Programming and Proving in

Fakultät für Informatik
Technische Universität München
Implication associates to the right:

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

Similarly for all other arrows: \( \implies , \rightarrow \)

\[
\frac{A_1 \ldots A_n}{B} \quad \text{means} \quad A_1 \implies \ldots \implies A_n \implies B
\]
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 $term = term$, 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 syntax:

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

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: \( \sin \pi \), \( \text{plus} \, x \, y \)

- **Function abstraction:**

  \( \lambda x. \, t \)

  is the function with parameter \( x \) and result \( t \),

  i.e. \( x \mapsto t \).

  Example: \( \lambda x. \, \text{plus} \, x \, x \)
Terms

Basic syntax:

\[ t ::= (t) \]
\[ a \quad \text{constant or variable (identifier)} \]
\[ t \; t \quad \text{function application} \]
\[ \lambda x. \; t \quad \text{function abstraction} \]
\[ \ldots \quad \text{lots of syntactic sugar} \]

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 \( \tau \)”. 

\[ \frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2} \]
Type inference

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:  \( f (x::\text{nat}) \)
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*

$$f \ a_1 \ \text{where} \ a_1 :: \tau_1$$
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[
\begin{align*}
& \text{Enclose } \textit{if} \text{ and } \textit{case} \text{ in parentheses:} \\
& f x + y \equiv (f x) + y \not\equiv f (x + y)
\end{align*}
\]
Isabelle text = Theory = Module

Syntax:    
theory $MyTh$
imports $ImpTh_1 \ldots ImpTh_n$
begin
(definitions, theorems, proofs, ...)*
end

$MyTh$: name of theory. Must live in file $MyTh\.thy$

$ImpTh_i$: name of imported theories. Import transitive.

Usually:  imports Main
Overview of Isabelle/HOL

Types and terms

Interfaces

By example: types bool, nat and list

Summary
Proof General

An Isabelle Interface

by David Aspinall
Proof General

Customized version of \((x)\)emacs:

- all of emacs
- Isabelle aware (when editing .thy files)
- mathematical symbols ("x-symbols")
  (eg \(\equiv\) instead of \(\Rightarrow\), \(\forall\) instead of \(\text{ALL}\))
Similar to ProofGeneral but

• based on jedit
• \(\Rightarrow\) easier to install
• \(\Rightarrow\) may be more familiar
• Has advantages and a few disadvantages
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
Overview of Isabelle/HOL

Types and terms

Interfaces

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

Summary
Type \textit{bool}

\textbf{datatype} \hspace{0.5cm} \textit{bool} = \textit{True} \mid \textit{False}

Predefined functions:
\&, \lor, \implies, \ldots : \textit{bool} \Rightarrow \textit{bool} \Rightarrow \textit{bool}

A logical formula is a term of type \textit{bool}

\textbf{if-and-only-if:} \equiv
Type \textit{nat}

datatype \textit{nat} = 0 | Suc \textit{nat}

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

Predefined functions: +, *, ... :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}

! Numbers and arithmetic operations are overloaded:

0,1,2,... :: 'a, + :: 'a \Rightarrow 'a \Rightarrow 'a

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

unless the context is unambiguous: Suc z
Nat_Demo.thy
Lemma $\text{add } m \ 0 = m$

Proof by induction on $m$.

- Case $0$ (the base case):
  $\text{add } 0 \ 0 = 0$ holds by definition of $\text{add}$.

- Case $\text{Suc } m$ (the induction step):
  We assume $\text{add } m \ 0 = m$ (induction hypothesis IH) and we need to show $\text{add } (\text{Suc } m) \ 0 = \text{Suc } m$.

  The proof is as follows:
  
  \[
  \text{add } (\text{Suc } m) \ 0 = \text{Suc } (\text{add } m \ 0) \quad \text{by def. of } \text{add}
  \]
  
  \[
  = \text{Suc } m \quad \text{by IH}
  \]
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")
- `[x₁, ..., xₙ] = `x₁ # ... `xₙ # `[]
Structural Induction for lists

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

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

\[
P([]) \land \forall x \; xs. \; P(xs) \implies P(x\#xs)
\]
List_Demo.thy
An informal proof

Lemma \( \text{app} \ (\text{app}\ xs\ ys)\ zs = \text{app} \ xs\ (\text{app}\ ys\ zs) \)

Proof by induction on \( xs \).

