Programming and Proving in

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

Notation

Implication associates to the right:

$$A \Longrightarrow B \Longrightarrow C \quad \text{means} \quad A \Longrightarrow (B \Longrightarrow C)$$

Similarly for all other arrows: $\Rightarrow$, $\longrightarrow$

$$\frac{A_1 \quad \ldots \quad A_n}{B} \quad \text{means} \quad A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

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: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\ldots$

Types

Basic syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & bool \mid nat \mid int \mid \ldots & \text{base types} \\
& | & 'a \mid 'b \mid \ldots & \text{type variables} \\
& | & \tau \Rightarrow \tau & \text{functions} \\
& | & \tau \times \tau & \text{pairs (ascii: } *) \\
& | & \tau\ list & \text{lists} \\
& | & \tau\ set & \text{sets} \\
& | & \ldots & \text{user-defined types}
\end{array}
$$

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: $\lambda x.\ plus\ x\ x$

Terms

Basic syntax:

$$
\begin{array}{rcll}
t & ::= & (t) & \\
 & \mid & a & \text{constant or variable (identifier)} \\
 & \mid & t\ t & \text{function application} \\
 & \mid & \lambda x.\ t & \text{function abstraction} \\
 & \mid & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

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 λ-calculus.

The computation rule of the λ-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 β-reduction.
- Isabelle performs β-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 τ".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}$$

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: $f\ (x{::}nat)$

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*

$f\ a_1$ where $a_1 :: \tau_1$

Predefined syntactic sugar

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

Prefix binds more strongly than infix:

$$! \quad f\ x + y \ \equiv\ (f\ x) + y \ \not\equiv\ f\ (x + y) \quad !$$

Enclose *if* and *case* in parentheses:

