

Concrete Semantics

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2014-5-6

Part I

Isabelle

Chapter 2

Programming and Proving

- ① Overview of Isabelle/HOL
- ② Type and function definitions
- ③ Induction Heuristics
- ④ Simplification

Notation

Implication associates to the right:

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

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

$$\frac{A_1 \quad \dots \quad A_n}{B} \quad \text{means} \quad A_1 \implies \dots \implies A_n \implies B$$

- ① Overview of Isabelle/HOL
- ② Type and function definitions
- ③ Induction Heuristics
- ④ Simplification

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

① Overview of Isabelle/HOL

Types and terms

Interface

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

Summary

Types

Basic syntax:

$\tau ::=$	(τ)	
	$bool \mid nat \mid int \mid \dots$	base types
	$'a \mid 'b \mid \dots$	type variables
	$\tau \Rightarrow \tau$	functions
	$\tau \times \tau$	pairs (ascii: *)
	$\tau \text{ list}$	lists
	$\tau \text{ set}$	sets
	$\dots$	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: $f\ t_1\ t_2\ \dots$

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	constant or variable (identifier)
	$t\ t$	function application
	$\lambda x. t$	function abstraction
	$\dots$	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 *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 ”.

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

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: $f(x::nat)$

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*: $+$, $-$, $*$, $\#$, $@$, $\dots$
- *Mixfix*: *if* $-$ *then* $-$ *else* $-$, *case* $-$ *of*, $\dots$

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:

$$! \quad (if \, - \, then \, - \, else \, -) \quad !$$

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

Concrete syntax

In .thy files:

Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides

① Overview of Isabelle/HOL

Types and terms

Interface

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

Summary

isabelle jedit

- Based on *jedit* editor
- Processes Isabelle text automatically when editing `.thy` files (like modern Java IDEs)

Overview_Demo.thy

① Overview of Isabelle/HOL

Types and terms

Interface

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

Summary

Type *bool*

datatype *bool* = *True* | *False*

Predefined functions:

$\wedge, \vee, \longrightarrow, \dots :: \textit{bool} \Rightarrow \textit{bool} \Rightarrow \textit{bool}$

A *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* ⇒ *nat* ⇒ *nat*

! Numbers and arithmetic operations are overloaded:

$0, 1, 2, \dots :: 'a, \quad + :: 'a \Rightarrow 'a \Rightarrow 'a$

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

Nat_Demo.thy

An informal proof

Lemma $\text{add } m \ 0 = m$

Proof by induction on m .

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

- Case $\text{Suc } m$ (the induction step):

We assume $\text{add } m \ 0 = m$,

the induction hypothesis (IH).

We need to show $\text{add } (\text{Suc } m) \ 0 = \text{Suc } m$.

The proof is as follows:

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

Type *'a list*

Lists of elements of type *'a*

datatype *'a list* = *Nil* | *Cons 'a ('a list)*

Some lists: *Nil*, *Cons 1 Nil*, *Cons 1 (Cons 2 Nil)*, ...

Syntactic sugar:

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

Structural Induction for lists

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

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

$$\frac{P([]) \quad \bigwedge x \ xs. P(xs) \implies P(x\#xs)}{P(xs)}$$

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:

$$\begin{aligned} & app (app (Cons x xs) ys) zs \\ &= Cons x (app (app xs ys) zs) && \text{by definition of } app \\ &= Cons x (app xs (app ys zs)) && \text{by IH} \\ &= app (Cons x xs) (app ys zs) && \text{by definition of } app \end{aligned}$$

Large library: HOL/List.thy

Included in Main.

Don't reinvent, reuse!