- **Case Nil**: \( \text{app} (\text{app} Nil\ ys)\ zs = \text{app}\ ys\ zs = \text{app} Nil\ (\text{app}\ ys\ zs) \) holds by definition of \( \text{app} \).
- **Case Cons \( x \) \( xs \)**: We assume \( \text{app} \ (\text{app}\ xs\ ys)\ zs = \text{app}\ xs\ (\text{app}\ ys\ zs) \) (IH), and we need to show \( \text{app} \ (\text{app} \ (\text{Cons}\ x\ xs)\ ys)\ zs = \text{app} \ (\text{Cons}\ x\ xs)\ (\text{app}\ ys\ zs) \).

The proof is as follows:

\[
\begin{align*}
\text{app} \ (\text{app} \ (\text{Cons}\ x\ xs)\ ys)\ zs &= \text{Cons} \ x\ (\text{app} \ (\text{app}\ xs\ ys)\ zs) \text{ by definition of } \text{app} \\
&= \text{Cons} \ x\ (\text{app}\ xs\ (\text{app}\ ys\ zs)) \text{ by IH} \\
&= \text{app} \ (\text{Cons}\ x\ xs)\ (\text{app}\ ys\ zs) \text{ by definition of } \text{app}
\end{align*}
\]
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!

Predefined: \( xs @ ys \) (append), \( \text{length} \), and \( \text{map} \):

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

\[
\text{fun } \text{map} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list } \text{ where}
\]

\[
\text{map } f \ [] = [] \quad | \\
\text{map } f \ (x#xs) = f \ x \ # \ \text{map } f \ xs
\]

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

Types and terms
Interfaces
By example: types \textit{bool, 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{induction} 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):

  \textasciitilde “\textasciitilde” is used only from left to right!
Proofs

General schema:

```plaintext
lemma name: "..."
apply (...)
apply (...)
:
done
```

If the lemma is suitable as a simplification rule:

```plaintext
lemma name[simp]: "..."
```
Top down proofs

Command

\texttt{sorry}

“completes” any proof.

Allows top down development:

\textit{Assume lemma first, prove it later.}
1. $\bigwedge x_1 \ldots x_p. \ A \implies B$

$x_1 \ldots x_p$ fixed local variables
$A$ local assumption(s)
$B$ actual (sub)goal
Preview: Multiple assumptions

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

abbreviates

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

; \approx \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

\texttt{type\_synonym} name = $\tau$

Introduces a \textit{synonym} name for type $\tau$

Examples:

\texttt{type\_synonym} string = char list
\texttt{type\_synonym} (\texttt{'}a,\texttt{'}b)\texttt{foo} = \texttt{'}a list $\times$ \texttt{'}b list

Type synonyms are expanded after parsing and are not present in internal representation and output.
**datatype** — the general case

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

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

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

Datatype values can be taken apart with *case*:

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

Wildcards: `_`

\[
\text{(case } m \text{ of } 0 \Rightarrow \text{Suc } 0 \mid \text{Suc } _{\ldots} \Rightarrow 0)\
\]

Nested patterns:

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

Complicated patterns mean complicated proofs!

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

Type definitions

Function definitions
Non-recursive definitions

Example:

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

No pattern matching, just \( f \ x_1 \ \ldots \ x_n \ = \ \ldots \)
The danger of nontermination