$$! \quad (\mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \_) \quad !$$

Isabelle text = Theory = Module

Syntax:

```
theory MyTh
imports ImpTh_1 ... 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`

1. 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 $\Longrightarrow$ instead of `==>`, $\forall$ instead of `ALL`)

isabelle jedit

Similar to ProofGeneral but

- based on jedit
- $\Longrightarrow$ easier to install
- $\Longrightarrow$ 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

1. Overview of Isabelle/HOL
   - Types and terms
   - Interfaces
   - By example: types $bool$, nat and $list$
   - Summary

Type $bool$

datatype $bool \; = \; True \; | \; False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots \; :: bool \Rightarrow bool \Rightarrow bool$

A logical formula is a term of type $bool$

if-and-only-if: $=$

Type nat

datatype $nat \ = \ 0 \ | \ Suc \ nat$

Values of type nat: 0, $Suc\ 0$, $Suc(Suc\ 0)$, ...

Predefined functions: $+$, $*$, ... $::\ nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,\ldots :: {'a}, \quad + :: \ {'a} \Rightarrow {'a} \Rightarrow {'a}$

You need type annotations: $1 :: nat$, $x + (y{::}nat)$
unless the context is unambiguous: $Suc\ z$

Nat_Demo.thy

An informal proof

Lemma $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$ (induction hypothesis IH)
  and we need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:

$$\begin{array}{lcll} add\ (Suc\ m)\ 0 & = & Suc\ (add\ m\ 0) & \text{by def. of } add \\ & = & Suc\ m & \text{by IH} \end{array}$$

Type $'a\ list$

Lists of elements of type $'a$

datatype $'a\ list\ =\ Nil\ \mid\ Cons\ 'a\ \ ('a\ list)$

Syntactic sugar:

- $[] = Nil$: empty list
- $x \# xs = Cons\ x\ xs$:
  list with first element x (*"head"*) and rest xs (*"tail"*)
- $[x_1, \ldots, x_n] = x_1 \# \ldots\ x_n \# []$

Structural Induction for lists

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

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

$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x\#xs)}{P(xs)}$$

List_Demo.thy

An informal proof

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$

Proof by induction on xs.

- Case Nil: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of app.
- Case $Cons\ x\ xs$: We assume $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$ (IH), and we need to show
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs =$
  $app\ (Cons\ x\ xs)\ (app\ ys\ zs)$.
  The proof is as follows:
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
  $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of app
  $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
  $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of app

Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

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

$$map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

fun $map :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$map\ f\ [] = []$ |
$map\ f\ (x\#xs) = f\ x\ \#\ map\ f\ xs$

Note: map takes *function* as argument.

1. Overview of Isabelle/HOL

   Types and terms

   Interfaces

   By example: types $bool$, nat and $list$

   Summary

- **datatype** defines (possibly) recursive data types.
- **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.

Proof methods

- *induction* performs structural induction on some variable (if the type of the variable is a datatype).

- *auto* solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

  "$=$" is used only from left to right!

Proofs

General schema:

lemma *name*: `"..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]: "..."`

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

The proof state

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

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

Preview: Multiple assumptions

$$\llbracket\, A_1; \ldots\ ; A_n \,\rrbracket \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;\quad \approx \quad$ "and"

2 Type and function definitions

Type definitions

Function definitions

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples:

type_synonym $string = char\ list$

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