Predefined: $xs @ ys$ (append), $length$, and map :

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

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

Interface

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

$$1. \bigwedge x_1 \dots x_p. A \implies B$$

$x_1 \dots x_p$ fixed local variables

A local assumption(s)

B actual (sub)goal

Preview: Multiple assumptions

$$\llbracket A_1; \dots ; A_n \rrbracket \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ “and”

- ① Overview of Isabelle/HOL
- ② Type and function definitions
- ③ Induction Heuristics
- ④ Simplification

② Type and function definitions

Type definitions

Function definitions

Type synonyms

type_synonym *name* = τ

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

$$\begin{array}{rcl} \mathbf{datatype} \ (\alpha_1, \dots, \alpha_n) \tau & = & C_1 \ \tau_{1,1} \dots \tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

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

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Datatype values can be taken apart with *case*:

$$(case\ xs\ of\ [] \Rightarrow \dots \mid y\#\!ys \Rightarrow \dots\ y\ \dots\ ys\ \dots)$$

Wildcards: $_$

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Need $(\)$ in context

Tree_Demo.thy

The *option* type

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

If *'a* has values $a_1, a_2, \dots$

then *'a option* has values *None*, *Some* a_1 , *Some* a_2 , $\dots$

Typical application:

fun *lookup* :: (*'a* $\times$ *'b*) *list* $\Rightarrow$ *'a* $\Rightarrow$ *'b option* **where**
lookup [] *x* = *None* |
lookup ((*a*,*b*) # *ps*) *x* =
 (*if* *a* = *x* *then Some b* *else lookup ps x*)

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 $f\ x_1 \dots x_n = \dots$

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

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

Example: Ackermann

fun *ack* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *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*, -)

primrec

- A restrictive version of **fun**
- Means *primitive recursive*
- Most functions are primitive recursive
- Frequently found in Isabelle theories

The essence of primitive recursion:

$$f(0) = \dots \quad \text{no recursion}$$

$$f(\text{Suc } n) = \dots f(n) \dots$$

$$g([]) = \dots \quad \text{no recursion}$$

$$g(x\#xs) = \dots g(xs) \dots$$

- ① Overview of Isabelle/HOL
- ② Type and function definitions
- ③ Induction Heuristics
- ④ 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 [] = [] |
 rev (x#xs) = *rev* xs @ [x]

A tail recursive version:

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

Induction_Demo.thy

Generalisation

Generalisation

- 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*:

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

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

$$f(e) = \dots f(r_1) \dots f(r_k) \dots$$

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

Induction follows course of (terminating!) computation

Motto: properties of f are best proved by rule $f.induct$

How to apply *f.induct*

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

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

Heuristic:

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

Induction_Demo.thy

Computation Induction

- ① Overview of Isabelle/HOL
- ② Type and function definitions
- ③ Induction Heuristics
- ④ Simplification

Simplification means ...

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *simplification rule*

Simplification = (Term) Rewriting

An example

Equations:

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

Rewriting:

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Example:

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

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

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

is suitable as a *simp*-rule only

if l is “bigger” than r and each P_i

$$n < m \Longrightarrow (n < \text{Suc } m) = \text{True} \quad \text{YES}$$

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

Proof method *simp*

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

apply(*simp add: eq₁ ... eq_n*)

Simplify $P_1 \dots P_m$ and C using

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

Variations:

- (*simp ... del: ...*) 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: ... simp del: ...*)

Rewriting with definitions

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

$$(\textit{simp add: } f_def \dots)$$

f is the function whose definition is to be unfolded.

Case splitting with *simp*

Automatic:

$$\begin{aligned} &P(\text{if } A \text{ then } s \text{ else } t) \\ &= \\ &(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t)) \end{aligned}$$

By hand:

$$\begin{aligned} &P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) \\ &= \\ &(e = 0 \longrightarrow P(a)) \wedge (\forall n. e = \text{Suc } n \longrightarrow P(b)) \end{aligned}$$

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

Or *auto*. Similar for any datatype *t*: *t.split*

Simp_Demo.thy

Chapter 3

Case Study: IMP Expressions

⑤ Case Study: IMP Expressions

⑤ Case Study: IMP Expressions

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

IMP *commands* are introduced later.

