proves \( \text{formula}_0 \Rightarrow \text{formula}_{n+1} \)
Isar core syntax

\[
\begin{align*}
\text{proof} & \;= \; \text{proof} \;[\text{method}] \;\text{step}^{*} \;\text{qed} \\
& \quad \mid \; \text{by} \;\text{method} \\
\text{method} & \;= \; (\text{simp} \;\ldots) \;\mid \; (\text{blast} \;\ldots) \;\mid \; (\text{induct} \;\ldots) \;\mid \;\ldots \\
\text{step} & \;= \; \text{fix} \;\text{variables} \;\mid \; (\wedge) \\
& \quad \mid \; \text{assume} \;\text{prop} \;\mid \; (\implies) \\
& \quad \mid \; [\text{from} \;\text{fact}^{+}] \; (\text{have} \;\mid \;\text{show}) \;\text{prop} \;\text{proof} \\
\text{prop} & \;= \; [\text{name:}] \;"\text{formula}" \\
\text{fact} & \;= \; \text{name} \;\mid \;\ldots
\end{align*}
\]
Isar: A Language for Structured Proofs

Isar by example

Proof patterns

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

Induction

Rule Induction
Example: Cantor’s theorem

lemma \\not\surj(f :: 'a \Rightarrow \text{set})

proof  
  default proof: assume \surj, show False
  
  assume \(a\): \surj f
  from \(a\) have \(b\): \(\forall A. \exists a. A = f a\)
    by (simp add: surj_def)
  from \(b\) have \(c\): \(\exists a. \{x. x \notin f x\} = f a\)
    by blast
  from \(c\) show False
    by blast
qed
Isar_Demo.thy

Cantor and abbreviations
Abbreviations

\[\textit{this} = \text{the previous proposition proved or assumed}\]
\[\textit{then} = \text{from this}\]
\[\textit{thus} = \text{then show}\]
\[\textit{hence} = \text{then have}\]
(have|show) prop using facts
   =
from facts (have|show) prop

  with facts
  =
from facts this
Structured lemma statement

lemma
  \textbf{fixes} \; f :: 'a \Rightarrow 'a set
  \textbf{assumes} \; s: \text{surj} \; f
  \textbf{shows} \; \text{False}

proof — \text{no automatic proof step}
  \textbf{have} \; \exists \; a. \; \{x. \; x \notin f \; x\} = f \; a \; \textbf{using} \; s
    \textbf{by} (\text{auto simp: surj_def})
  \textbf{thus} \; \text{False} \; \textbf{by} \; \text{blast}

qed

\textit{Proves} \; \text{surj} \; f \; \implies \; \text{False}

but \; \text{surj} \; f \; \text{becomes local fact} \; s \; \text{in proof.}
The essence of structured proofs

Assumptions and intermediate facts can be named and referred to explicitly and selectively
Structured lemma statements

\[
\text{fixes } x :: \tau_1 \text{ and } y :: \tau_2 \ldots \\
\text{assumes } a: P \text{ and } b: Q \ldots \\
\text{shows } R
\]

- \textbf{fixes} and \textbf{assumes} sections optional
- \textbf{shows} optional if no \textbf{fixes} and \textbf{assumes}
Isar: A Language for Structured Proofs

Isar by example

Proof patterns

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

Induction

Rule Induction
Case distinction

show \( R \)
proof cases
  assume \( P \)
  : 
  show \( R \) \ldots
next
  assume \( \neg P \)
  : 
  show \( R \) \ldots
qed

have \( P \lor Q \) \ldots
then show \( R \)
proof
  assume \( P \)
  : 
  show \( R \) \ldots
next
  assume \( Q \)
  : 
  show \( R \) \ldots
qed
show \( \neg P \)
proof
  assume \( P \)
  :  
      show \( False \)  
    qed

show \( P \)
proof \((\text{rule } \text{ccontr})\)
  assume \( \neg P \)
  :  
      show \( False \)  
    qed
show $P \iff Q$

proof

assume $P$

::

show $Q$ ... 

next

assume $Q$

::

::

show $P$ ...