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

```plaintext
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

\textbf{fun} \texttt{ack} :: \texttt{nat} \Rightarrow \texttt{nat} \Rightarrow \texttt{nat} \textbf{where}

\begin{align*}
\texttt{ack} \ 0 \ & \ n \quad = \ \texttt{Suc} \ n \\
\texttt{ack} \ (\texttt{Suc} \ m) \ 0 \ & \ = \ \texttt{ack} \ m \ (\texttt{Suc} \ 0) \\
\texttt{ack} \ (\texttt{Suc} \ m) \ (\texttt{Suc} \ n) \ & \ = \ \texttt{ack} \ m \ (\texttt{ack} \ (\texttt{Suc} \ m) \ n)
\end{align*}

Terminates because the arguments decrease \textit{lexicographically} with each recursive call:

\begin{itemize}
  \item \((\texttt{Suc} \ m, \ 0) > (m, \ \texttt{Suc} \ 0)\)
  \item \((\texttt{Suc} \ m, \ \texttt{Suc} \ n) > (\texttt{Suc} \ m, \ n)\)
  \item \((\texttt{Suc} \ m, \ \texttt{Suc} \ n) > (m, \ \_)\)
\end{itemize}
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:

\[
\text{fun } \textit{rev} :: \ 'a \ list \Rightarrow \ 'a \ list \ \text{where} \\
\textit{rev} \ [] = [] | \\
\textit{rev} \ (x \# xs) = \textit{rev} \ xs @ [x]
\]

A tail recursive version:

\[
\text{fun } \textit{itrev} :: \ 'a \ list \Rightarrow \ 'a \ list \Rightarrow \ 'a \ list \ \text{where} \\
\textit{itrev} \ [] \ ys = ys | \\
\textit{itrev} \ (x \# xs) \ ys = \\
\text{lemma } \textit{itrev} \ xs \ [] = \textit{rev} \ xs
\]
Induction_Demo.thy

Generalization
Generalization

- Replace constants by variables
- Generalize free variables
  - by \textit{arbitrary} in induction proof
  - (or by universal quantifier in formula)
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.
Computing Induction: Example

\[
\textbf{fun} \; \text{div2} :: \; \texttt{nat} \Rightarrow \texttt{nat} \; \textbf{where}
\]
\[
div2 \; 0 = 0 \; | \\
div2 \; (\text{Suc} \; 0) = 0 \; | \\
div2 \; (\text{Suc}(\text{Suc} \; n)) = \text{Suc}(\text{div2} \; n)
\]

\[\leadsto \; \text{induction rule div2.induct:}\]
\[
P(0) \quad P(\text{Suc} \; 0) \quad \land n. \; P(n) \implies P(\text{Suc}(\text{Suc} \; n)) \\
P(m)
\]
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 defining equation

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

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

Induction follows course of (terminating!) computation

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

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

$$(induction \ a_1 \ldots \ a_n \ rule: \ f.induct)$$

Heuristic:

- there should be a call $f \ a_1 \ldots \ a_n$ in your goal
- ideally the $a_i$ should be variables.
Induction_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 = (Term) Rewriting
An example

Equations:

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

\[ (\text{Suc } m) + n = \text{Suc } (m + n) \]  \hspace{1cm} (2)

\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \]  \hspace{1cm} (3)

\[ (0 \leq m) = \text{True} \]  \hspace{1cm} (4)

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{1cm} (1) \overset{=} \rightarrow

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{1cm} (2) \overset{=} \rightarrow

\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \]  \hspace{1cm} (3) \overset{=} \rightarrow

\[ 0 \leq 0 + x \]  \hspace{1cm} (4) \overset{=} \rightarrow

\[ \text{True} \]
Conditional rewriting

Simplification rules can be conditional:

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

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

Example:

\[
p(0) = True
\]

\[
p(x) \implies f(x) = g(x)
\]

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 \textit{simp}-rules (almost) blindly from left to right.

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

\[
\left[ P_1; \ldots; P_k \right] \implies l = r
\]