⑤ Case Study: IMP Expressions

Arithmetic Expressions

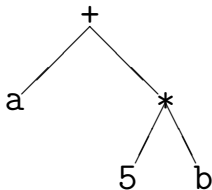
Boolean Expressions

Stack Machine and Compilation

Concrete and abstract syntax

Concrete syntax: strings, eg "a+5*b"

Abstract syntax: trees, eg



Parser: function from strings to trees

Linear view of trees: terms, eg *Plus a (Times 5 b)*

Abstract syntax trees/terms are datatype values!

Concrete syntax is defined by a context-free grammar, eg

$$a ::= n \mid x \mid (a) \mid a + a \mid a * a \mid \dots$$

where n can be any natural number and x any variable.

We focus on *abstract* syntax
which we introduce via datatypes.

Datatype *aexp*

Variable names are strings, values are integers:

type_synonym *vname* = *string*

datatype *aexp* = *N int* | *V vname* | *Plus aexp aexp*

Concrete	Abstract
5	$N\ 5$
x	$V\ "x"$
x+y	$Plus\ (V\ "x")\ (V\ "y")$
2+(z+3)	$Plus\ (N\ 2)\ (Plus\ (V\ "z")\ (N\ 3))$

Warning

This is syntax, not (yet) semantics!

$$N\ 0 \neq Plus\ (N\ 0)\ (N\ 0)$$

The (program) state

What is the value of $x+1$?

- The value of an expression depends on the value of its variables.
- The value of all variables is recorded in the *state*.
- The state is a function from variable names to values:

type_synonym $val = int$

type_synonym $state = vname \Rightarrow val$

Function update notation

If $f :: \tau_1 \Rightarrow \tau_2$ and $a :: \tau_1$ and $b :: \tau_2$ then

$$f(a := b)$$

is the function that behaves like f
except that it returns b for argument a .

$$f(a := b) = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f\ x)$$

How to write down a state

Some states:

- $\lambda x. 0$
- $(\lambda x. 0)(\text{"a"} := 3)$
- $((\lambda x. 0)(\text{"a"} := 5))(\text{"x"} := 3)$

Nicer notation:

$$\langle \text{"a"} := 5, \text{"x"} := 3, \text{"y"} := 7 \rangle$$

Maps everything to 0, but "a" to 5, "x" to 3, etc.

AExp.thy

⑤ Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation

BExp.thy

⑤ Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation

ASM.thy

This was easy.

Because evaluation of expressions always terminates.

But execution of programs may *not* terminate.

Hence we cannot define it by a total recursive function.

We need more logical machinery
to define program execution and reason about it.

Chapter 4

Logic and Proof Beyond Equality

⑥ Logical Formulas

⑦ Proof Automation

⑧ Single Step Proofs

⑨ Inductive Definitions

⑥ Logical Formulas

⑦ Proof Automation

⑧ Single Step Proofs

⑨ Inductive Definitions

Syntax (in decreasing precedence):

$$\begin{array}{lcl} \text{form} ::= & (\text{form}) & | \text{ term} = \text{term} \quad | \quad \neg \text{form} \\ & | \text{ form} \wedge \text{form} & | \text{ form} \vee \text{form} \quad | \quad \text{form} \longrightarrow \text{form} \\ & | \forall x. \text{form} & | \exists x. \text{form} \end{array}$$

Examples:

$$\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$$

$$s = t \wedge C \equiv (s = t) \wedge C$$

$$A \wedge B = B \wedge A \equiv A \wedge (B = B) \wedge A$$

$$\forall x. P x \wedge Q x \equiv \forall x. (P x \wedge Q x)$$

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

Variable binding convention:

$$\forall x\ y. P\ x\ y \equiv \forall x. \forall y. P\ x\ y$$

Similarly for $\exists$ and λ .

Warning

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

$$! \quad P \wedge \forall x. Q x \rightsquigarrow P \wedge (\forall x. Q x) \quad !$$

Mathematical symbols

... and their ascii representations:

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

Sets over type $'a$

$'a$ set

- $\{\}, \{e_1, \dots, e_n\}$
- $e \in A, A \subseteq B$
- $A \cup B, A \cap B, A - B, -A$
- ...

