Concrete Semantics
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2021-3-30
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: \( \Rightarrow, \to \)

\[ \begin{array}{c}
\frac{A_1 \ldots A_n}{B} \\
\end{array} \quad \text{means} \quad A_1 \implies \cdots \implies A_n \implies B \]
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 \( \text{term} = \text{term} \), e.g. \( 1 + 2 = 4 \)
- Later: \( \land, \lor, \rightarrow, \forall, \ldots \)
1 Overview of Isabelle/HOL

Types and terms

Interface

By example: types bool, nat and list

Summary
Types

Basic syntax:

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

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:

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

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 \( \tau \)”. 

\[
\begin{array}{c}
    t :: \tau_1 \Rightarrow \tau_2 \\
    u :: \tau_1 \\
\end{array}
\]

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

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

- *Infix:* $+, -, *, \#, @, \ldots$
- *Mixfix:* `if __ then __ else __`, `case __ of`, \ldots

Prefix binds more strongly than infix:

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

Enclose *if* and *case* in parentheses:

$$\text{(if __ then __ else __)}$$
Theory = Isabelle Module

Syntax:  
\text{theory } \texttt{MyTh} \\
\text{imports } T_1 \ldots T_n \\
\text{begin} \\
(\text{definitions, theorems, proofs, } \ldots)^* \\
\text{end}

\textbf{MyTh:} name of theory. Must live in file \texttt{MyTh.thy}

\textbf{T_i:} names of \textit{imported} theories. Import transitive.

Usually: \texttt{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
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
Overview of Isabelle/HOL

Types and terms

Interface

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

Summary
**Type** bool

**Datatype**
\[ \texttt{bool} = \texttt{True} \mid \texttt{False} \]

Predefined functions:
\[ \land, \lor, \rightarrow, \ldots \quad \vdash \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool} \]

*A formula is a term of type* \texttt{bool}

*if-and-only-if:* \[ = \]
Type \textit{nat}

\textbf{datatype} \ \texttt{nat} = \texttt{0} \mid \texttt{Suc nat}

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

Predefined functions: \texttt{+}, \ \texttt{*}, \ldots \ :: \texttt{nat} \Rightarrow \texttt{nat} \Rightarrow \texttt{nat}

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

You need type annotations: \texttt{1 :: nat}, \texttt{x + (y::nat)} unless the context is unambiguous: \texttt{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 \( \text{Suc} \ m \) (the induction step):
  We assume \( add \ m \ 0 = m \),
  the induction hypothesis (IH).
  We need to show \( add \ (\text{Suc} \ m) \ 0 = \text{Suc} \ m \).

The proof is as follows:

\[
\begin{align*}
add \ (\text{Suc} \ m) \ 0 & = \text{Suc} \ (add \ m \ 0) \quad \text{by def. of add} \\
& = \text{Suc} \ m \quad \text{by IH}
\end{align*}
\]
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, \ldots, x_n] = x_1 \# \ldots \# x_n \# []
To prove that $P(xs)$ for all lists $xs$, prove

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

\[
P([]) \land \forall x \; xs. \; P(xs) \implies P(x\#xs)
\]

\[
\frac{}{P(xs)}
\]
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** \( \text{Nil} \): \( \text{app} (\text{app} \, \text{Nil} \, ys) \, zs = \text{app} \, ys \, zs = \text{app} \, \text{Nil} \, (\text{app} \, ys \, zs) \) holds by definition of \( \text{app} \).

- **Case** \( \text{Cons} \, x \, xs \): 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:

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

Included in Main.

Don’t reinvent, reuse!

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

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

If the lemma is suitable as a simplification rule:

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

Command

\texttt{sorry}

“completes” any proof.

Allows top down development:

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

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

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

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

abbreviates

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

; \quad \approx \quad \text{“and”}
1 Overview of Isabelle/HOL

2 Type and function definitions

3 Induction Heuristics

4 Simplification
Type and function definitions

Type definitions

Function definitions
Type synonyms

define type synonym name = \tau

Introduces a synonym name for type \tau

Examples

define type synonym string = char list

define type synonym (\texttt{\textquoteleft}a,\texttt{\textquoteleft}b)\texttt{foo} = \texttt{\textquoteleft}a list \times \texttt{\textquoteleft}b list

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

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

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

\[
\begin{align*}
\text{(case } x s \text{ of } & \text{ } [ ] \Rightarrow \ldots \mid y \# y s \Rightarrow \ldots y \ldots y s \ldots) \\
\text{Wildcards: } _ & \\
\text{(case } m \text{ of } & \text{ } 0 \Rightarrow \text{Suc } 0 \mid \text{Suc } _ \Rightarrow 0) \\
\text{Nested patterns: } & \\
\text{(case } x s \text{ of } & \text{ } [0] \Rightarrow 0 \mid [\text{Suc } n] \Rightarrow n \mid _ \Rightarrow 2)
\end{align*}
\]

Complicated patterns mean complicated proofs!