is suitable as a \textit{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 \textit{simp}

Goal: \begin{align*} & 1. \left[ P_1; \ldots; P_m \right] \implies C \end{align*}

\textbf{apply} \((simp\ add: eq_1 \ldots eq_n)\)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute \textit{simp}
- rules from \textit{fun} and \textit{datatype}
- additional lemmas \(eq_1 \ldots eq_n\)
- assumptions \(P_1 \ldots P_m\)

Variations:

- \((simp \ldots del: \ldots)\) removes \textit{simp}-lemmas
- \textit{add} and \textit{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: ...)

67
Rewriting with definitions

Definitions (definition) must be used explicitly:

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

\(f\) 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
Overview of Isabelle/HOL

Type and function definitions

Induction and Simplification

Logic and Proof beyond “=”

Isar: A Language for Structured Proofs
4 Logic and Proof beyond “=”

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

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

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: \(\leftarrow\longrightarrow\) (same precedence as \(\rightarrow\))
Warning

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

\[
! P \land \forall x. Q x \simarrow P \land (\forall x. Q x) !
\]
X-Symbols

... and their ascii representations:

\(
\forall \quad \text{\textbackslash{forall}} \quad \text{ALL}
\)

\(
\exists \quad \text{\textbackslash{exists}} \quad \text{EX}
\)

\(
\lambda \quad \text{\textbackslash{lambda}} \quad \%
\)

\(
\rightarrow \quad \rightarrow
\)

\(
\leftrightarrow \quad \leftrightarrow
\)

\(
\wedge \quad \\wedge \quad \&
\)

\(

\vee \quad \\vee \quad |
\)

\(
\neg \quad \text{\textbackslash{not}} \quad ~
\)

\(
\neq \quad \text{\textbackslash{noteq}} \quad ~=\)
Sets over type 'a

'a set

- \{\}, \{e_1, \ldots, e_n\}
- e \in A, \ A \subseteq B
- A \cup B, \ A \cap B, \ A - B, \ \neg A
- \ldots

\in \ \\textlangle in\rangle
\subseteq \ \textlangle subseteq\rangle \ \leq
\cup \ \textlangle union\rangle \ \text{Un}
\cap \ \textlangle inter\rangle \ \text{Int}
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
fastforce

- 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

Characteristics:

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

Do you feel lucky?

\[1\text{Automatic Theorem Provers}\]
by($proof\text{-}method$) 

\approx 

apply($proof\text{-}method$) 
done
Auto_Proof_Demo.thy
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 \( \texttt{?V} \).

Example: theorem \texttt{conjI}: \[ \texttt{[?P; ?Q] \implies ?P \land ?Q} \]

These ?-variables can later be instantiated:

- By hand:
  \[
  \texttt{conjI[of "a=b" "False"] \implies [a = b; False] \implies a = b \land False}
  \]

- By unification:
  
  unifying \( ?P \land ?Q \) with \( a=b \land False \)

  sets \( ?P \) to \( a=b \) and \( ?Q \) to \( False \).
Rule application

Example: rule: \([ ?P; ?Q] \implies ?P \land ?Q\)

subgoal: 1. \(\ldots \implies A \land B\)

Result: 1. \(\ldots \implies A\)

2. \(\ldots \implies B\)

The general case: applying rule \([ A_1; \ldots ; A_n ] \implies A\) to subgoal \(\ldots \implies C\):

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

\textbf{apply(} rule \textit{xyz} \textbf{)}

“Backchaining”
Typical backwards rules

\[
\frac{\Phi \quad \Theta}{\Phi \land \Theta} \text{ conjI}
\]

\[
\frac{\Phi \iff \Theta}{\Phi \implies \Theta} \text{ impI} \quad \frac{\land x. \Phi x}{\forall x. \Phi x} \text{ allI}
\]

\[
\frac{\Phi \iff \Theta \quad \Theta \iff \Phi}{\Phi = \Theta} \text{ iffI}
\]

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

If \( r \) is a theorem \([ A_1; \ldots; A_n ] \implies A\) then

\[ (\text{blast intro: } r) \]

allows `blast` to backchain on \( r \) during proof search.