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

Set comprehension

- $\{x. P\}$ where x is a variable
- But not $\{t. P\}$ where t is a proper term
- Instead: $\{t \mid x \ y \ z. P\}$
is short for $\{v. \exists x \ y \ z. v = t \wedge P\}$
where x, y, z are the free variables in t

⑥ Logical Formulas

⑦ Proof Automation

⑧ Single Step Proofs

⑨ 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

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

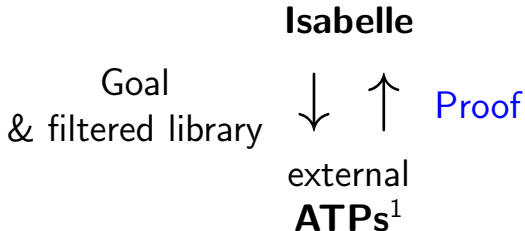
arith:

- proves linear formulas (no “*”)
- complete for quantifier-free *real* arithmetic
- complete for first-order theory of *nat* and *int* (Presburger arithmetic)

Sledgehammer



Architecture:



Characteristics:

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

Do you feel lucky?

¹Automatic Theorem Provers

by(*proof-method*)

$\approx$

apply(*proof-method*)
done

Auto_Proof_Demo.thy

⑥ Logical Formulas

⑦ Proof Automation

⑧ Single Step Proofs

⑨ 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: $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

These *?-variables* can later be instantiated:

- By hand:

$\text{conjI}[\text{of } "a=b" \text{ } "False"] \rightsquigarrow$
 $\llbracket a = b; False \rrbracket \Longrightarrow a = b \wedge False$

- By **unification**:

unifying $?P \wedge ?Q$ with $a=b \wedge False$
sets $?P$ to $a=b$ and $?Q$ to $False$.

Rule application

Example: rule: $\llbracket ?P; ?Q \rrbracket \implies ?P \wedge ?Q$
subgoal: 1. $\dots \implies A \wedge B$

Result: 1. $\dots \implies A$
2. $\dots \implies B$

The general case: applying rule $\llbracket A_1; \dots ; A_n \rrbracket \implies A$
to subgoal $\dots \implies C$:

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

apply(*rule xyz*)

“Backchaining”

Typical backwards rules

$$\frac{?P \quad ?Q}{?P \wedge ?Q} \text{conjI}$$

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

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

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

Teaching *blast* new intro rules

If r is a theorem $\llbracket A_1; \dots; A_n \rrbracket \implies A$ then

$(blast\ intro: r)$

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

Example:

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

goal 1. $\llbracket a \leq b; b \leq c; c \leq d \rrbracket \implies a \leq d$

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

Can greatly increase the search space!

Forward proof: OF

If r is a theorem $A \implies B$

and s is a theorem that unifies with A then

$$r[OF\ s]$$

is the theorem obtained by proving A with s .

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

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

$\leadsto$

$$?Q \implies a = a \wedge ?Q$$

The general case:

If r is a theorem $\llbracket A_1; \dots; A_n \rrbracket \implies A$
and $r_1, \dots, r_m$ ($m \leq n$) are theorems then

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

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

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: $\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A$

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

Phrase theorems like this $\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A$
not like this $A_1 \wedge \dots \wedge A_n \longrightarrow A$

⑥ Logical Formulas

⑦ Proof Automation

⑧ Single Step Proofs

⑨ 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 |  
  even (Suc 0) = False |  
  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**

$\llbracket I\ a_1; \dots ; I\ a_n \rrbracket \Longrightarrow I\ a \mid$

$\vdots$

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

$$\llbracket I\ a_1; \dots ; I\ a_n \rrbracket \Longrightarrow I\ a$$

that P is preserved:

$$\llbracket I\ a_1; P\ a_1; \dots ; I\ a_n; P\ a_n \rrbracket \Longrightarrow P\ a$$

! Rule induction is absolutely central !
to (operational) semantics
and the rest of this lecture course

Inductive_Demo.thy

Inductively defined sets

inductive_set $I :: \tau$ *set* **where**

$$\begin{array}{l} \llbracket a_1 \in I; \dots ; a_n \in I \rrbracket \implies a \in I \mid \\ \vdots \end{array}$$