datatype — the general case

$$\textbf{datatype}\ (\alpha_1,\dots,\alpha_n)\tau \quad = \quad C_1\ \tau_{1,1}\dots\tau_{1,n_1} \quad | \quad \dots \quad | \quad C_k\ \tau_{k,1}\dots\tau_{k,n_k}$$

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

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Datatype values can be taken apart with $case$:

$$(case\ xs\ of\ \ [] \Rightarrow \ldots \ \ |\ \ y\#ys \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

Wildcards: _

$$(case\ m\ of\ \ 0 \Rightarrow Suc\ 0\ \ |\ \ Suc\ \_ \Rightarrow 0)$$

Nested patterns:

$$(case\ xs\ of\ \ [0] \Rightarrow 0\ \ |\ \ [Suc\ n] \Rightarrow n\ \ |\ \ \_ \Rightarrow 2)$$

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

2 Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example:

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

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

$$
\begin{array}{l}
\textbf{fun } sep :: {}'a \Rightarrow {}'a\ list \Rightarrow {}'a\ list \textbf{ where} \\
sep\ a\ (x\#y\#zs) = x\ \#\ a\ \#\ sep\ a\ (y\#zs) \quad | \\
sep\ a\ xs = xs
\end{array}
$$

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$
\begin{array}{lll}
ack\ 0 & n & = Suc\ n \quad | \\
ack\ (Suc\ m)\ 0 & & = ack\ m\ (Suc\ 0) \quad | \\
ack\ (Suc\ m)\ (Suc\ n) & & = ack\ m\ (ack\ (Suc\ m)\ n)
\end{array}
$$

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 $rev :: 'a\ list \Rightarrow 'a\ list$ **where**
$rev\ []\ =\ []\ \mid$
$rev\ (x\#xs)\ =\ rev\ xs\ @\ [x]$

A tail recursive version:

fun $itrev :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$itrev\ []\ \ ys = ys\ \mid$
$itrev\ (x\#xs)\ \ ys =$

lemma $itrev\ xs\ [] = rev\ xs$

Induction_Demo.thy

Generalization

Generalization

- Replace constants by variables
- Generalize free variables
    - by $arbitrary$ 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.

Computation Induction: Example

fun $div2 :: nat \Rightarrow nat$ **where**
$div2\ 0 = 0\ \ |$
$div2\ (Suc\ 0) = 0\ \ |$
$div2\ (Suc(Suc\ n)) = Suc(div2\ n)$

$\leadsto$ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad \bigwedge n.\ \ P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

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 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 $f.induct$

How to apply $f.induct$

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

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

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

3 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:

$$
\begin{aligned}
0 + n &= n & (1)\\
(Suc\ m) + n &= Suc\ (m + n) & (2)\\
(Suc\ m \le Suc\ n) &= (m \le n) & (3)\\
(0 \le m) &= True & (4)
\end{aligned}
$$

Rewriting:

$$
\begin{aligned}
0 + Suc\ 0 \ &\le\ Suc\ 0 + x & \stackrel{(1)}{=}\\
Suc\ 0 \ &\le\ Suc\ 0 + x & \stackrel{(2)}{=}\\
Suc\ 0 \ &\le\ Suc\ (0 + x) & \stackrel{(3)}{=}\\
0 \ &\le\ 0 + x & \stackrel{(4)}{=}\\
&True
\end{aligned}
$$

Conditional rewriting

Simplification rules can be conditional:

$$[\![\; P_1; \; \ldots ; \; P_k \;]\!] \Longrightarrow l = r$$

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

Example:

$$\begin{array}{rcl} p(0) & = & True \\ p(x) \Longrightarrow \; f(x) & = & g(x) \end{array}$$

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

$$\llbracket\ P_1;\ \ldots;\ P_k\ \rrbracket \Longrightarrow l = r$$

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

$$n < m \Longrightarrow (n < Suc\ m) = True \quad \text{YES}$$
$$Suc\ n < m \Longrightarrow (n < m) = True \quad \text{NO}$$

Proof method $simp$

Goal: 1. $\llbracket P_1; \ldots; P_m \rrbracket \Longrightarrow C$

apply$(simp\ add\!: eq_1 \ \ldots \ eq_n)$

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

- lemmas with attribute $simp$
- rules from **fun** and **datatype**
- additional lemmas $eq_1 \ \ldots \ eq_n$
- assumptions $P_1 \ \ldots \ P_m$

Variations:

- $(simp \ \ldots \ del\!: \ldots)$ removes $simp$-lemmas
- add and 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:
  $(auto\ simp\ add\colon \ldots\ simp\ del\colon \ldots)$

Rewriting with definitions

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

$$(simp\ add\colon f\_def \dots)$$

f is the function whose definition is to be unfolded.