Example:

**Theorem** `trans`: \([ \ ?x \leq \ ?y; \ ?y \leq \ ?z \ ] \implies \ ?x \leq \ ?z``

**Goal** 1. \([ \ a \leq b; \ b \leq c; \ c \leq d \ ] \implies \ a \leq d``

**Proof** `apply(blast intro: trans)`

Can greatly increase the search space!
Forward proof: OF

If $r$ is a theorem $\left[ A_1; \ldots; A_n \right] \Longrightarrow A$
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: $\ ?t = \ ?t$

$$\\text{conjI}[\text{OF refl[of "a"] refl[of "b"]}]$$

\[ \Rightarrow \]

$$a = a \land b = b$$
From now on: ? mostly suppressed on slides
Single_Step_Demo.thy
⇒ versus →

⇒ is part of the Isabelle framework. It structures theorems and proof states: \[ [ A_1; \ldots; A_n ] \Rightarrow A \]

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

Phrase theorems like this \[ [ A_1; \ldots; A_n ] \Rightarrow A \]
not like this \( A_1 \land \ldots \land A_n \Rightarrow A \)
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:

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

\[ ev \ 0 \implies ev \ 2 \implies ev \ 4 \]
Consider

**fun** even :: nat ⇒ bool **where**

even 0 = True |
even (Suc 0) = False |
even (Suc (Suc n)) = even n

A trickier proof: \( ev \ m \implies even \ m \)

By induction on the *structure* of the derivation of \( ev \ m \)

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

- **rule** \( ev \ 0 \)
  \[ \implies m = 0 \implies even \ m = True \]

- **rule** \( ev \ n \implies ev \ (n+2) \)
  \[ \implies m = n+2 \text{ and } even \ n \text{ (IH)} \]
  \[ \implies even \ m = even \ (n+2) = even \ n = True \]
Rule induction for $ev$

To prove

$$ev\ n \iff P\ n$$

by rule induction on $ev\ n$ we must prove

- $P\ 0$
- $P\ n \iff P(n+2)$

Rule $ev\ .\ induct$:

$$
\begin{align*}
& ev\ n & P\ 0 & \land n.\ [ ev\ n;\ P\ n ] \iff P(n+2) \\
& P\ n \quad \quad & & \quad \\
\end{align*}
$$
Format of inductive definitions

\textbf{inductive} \ I :: \ \tau \ \Rightarrow \ bool \ \textbf{where}

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

\vdots

\textbf{Note:}

- \( I \) may have multiple arguments.
- Each rule may also contain \textit{side conditions} not involving \( I \).
Rule induction in general

To prove

\[ I \ x \Rightarrow P \ x \]

by rule induction on \( I \ x \)
we must prove for every rule

\[ [ I \ a_1; \ldots ; I \ a_n ] \Rightarrow I \ a \]

that \( P \) is preserved:

\[ [ I \ a_1; P \ a_1; \ldots ; I \ a_n; P \ a_n ] \Rightarrow P \ a \]
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

\[\text{assume } \text{formula}_0\]
\[\text{have } \text{formula}_1 \text{ by simp} \]
\[\vdots\]
\[\text{have } \text{formula}_n \text{ by blast} \]
\[\text{show } \text{formula}_{n+1} \text{ by } \ldots\]

qed

proves \[\text{formula}_0 \implies \text{formula}_{n+1}\]
Isar core syntax

\[
\text{proof} = \text{proof [method]} \ \text{step}^* \ \text{qed} \\
\quad \mid \ \text{by method}
\]

\[
\text{method} = (\text{simp} \ldots) \mid (\text{blast} \ldots) \mid (\text{induction} \ldots) \mid \ldots
\]

\[
\text{step} = \text{fix variables} \quad (\land) \\
\quad \mid \ \text{assume prop} \quad (\iff) \\
\quad \mid \ [\text{from fact}^+] \ (\text{have} | \text{show}) \ \text{prop} \ \text{proof}
\]

\[
\text{prop} = [\text{name:}] "\text{formula}"
\]

\[
\text{fact} = \text{name} | \ldots
\]
Isar: A Language for Structured Proofs

Isar by example

Proof patterns
Pattern Matching and Quotations
Top down proof development
Induction
Rule Induction
Rule Inversion
Example: Cantor’s theorem

lemma \( \neg \text{surj}(f :: \ 'a \Rightarrow \ 'a \ \text{set}) \)

proof  
  default proof: assume \( \text{surj} \), show \( \text{False} \)
  
  assume \( a: \ \text{surj} \ f \)

  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 \( \text{False} \)
  