Difference to **inductive**:

- arguments of I are tupled, not curried
- I can later be used with set theoretic operators, eg $I \cup \dots$

Chapter 5

Isar: A Language for Structured Proofs

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

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** $\dots$

qed

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

Isar core syntax

proof = **proof** [method] step* **qed**
| **by** method

method = (*simp* ...) | (*blast* ...) | (*induction* ...) | ...

step = **fix** variables $(\wedge)$
| **assume** prop $(\implies)$
| [**from** fact⁺] (**have** | **show**) prop proof

prop = [name:] "formula"

fact = name | ...

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Example: Cantor's theorem

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

<i>this</i>	=	the previous proposition proved or assumed
then	=	from <i>this</i>
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 $f :: 'a \Rightarrow 'a \text{ set}$

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 $(\text{auto simp: surj_def})$

thus False **by** blast

qed

Proves $\text{surj } f \Longrightarrow \text{False}$

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** $y :: \tau_2 \dots$
assumes $a: P$ **and** $b: Q \dots$
shows R

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

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Case distinction

show R
proof *cases*
 assume P
 :
 show $R \dots$
next
 assume $\neg P$
 :
 show $R \dots$
qed

have $P \vee Q \dots$
then show R
proof
 assume P
 :
 show $R \dots$
next
 assume Q
 :
 show $R \dots$
qed

Contradiction

```
show  $\neg P$   
proof  
  assume  $P$   
   $\vdots$   
  show False ...  
qed
```

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



```
show  $P \longleftrightarrow Q$   
proof  
  assume  $P$   
   $\vdots$   
  show  $Q \dots$   
next  
  assume  $Q$   
   $\vdots$   
  show  $P \dots$   
qed
```

$\forall$ and $\exists$ introduction

show $\forall x. P(x)$

proof

fix x local fixed variable

show $P(x)$...

qed

show $\exists x. P(x)$

proof

$\vdots$

show $P(witness)$...

qed

$\exists$ elimination: **obtain**

have $\exists x. P(x)$

then obtain x **where** $p: P(x)$ **by** *blast*

$\vdots$ x fixed local variable

Works for one or more x

obtain example

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 $a \notin f a \longleftrightarrow a \in f a$ **by** blast

thus False **by** blast

qed

Set equality and subset

show $A = B$

proof

show $A \subseteq B \dots$

next

show $B \subseteq A \dots$

qed

show $A \subseteq B$

proof

fix x

assume $x \in A$

$\vdots$

show $x \in B \dots$

qed

Isar_Demo.thy

Exercise

⑩ Isar by example

⑪ Proof patterns

⑫ Streamlining Proofs

⑬ Proof by Cases and Induction

12 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

Example: pattern matching

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

?thesis

show *formula* (*is ?thesis*)

proof -

⋮

show *?thesis* ...

qed

Every show implicitly defines *?thesis*

let

Introducing local abbreviations in proofs:

let *?t* = "*some-big-term*"

:

have "... *?t* ... "

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

12 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

Example

lemma

$$(\exists ys\ zs.\ xs = ys @ zs \wedge length\ ys = length\ zs) \vee \\ (\exists ys\ zs.\ xs = ys @ zs \wedge length\ ys = length\ zs + 1)$$

proof ???

Isar_Demo.thy

Top down proof development

When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

have ... **using** ...

apply -

to make incoming facts
part of proof state
or whatever

apply *auto*

apply ...

At the end:

- **done**
- Better: convert to structured proof

12 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks

moreover—ultimately

have $P_1 \dots$

moreover

have $P_2 \dots$

moreover

$\vdots$

moreover

have $P_n \dots$

ultimately

have $P \dots$

$\approx$

have $lab_1: P_1 \dots$

have $lab_2: P_2 \dots$

$\vdots$

have $lab_n: P_n \dots$

from $lab_1 lab_2 \dots$

have $P \dots$

With names

Raw proof blocks

{ **fix** $x_1 \dots x_n$
 assume $A_1 \dots A_m$
 :
 have B
}

proves $\llbracket A_1; \dots ; A_m \rrbracket \Longrightarrow B$

where all x_i have been replaced by $?x_i$.