Case splitting with $simp$

Automatic:

$$P(\mathit{if}\ A\ \mathit{then}\ s\ \mathit{else}\ t) \\ = \\ (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

By hand:

$$P(\mathit{case}\ e\ \mathit{of}\ 0 \Rightarrow a \mid \mathit{Suc}\ n \Rightarrow b) \\ = \\ (e = 0 \longrightarrow P(a)) \wedge (\forall\, n.\ e = \mathit{Suc}\ n \longrightarrow P(b))$$

Proof method: $(\mathit{simp}\ \mathit{split}{:}\ \mathit{nat.split})$
Or auto. Similar for any datatype t: $\mathit{t.split}$

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):

$$
\begin{array}{lclclcl}
\mathit{form} & ::= & (\mathit{form}) & \mid & \mathit{term} = \mathit{term} & \mid & \neg \mathit{form} \\
& \mid & \mathit{form} \wedge \mathit{form} & \mid & \mathit{form} \vee \mathit{form} & \mid & \mathit{form} \longrightarrow \mathit{form} \\
& \mid & \forall x.\ \mathit{form} & \mid & \exists x.\ \mathit{form} & &
\end{array}
$$

Examples:

$$
\begin{array}{rcl}
\neg\ A \wedge B \vee\ C & \equiv & ((\neg\ A) \wedge\ B) \vee\ C \\
s = t \wedge C & \equiv & (s = t) \wedge\ C \\
A \wedge\ B = B \wedge\ A & \equiv & A \wedge\ (B = B) \wedge\ A \\
\forall\, x.\ P\ x \wedge\ Q\ x & \equiv & \forall\, x.\ (P\ x \wedge\ Q\ x)
\end{array}
$$

Input syntax: $\longleftrightarrow$ (same precedence as $\longrightarrow$)

Warning

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

$$\textbf{!} \quad P \wedge \forall\, x.\; Q\; x \;\rightsquigarrow\; P \wedge (\forall\, x.\; Q\; x) \quad \textbf{!}$$

X-Symbols

. . . and their ascii representations:

<table>
<tr><td>$\forall$</td><td>\<forall></td><td>ALL</td></tr>
<tr><td>$\exists$</td><td>\<exists></td><td>EX</td></tr>
<tr><td>λ</td><td>\<lambda></td><td>%</td></tr>
<tr><td>$\longrightarrow$</td><td>--></td><td></td></tr>
<tr><td>$\longleftrightarrow$</td><td><-></td><td></td></tr>
<tr><td>$\wedge$</td><td>/\</td><td>&</td></tr>
<tr><td>$\vee$</td><td>\/</td><td>|</td></tr>
<tr><td>$\neg$</td><td>\<not></td><td>~</td></tr>
<tr><td>$\neq$</td><td>\<noteq></td><td>~=</td></tr>
</table>

Sets over type $'a$

$'a\ set$

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

$\in$	`\<in>`	`:`
$\subseteq$	`\<subseteq>`	`<=`
$\cup$	`\<union>`	`Un`
$\cap$	`\<inter>`	`Int`

4 Logic and Proof beyond "="

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

blast

- 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 $\downarrow$ $\uparrow$ Proof = lemmas used

external **ATPs**[1]

Characteristics:

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

Do you feel lucky?

[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 `conjI`: $[\![?P; ~?Q]\!] \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$
  $[\![a = b; ~False]\!] \Longrightarrow a = b \wedge False$
- By unification:
  unifying $?P \wedge ?Q$ with $a{=}b \wedge False$
  sets $?P$ to $a{=}b$ and $?Q$ to $False$.

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

Result: $1.\ \ldots \Longrightarrow A$

$2.\ \ldots \Longrightarrow B$

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

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

apply$(rule\ xyz)$

"Backchaining"

Typical backwards rules

$$\frac{?P \quad ?Q}{?P \land ?Q}\ \texttt{conjI}$$

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ \texttt{impI} \qquad \frac{\bigwedge x.\ ?P\ x}{\forall x.\ ?P\ x}\ \texttt{allI}$$

$$\frac{?P \Longrightarrow ?Q \quad ?Q \Longrightarrow ?P}{?P = ?Q}\ \texttt{iffI}$$

They are known as introduction rules
because they *introduce* a particular connective.

Teaching *blast* new intro rules

If r is a theorem $\llbracket\ A_1; \ldots; A_n\ \rrbracket \Longrightarrow A$ then

$$(\mathit{blast\ intro\colon\ r})$$

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

Example:

theorem trans: $\llbracket\ ?x \leq\ ?y;\ ?y \leq\ ?z\ \rrbracket \Longrightarrow\ ?x \leq\ ?z$

goal $\mathit{1.}\ \llbracket\ a \leq b;\ b \leq c;\ c \leq d\ \rrbracket \Longrightarrow a \leq d$

proof **apply**$(\mathit{blast\ intro\colon\ trans})$

Can greatly increase the search space!

Forward proof: OF

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

$$r[OF\ r_1\ \ldots\ r_m]$$

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

Example: theorem `refl`: $?t = ?t$