qed
∀ and ∃ introduction

show \( \forall x. \ P(x) \)
proof
  fix \( x \) local fixed variable
  show \( P(x) \) . . .
qed

show \( \exists x. \ P(x) \)
proof
  : :
    show \( P(\text{witness}) \) . . .
qed
∃ elimination: obtain

have ∃ x. P(x)
then obtain x where p: P(x) by blast

: x fixed local variable

Works for one or more x
lemma \( \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set}) \)

proof

  assume \( \text{surj } f \)
  hence \( \exists a. \{x. x \notin f x\} = f a \) by (auto simp: surj_def)
  then obtain \( a \) where \( \{x. x \notin f x\} = f a \) by blast
  hence \( a \notin f a \leftrightarrow a \in f a \) by blast
  thus \( \text{False} \) by blast

qed
Set equality and subset

\begin{align*}
\text{show} & \quad A = B \\
\text{proof} & \\
\quad \text{show} & \quad A \subseteq B \quad \ldots \\
\text{next} & \\
\quad \text{show} & \quad B \subseteq A \quad \ldots \\
\text{qed} & \end{align*}

\begin{align*}
\text{show} & \quad A \subseteq B \\
\text{proof} & \\
\quad \text{fix} & \quad x \\
\quad \text{assume} & \quad x \in A \\
\quad : & \\
\quad \text{show} & \quad x \in B \quad \ldots \\
\text{qed} &
\end{align*}
Isar_Demo.thy

Exercise
Isar: A Language for Structured Proofs

Isar by example
Proof patterns

Pattern Matching and Quotations
Top down proof development

moreover and raw proof blocks

Induction
Rule Induction
Example: pattern matching

\[ \text{show } \text{formula}_1 \leftrightarrow \text{formula}_2 \quad (\text{is } ?L \leftrightarrow ?R) \]

proof

\begin{align*}
\text{assume } & ?L \\
\vdots & \\
\text{show } & ?R \ldots \\
\text{next } & \\
\text{assume } & ?R \\
\vdots & \\
\text{show } & ?L \ldots \\
\text{qed}
\end{align*}
show formula (is \(?thesis\))
proof -
  :
  
  show \(?thesis\) ...
qed

Every show implicitly defines \(?thesis\)
Introducing local abbreviations in proofs:

```latex
let \(?t\) = "some-big-term"

: 

have "\ldots ?t \ldots "
```
Quoting facts by value

By name:

\[
\text{have } x0: "x > 0" \ldots
\]
\[
\vdots
\]
\[
\text{from } x0 \ldots
\]

By value:

\[
\text{have } "x > 0" \ldots
\]
\[
\vdots
\]
\[
\text{from } 'x>0' \ldots
\]
\[
\uparrow \quad \uparrow
\]
\[\text{back quotes}\]
Isar_Demo.thy

Pattern matching and quotation
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Lemma

**assumes** $xs = rev \ xs$

**shows** $(\exists ys. \ xs = ys @ rev ys) \lor$

$(\exists ys \ a. \ xs = ys @ a \neq rev ys)$

**proof** ???
Isar_Demo.thy

Top down proof development
When automation fails

Split proof up into smaller steps.

Or explore by **apply**: 

- **have ... using ...**
- **apply** `-`
- **apply** `auto`
- **apply** `...`

At the end:

- **done**
- Better: **convert to structured proof**
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
moreover—ultimately

have $P_1$ ... 
moreover
have $P_2$ ...
moreover
: 
moreover
have $P_n$ ...
ultimately
have $P$ ...

have $lab_1$: $P_1$ ...
have $lab_2$: $P_2$ ...
: 
have $lab_n$: $P_n$ ...
from $lab_1$ $lab_2$ ...
have $P$ ...