  by blast

qed
Isar_Demo.thy

Cantor and abbreviations
Abbreviations

\[ this = \text{the previous proposition proved or assumed} \]
\[ then = \text{from this} \]
\[ thus = \text{then show} \]
\[ hence = \text{then have} \]
using and with

(\text{have}|\text{show}) \text{ prop using facts} = \text{from facts (have}|\text{show}) \text{ prop}

\text{with facts} = \text{from facts this}
Structured lemma statement

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

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

qed

Proves \( \text{ surj } f \implies False \)

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

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

- **fixes** $x :: \tau_1$ and $y :: \tau_2$ ...
- **assumes** $a: P$ and $b: Q$ ...
- **shows** $R$

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**
Isar: A Language for Structured Proofs

Isar by example

Proof patterns

Pattern Matching and Quotations

Top down proof development

Induction

Rule Induction

Rule Inversion
Case distinction

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

have \( P \lor Q \) 
then show \( R \)
proof
  assume \( P \)
  :
  show \( R \) 
next
  assume \( Q \)
  :
  show \( R \) 
qed
Contradiction

\[
\begin{align*}
\text{show } & \neg P \\
\text{proof} & \\
\quad \text{assume } P & \\
\quad : & \\
\quad \text{show } & \text{False} \ldots \\
\text{qed} & \\
\end{align*}
\]

\[
\begin{align*}
\text{show } & P \\
\text{proof} & (\text{rule } \texttt{ccontr}) \\
\quad \text{assume } & \neg P \\
\quad : & \\
\quad \text{show } & \text{False} \ldots \\
\text{qed} & \\
\end{align*}
\]
show $P \leftrightarrow Q$

proof

assume $P$

::

show $Q$ . . .

next

assume $Q$

::

show $P$ . . .

qed
∀ and ∃ introduction

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

show ∃ x. P(x)
proof
  :  
    show P(witness)  ...
qed
∃ elimination: obtain

have $\exists x. \ P(x)$
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 \((\text{auto simp: surj_def})\)

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
show $A = B$
proof
  show $A \subseteq B$ . . .
next
  show $B \subseteq A$ . . .
qed

show $A \subseteq B$
proof
  fix $x$
  assume $x \in A$
  :
  show $x \in B$ . . .
qed
Isar_Demo.thy

Exercise
Isar: A Language for Structured Proofs

Isar by example
Proof patterns

Pattern Matching and Quotations
Top down proof development
Induction
Rule Induction
Rule Inversion
Example: pattern matching

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

\text{proof}

\begin{align*}
\text{assume } \ ?L \\
\vdots \\
\text{show } \ ?R \\
\vdots \\
\text{show } \ ?L \\
\end{align*}

\text{next}

\begin{align*}
\text{assume } \ ?R \\
\vdots \\
\text{show } \ ?L \\
\vdots \\
\text{show } \ ?L \\
\end{align*}

\text{qed}
show \( \text{formula} \ (\text{is} \ \text{thesis}) \)

proof -

\begin{align*}
&::

&\quad \text{show} \ \text{thesis} \ \ldots
\end{align*}

qed

Every show implicitly defines \( \text{thesis} \)
Quoting facts by value

By name:

\[
\text{have } x_0: \ "x > 0" \ldots \\
\vdots \\
\text{from } x_0 \ldots
\]

By value:

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

↑ ↑
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
Induction
Rule Induction
Rule Inversion
Example

lemma
assumes \( xs = \text{rev} \; xs \)
shows \((\exists \; ys. \; xs = ys \; @ \; \text{rev} \; ys) \lor (\exists \; ys \; a. \; xs = ys \; @ \; a \neq \text{rev} \; ys)\)
proof ???
Isar_Demo.thy

Top down proof development
In general: **proof** method

Applies *method* and generates subgoal(s):

\[ \forall x_1 \ldots x_n \left[ A_1; \ldots ; A_m \right] \Rightarrow B \]

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

Induction
Rule Induction
Rule Inversion