```
conjI[OF refl[of "a"] refl[of "b"]]
```

$$\rightsquigarrow$$

$$a = a \wedge b = b$$

From now on: ? mostly suppressed on slides

Single_Step_Demo.thy

$\Longrightarrow$ versus $\longrightarrow$

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

$\longrightarrow$ is part of HOL and can occur inside the logical formulas A_i and A.

Phrase theorems like this $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$
not like this $A_1 \land \ldots \land A_n \longrightarrow A$

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:

inductive $ev :: nat \Rightarrow bool$
where
$ev\ 0 \quad |$
$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: $ev\ 4$

$$ev\ 0 \Longrightarrow ev\ 2 \Longrightarrow ev\ 4$$

Consider

fun $even :: nat \Rightarrow bool$ **where**
$even\ 0 = True \mid$
$even\ (Suc\ 0) = False \mid$
$even\ (Suc\ (Suc\ n)) = even\ n$

A trickier proof: $ev\ m \Longrightarrow even\ m$

By induction on the *structure* of the derivation of $ev\ m$

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow even\ m = True$
- rule $ev\ n \Longrightarrow ev\ (n{+}2)$
  $\Longrightarrow m = n{+}2$ and $even\ n$ (IH)
  $\Longrightarrow even\ m = even\ (n{+}2) = even\ n = True$

Rule induction for ev

To prove

$$ev\ n \Longrightarrow P\ n$$

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

- $P\ 0$
- $P\ n \Longrightarrow P(n{+}2)$

Rule `ev.induct`:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ \llbracket\ ev\ n;\ P\ n\ \rrbracket \Longrightarrow P(n{+}2)}{P\ n}$$

Format of inductive definitions

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

$\llbracket\ I\ a_1; \ \ldots\ ;\ I\ a_n\ \rrbracket \Longrightarrow I\ a \ \ |$

$\vdots$

Note:

- I may have multiple arguments.
- Each rule may also contain *side conditions* not involving I.

Rule induction in general

To prove

$$I\ x \Longrightarrow P\ x$$

by *rule induction* on $I\ x$
we must prove for every rule

$$[\![\ I\ a_1;\ \dots\ ;\ I\ a_n\ ]\!] \Longrightarrow I\ a$$

that P is preserved:

$$[\![\ I\ a_1;\ P\ a_1;\ \dots\ ;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

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 $formula_0$
  have $formula_1$ **by** $simp$
  $\vdots$
  have $formula_n$ **by** $blast$
  show $formula_{n+1}$ **by** $\ldots$
qed

proves $formula_0 \Longrightarrow formula_{n+1}$

Isar core syntax

$$
\begin{array}{lcl}
\text{proof} & = & \textbf{proof}\ [\text{method}]\ \ \text{step}^*\ \ \textbf{qed} \\
 & | & \textbf{by}\ \text{method}
\end{array}
$$

$$\text{method} \;=\; (simp\ \ldots) \mid (blast\ \ldots) \mid (induction\ \ldots) \mid \ldots$$

$$
\begin{array}{lcll}
\text{step} & = & \textbf{fix}\ \text{variables} & (\bigwedge) \\
 & | & \textbf{assume}\ \text{prop} & (\Longrightarrow) \\
 & | & [\textbf{from}\ \text{fact}^+]\ \ (\textbf{have} \mid \textbf{show})\ \text{prop}\ \ \text{proof} &
\end{array}
$$

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

$$\text{fact} \;=\; \text{name} \mid \ldots$$

5 Isar: A Language for Structured Proofs

Example: Cantor's theorem

lemma $\neg$ $surj(f :: 'a \Rightarrow 'a\ set)$
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

$this$	$=$	the previous proposition proved or assumed
then	$=$	**from** $this$
thus	$=$	**then show**
hence	$=$	**then have**

using and **with**

(**have**|**show**) prop **using** facts

=

from facts (**have**|**show**) prop

with facts

=

from facts $this$

Structured lemma statement

lemma
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes s: $surj\ f$
  shows $False$
proof $-$ no automatic proof step
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** s
    by$(auto\ simp\text{: }surj\_def)$
  thus $False$ **by** $blast$
qed

Proves $surj\ f \Longrightarrow False$
but $surj\ f$ *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** $y :: \tau_2 \ldots$
assumes *a:* P **and** *b:* $Q \ldots$
shows R

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**

5 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
  $\vdots$
  show R $\ldots$
next
  assume $\neg\ P$
  $\vdots$
  show R $\ldots$
qed

have $P \vee Q$ $\ldots$
then show R
proof
  assume P
  $\vdots$
  show R $\ldots$
next
  assume Q
  $\vdots$
  show R $\ldots$
qed

Contradiction

show $\neg\ P$
proof
  assume P
  $\vdots$
  show $False$ $\ \ldots$
qed

show P
proof $(rule\ ccontr)$
  assume $\neg P$
  $\vdots$
  show $False$ $\ \ldots$
qed

$\longleftrightarrow$

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
  show Q $\ldots$
next
  assume Q
  $\vdots$
  show P $\ldots$
qed

$\forall$ and $\exists$ introduction

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

show $\exists\, x.\ P(x)$
proof
  $\vdots$
  show $P(witness)$ $\ldots$
qed

$\exists$ elimination: **obtain**

have $\exists\, x.\ P(x)$
then obtain x **where** *p:* $P(x)$ **by** *blast*
$\vdots$ $\quad x$ fixed local variable

Works for one or more x

obtain example

lemma $\neg\ surj(f :: {'a} \Rightarrow {'a}\ set)$
proof
  assume $surj\ f$
  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 $a \notin f\ a \longleftrightarrow a \in f\ a$ **by** $blast$
  thus $False$ **by** $blast$
qed

Set equality and subset

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

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\ldots$
qed

Isar_Demo.thy

Exercise

5 Isar: A Language for Structured Proofs

Example: pattern matching

show $formula_1 \longleftrightarrow formula_2$ (**is** $?L \longleftrightarrow ?R$)
proof
  assume $?L$
  $\vdots$
  show $?R$ ...
next
  assume $?R$
  $\vdots$
  show $?L$ ...
qed

$?thesis$