With names
\begin{align*}
\{ & \textbf{fix } x_1 \ldots x_n \\
& \textbf{assume} \ A_1 \ldots A_m \\
& \vdots \\
& \textbf{have} \ B \\
\} \\
\end{align*}

proves \( [A_1; \ldots; A_m] \implies B \)

where all \( x_i \) have been replaced by \( ?x_i \).
Isar_Demo.thy

moreover and { }
In general: **proof method**

Applies *method* and generates subgoal(s):

\[ \land x_1 \ldots x_n \ [ A_1; \ldots ; A_m ] \implies B \]

How to prove each subgoal:

- **fix** \( x_1 \ldots x_n \)
- **assume** \( A_1 \ldots A_m \)
- : 
- **show** \( B \)

Separated by **next**
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Isar_Induct_Demo.thy

Case distinction
Datatype case distinction

datatype \( t = C_1 \vec{\tau} \mid \ldots \)

proof (cases "term")

\[
\begin{align*}
\text{case } (C_1 \ x_1 \ldots \ x_k) \\
\ldots \ x_j \ldots \\
\text{next} \\
\vdots \\
\text{qed}
\end{align*}
\]

where \( \text{case } (C_i \ x_1 \ldots \ x_k) \equiv \)

fix \( x_1 \ldots \ x_k \)

assume \( C_i: \)

\[
\text{label} \quad \text{term} = (C_i \ x_1 \ldots \ x_k) \\
\text{formula}
\]
Isar_Induct_Demo.thy

Structural induction for $nat$
Structural induction for \( \textit{nat} \)

\[
\text{show } P(n) \\
\text{proof } (\text{induct } n) \\
\text{  case } 0 \equiv \text{let } \ ?\text{case} = P(0) \\
\vdots \\
\text{  show } ?\text{case} \\
\text{next} \\
\text{  case } (\text{Suc } n) \equiv \text{fix } n \text{ assume } \text{Suc: } P(n) \\
\vdots \\
\text{  let } ?\text{case} = P(\text{Suc } n) \\
\text{  show } ?\text{case} \\
\text{qed}
\]
Structural induction with $\Rightarrow$

```
show $A(n) \Rightarrow P(n)$
proof (induct $n$)
  case 0
    :
    show $?case$
  next
  case $(Suc \ n)$
    :
    :
    :
    :
    show $?case$
qed
```

\[\begin{align*}
\text{\textbf{show} } & A(n) \Rightarrow P(n) \\
\text{\textbf{proof} (\textit{induct } n)} & \equiv \text{\textbf{fix } } x \text{\textbf{ assume } 0: } A(0) \\
& \text{\textbf{let } } ?\text{\textit{case } } = \ P(0) \\
\text{\textbf{next} } & \equiv \text{\textbf{fix } } n \\
& \text{\textbf{assume } } Suc: \ A(n) \Rightarrow P(n) \\
& \quad A(Suc \ n) \\
& \text{\textbf{let } } ?\text{\textit{case } } = \ P(Suc \ n)
\end{align*}\]
A remark on style

- **case** \((\text{Suc } n) \ldots \text{show } ?\text{case}\)
  is easy to write and maintain
- **fix** \(n\) **assume** \(\text{formula} \ldots \text{show } \text{formula}'\)
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. \(?\text{case}\)
5 Isar: A Language for Structured Proofs
Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
Induction
Rule Induction
Isar_Induct_Demo.thy

Rule induction
**Rule induction**

inductive $I :: \tau \Rightarrow \sigma \Rightarrow \text{bool}$

where

rule$_1$: . . .

. . .

rule$_n$: . . .

show $I \ x \ y \Rightarrow P \ x \ y$

proof (induct rule: $I$.induct)

  case rule$_1$

  . . .

  show $\ ?\ case$

next

  . . .

  case rule$_n$

  . . .

  show $\ ?\ case$

qed
Fixing your own variable names

\texttt{case (rule}_i \ x_1 \ldots \ x_k \texttt{)}

Renames the first $k$ variables in $rule_i$ (from left to right) to $x_1 \ldots x_k$. 
The named assumptions

Given: an inductive definition of $I$.
In a proof of

$$I \ldots \implies A_1 \implies \ldots \implies A_n \implies B,$$

in the context of

```
case R
``` we have

```
R.hyps the assumptions of rule $R$, plus the induction hypothesis for each assumption $I \ldots$
```

```
R.prems the premises $A_i$
```

$$R = R.hyps @ R.prems$$