Isar_Demo.thy

moreover and { }

Proof state and Isar text

In general: **proof** *method*

Applies *method* and generates subgoal(s):

$$\bigwedge x_1 \dots x_n \llbracket A_1; \dots ; A_m \rrbracket \Longrightarrow B$$

How to prove each subgoal:

```
fix  $x_1 \dots x_n$   
assume  $A_1 \dots A_m$   
 $\vdots$   
show  $B$ 
```

Separated by **next**

10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction

Isar_Induction_Demo.thy

Proof by cases

Datatype case analysis

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

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

where $\mathbf{case} \ (C_i \ x_1 \ \dots \ x_k) \equiv$

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

Isar_Induction_Demo.thy

Structural induction for *nat*

Structural induction for nat

show $P(n)$

proof (*induction* n)

case 0 $\equiv$ **let** $?case = P(0)$

$\vdots$

show $?case$

next

case ($Suc\ n$) $\equiv$ **fix** n **assume** $Suc: P(n)$

$\vdots$
 $\vdots$

show $?case$

qed

let $?case = P(Suc\ n)$

Structural induction with $\implies$

show $A(n) \implies P(n)$

proof (*induction n*)

case 0

$\equiv$ **assume** $0: A(0)$

$\vdots$

let $?case = P(0)$

show $?case$

next

case ($Suc\ n$)

$\equiv$ **fix** n

$\vdots$

assume $Suc: A(n) \implies P(n)$
 $A(Suc\ n)$

$\vdots$

let $?case = P(Suc\ n)$

show $?case$

qed

Named assumptions

In a proof of

$$A_1 \implies \dots \implies A_n \implies B$$

by structural induction:

In the context of

case C

we have

$C.IH$ the induction hypotheses

$C.prem_s$ the premises A_i

C $C.IH + C.prem_s$

A remark on style

- **case** (*Suc n*) ... **show** *?case*
is easy to write and maintain
- **fix** *n* **assume** *formula* ... **show** *formula'*
is easier to read:
 - all information is shown locally
 - no contextual references (e.g. *?case*)

13 Proof by Cases and Induction

Rule Induction

Rule Inversion

Isar_Induction_Demo.thy

Rule induction

Rule induction

```
inductive  $I :: \tau \Rightarrow \sigma \Rightarrow \text{bool}$   
where  
   $\text{rule}_1: \dots$   
   $\vdots$   
   $\text{rule}_n: \dots$ 
```

```
show  $I\ x\ y \Longrightarrow P\ x\ y$   
proof (induction rule: I.induct)  
  case  $\text{rule}_1$   
     $\dots$   
    show  $?case$   
next  
   $\vdots$   
next  
  case  $\text{rule}_n$   
     $\dots$   
    show  $?case$   
qed
```

Fixing your own variable names

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

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

Named assumptions

In a proof of

$$I \dots \Longrightarrow A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

by rule induction on $I \dots$:

In the context of

case R

we have

R.IH the induction hypotheses

R.hyps the assumptions of rule R

R.premis the premises A_i

R $R.IH + R.hyps + R.premis$

13 Proof by Cases and Induction

Rule Induction

Rule Inversion

Rule inversion

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

$ev0$: $ev\ 0 \mid$

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

What can we deduce from $ev\ n$?

That it was proved by either $ev0$ or $evSS$!

$$ev\ n \Longrightarrow n = 0 \vee (\exists k. n = Suc\ (Suc\ k) \wedge ev\ k)$$

Rule inversion = case distinction over rules

Isar_Induction_Demo.thy

Rule inversion

Rule inversion template

from 'ev n' **have** P

proof *cases*

case $ev0$

$$n = 0$$

$\vdots$

show *?thesis* ...

next

case $(evSS\ k)$

$$n = Suc\ (Suc\ k),\ ev\ k$$

$\vdots$

show *?thesis* ...

qed

Impossible cases disappear automatically