Isar_Induction_Demo.thy

Case distinction
**Datatype case distinction**

\[
datatype \quad t = \ C_1 \overrightarrow{\tau} \ | \ \ldots
\]

**proof** \((cases "term")\)

**case** \((C_1 \ x_1 \ldots \ x_k)\)

\[
\ldots \ x_j \ldots
\]

**next**

: 

**qed**

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

\(\text{fix } x_1 \ldots x_k\)

\(\text{assume } C_i: \quad \underbrace{\text{label}}_{\text{term}} = (C_i \ x_1 \ldots x_k)\)

\(\underbrace{\text{formula}}_{\text{term}}\)
Isar_Induction_Demo.thy

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

\[
\begin{align*}
\text{show } & \ P(n) \\
\text{proof } & \ (\text{induction } n) \\
\quad \text{case } & \ 0 \\
\vdots & \\
\quad \text{show } & \ ?\text{case} \\
\end{align*}
\]

\[
\begin{align*}
\equiv & \quad \text{let } ?\text{case} = P(0) \\
\text{next } & \\
\quad \text{case } & \ (\text{Suc } n) \\
\vdots & \\
\vdots & \\
\quad \text{show } & \ ?\text{case} \\
\end{align*}
\]

\[
\begin{align*}
\equiv & \quad \text{fix } n \text{ assume } \text{Suc: } P(n) \\
\quad \text{let } & \ ?\text{case} = P(\text{Suc } n) \\
\quad \text{show } & \ ?\text{case} \\
\text{qed} & 
\end{align*}
\]
Structural induction with \( \implies \)

\[
\text{show } A(n) \implies P(n)
\]

\textbf{proof} \quad (\text{induction } n)

\begin{align*}
\text{case } 0 & \equiv \text{assume } 0: A(0) \\
\text{let } ?\text{case} = P(0) \\
\text{next} & \equiv \text{fix } n \\
\text{assume } \text{Suc}: & \quad A(n) \implies P(n) \\
& \quad A(\text{Suc } n) \\
\text{let } ?\text{case} = P(\text{Suc } n)
\end{align*}

\text{show } ?\text{case}

\text{qed}
Named assumptions

In a proof of

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

by structural induction:
In the context of

**case** \( C \)

we have

- **C.IH** the induction hypotheses
- **C.prems** the premises \( A_i \)

\[ C \quad C.IH + C.prems \]
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
Induction
Rule Induction
Rule Inversion
Isar_Induction_Demo.thy

Rule induction
Rule induction

inductive $I : \tau \Rightarrow \sigma \Rightarrow \text{bool}$
where
rule_1: $\ldots$
\vdots
rule_n: $\ldots$

\[
\begin{array}{c}
\text{show } I \; x \; y \Rightarrow P \; x \; y \\
\text{proof } (\text{induction rule: } I.\text{induct}) \\
\quad \text{case } rule_1 \\
\quad \ldots \\
\quad \text{show } ?\text{case} \\
\text{next} \\
\quad \vdots \\
\text{next} \\
\quad \text{case } rule_n \\
\quad \ldots \\
\quad \text{show } ?\text{case} \\
\text{qed}
\end{array}
\]
Fixing your own variable names

\[
\text{case } (\text{rule}_i \ x_1 \ldots \ x_k)
\]

Renames the first \( k \) variables in \( \text{rule}_i \) (from left to right) to \( x_1 \ldots x_k \).
Named assumptions

In a proof of

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

by rule induction on \( I \ldots \): 

In the context of 

\textbf{case} \( R \)

we have 

\( R.IH \) the induction hypotheses 
\( R.hyps \) the assumptions of rule \( R \) 
\( R.prems \) the premises \( A_i \) 

\( R \quad R.IH + R.hyps + R.prems \)
Isar: A Language for Structured Proofs

Isar by example
Proof patterns
Pattern Matching and Quotations
Top down proof development
Induction
Rule Induction
Rule Inversion
Rule inversion

**inductive** $ev :: nat \Rightarrow bool$ where

$ev0$: $ev \ 0$

$evSS$: $ev \ n \Rightarrow ev(Suc(Suc \ n))$

What can we deduce from $ev \ n$?
That it was proved by either $ev0$ or $evSS$!

$ev \ n \Rightarrow n = 0 \lor (\exists \ k. \ n = Suc \ (Suc \ k) \land ev \ k)$

*Rule inversion = case distinction over rules*
Isar_Induction_Demo.thy

Rule inversion
Rule inversion template

from 'ev n' have P
proof cases
  case ev0
  : show ?thesis  ...
next
  case (evSS k)
  : show ?thesis  ...
qed

Impossible cases disappear automatically