```
show formula  (is ?thesis)
proof -
  ⋮
  show ?thesis  . . .
qed
```

Every show implicitly defines $?thesis$

Quoting facts by value

By name:

> **have** *x0:* $''x > 0''$ ...
> ⋮
> **from** *x0* ...

By value:

> **have** $''x > 0''$ ...
> ⋮
> **from** $`x{>}0`$ ...
>
> ↑ ↑
>
> *back quotes*

Isar_Demo.thy

Pattern matching and quotation

5 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 $xs = rev\ xs$
shows $(\exists\, ys.\ xs = ys\ @\ rev\ ys) \lor$
$(\exists\, ys\ a.\ xs = ys\ @\ a \# rev\ ys)$
proof ???

Isar_Demo.thy

Top down proof development

Proof state and Isar text

In general: **proof** *method*

Applies *method* and generates subgoal(s):

$$\bigwedge x_1 \ \ldots \ x_n \ [\![\ A_1; \ \ldots \ ; \ A_m \ ]\!] \Longrightarrow B$$

How to prove each subgoal:

fix $x_1 \ \ldots \ x_n$
assume $A_1 \ \ldots \ A_m$
$\vdots$
show B

Separated by **next**

Isar_Induction_Demo.thy

Case distinction

Datatype case distinction

datatype $t = C_1\ \vec{\tau}\ \mid\ \ldots$

proof $(cases\ "term")$
  case $(C_1\ x_1\ \ldots\ x_k)$
  $\ldots\ x_j\ \ldots$
next
$\vdots$
qed

where **case** $(C_i\ x_1\ \ldots\ x_k)$ $\equiv$

fix $x_1\ \ldots\ x_k$
assume $\underbrace{C_i\!:}_{\text{label}}\ \underbrace{term = (C_i\ x_1\ \ldots\ x_k)}_{\text{formula}}$

Isar_Induction_Demo.thy

Structural induction for nat

Structural induction for nat

show $P(n)$
proof $(induction\ n)$
  case $\mathit{0}$ $\equiv$ **let** $?case = P(\mathit{0})$
  $\vdots$
  show $?case$
next
  case $(Suc\ n)$ $\equiv$ **fix** n **assume** Suc: $P(n)$
  let $?case = P(Suc\ n)$
  $\vdots$
  $\vdots$
  show $?case$
qed

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$
proof $(induction\ n)$
  case $\mathit{0}$ $\equiv$ **assume** $\mathit{0}$: $A(\mathit{0})$
  $\vdots$ **let** $?case = P(\mathit{0})$
  show $?case$
next
  case $(Suc\ n)$ $\equiv$ **fix** n
  $\vdots$ **assume** Suc: $A(n) \Longrightarrow P(n)$
  $A(Suc\ n)$
  $\vdots$ **let** $?case = P(Suc\ n)$
  show $?case$
qed

Named assumptions

In a proof of

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by structural induction:
In the context of

case C

we have

$C.IH$	the induction hypotheses
$C.prems$	the premises A_i
C	$C.IH + C.prems$

5 Isar: A Language for Structured Proofs

Isar_Induction_Demo.thy

Rule induction

Rule induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: ...
⋮
$rule_n$: ...

show $I\ x\ y \Longrightarrow P\ x\ y$
proof $(induction\ rule{:}\ I.induct)$
  case $rule_1$
  ...
  show $?case$
next
⋮
next
  case $rule_n$
  ...
  show $?case$
qed

Fixing your own variable names

case $(rule_i\ x_1\ \ldots\ x_k)$

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

Named assumptions

In a proof of

$$I \ldots \Longrightarrow A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by rule induction on $I \ldots$:
In the context of

case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

5 Isar: A Language for Structured Proofs

Rule inversion

inductive $ev :: nat \Rightarrow bool$ **where**
$ev0$: $ev\ 0$ |
$evSS$: $ev\ n \Longrightarrow ev(Suc(Suc\ n))$

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

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

Rule inversion = case distinction over rules

Isar_Induction_Demo.thy

Rule inversion

Rule inversion template

from $`ev\ n`$ **have** P
proof $cases$
  case $ev0$ $\qquad n = 0$
  $\vdots$
  show $?thesis\ \ldots$
next
  case $(evSS\ k)$ $\qquad n = Suc\ (Suc\ k),\ ev\ k$
  $\vdots$
  show $?thesis\ \ldots$
qed

Impossible cases disappear automatically