Need ( ) in context
Tree_Demo.thy
The `option` type

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

If `'a` has values $a_1$, $a_2$, ... then `'a option` has values `None`, `Some a_1`, `Some a_2`, ...

Typical application:

```haskell
fun lookup :: ('a × 'b) list ⇒ 'a ⇒ '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 \textit{sq} :: \textit{nat} \rightarrow \textit{nat} where \textit{sq} \texttt{n} = \texttt{n} \ast \texttt{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
fun sep :: 'a ⇒ 'a list ⇒ 'a list where
sep a (x#y#zs) = x # a # sep a (y#zs) |
sep a xs = xs
Example: Ackermann

```
fun ack :: nat ⇒ nat ⇒ nat where
ack 0 n = Suc n |
ack (Suc m) 0 = ack m (Suc 0) |
ack (Suc m) (Suc n) = ack m (ack (Suc m) n)
```

Terminates because the arguments decrease lexicographically with each recursive call:

- \((\text{Suc} \ m, 0) > (m, \text{Suc} \ 0)\)
- \((\text{Suc} \ m, \text{Suc} \ n) > (\text{Suc} \ m, n)\)
- \((\text{Suc} \ m, \text{Suc} \ n) > (m, \_ )\)
A restrictive version of `fun`

Means *primitive recursive*

Most functions are primitive recursive

Frequently found in Isabelle theories

**The essence of primitive recursion:**

\[
\begin{align*}
f(0) & = \ldots \quad \text{no recursion} \\
f(Suc \ n) & = \ldots f(n)\ldots \\
g([]) & = \ldots \quad \text{no recursion} \\
g(x\#xs) & = \ldots g(xs)\ldots
\end{align*}
\]
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:

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

A tail recursive version:

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

lemma itrev xs [] = rev xs
```
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 ⇒ nat where

\[
\begin{align*}
\text{div2 } 0 &= 0 \\
\text{div2 } (\text{Suc } 0) &= 0 \\
\text{div2 } (\text{Suc}(\text{Suc } n)) &= \text{Suc}(\text{div2 } n)
\end{align*}
\]

⇝ induction rule div2.induct:

\[
\begin{array}{c}
P(0) \quad P(\text{Suc } 0) \\
\wedge n. \ P(n) \implies P(\text{Suc}(\text{Suc } n))
\end{array}
\]

\[
P(m)
\]
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 $f.induct$
How to apply $f\.induct$

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
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 $= (\text{Term})$ Rewriting
An example

Equations:

\[ 0 + n = n \quad (1) \]
\[ (Succ \ m) + n = Succ \ (m + n) \quad (2) \]
\[ (Succ \ m \leq Succ \ n) = (m \leq n) \quad (3) \]
\[ (0 \leq m) = True \quad (4) \]

Rewriting:

\[ 0 + Succ \ 0 \leq Succ \ 0 + x \quad (1) \]
\[ Succ \ 0 \leq Succ \ 0 + x \quad (2) \]
\[ Succ \ 0 \leq Succ \ (0 + x) \quad (3) \]
\[ 0 \leq 0 + x \quad (4) \]
\[ True \]
Conditional rewriting

Simplification rules can be conditional:

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

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

Example

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

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

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

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

Principle:

\[
\begin{align*}
\left[ P_1; \ldots; P_k \right] \implies l = r
\end{align*}
\]

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

\[
\begin{align*}
n < m \implies (n < \text{Suc} \ m) & = True \quad \text{YES} \\
\text{Suc} \ n < m \implies (n < m) & = True \quad \text{NO}
\end{align*}
\]
Proof method \textit{simp}

Goal: 1. $\left\lbrack P_1; \ldots; P_m \right\rbrack \Rightarrow C$

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

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

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

Variations:

- $(\textit{simp} \ldots \textit{del:} \ldots)$ 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: ...)

70
Rewriting with definitions

Definitions (definition) must be used explicitly:

\[(simp \ add: f\_def \ldots)\]

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

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
Chapter 3

Case Study: IMP Expressions
Case Study: IMP Expressions
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

```
+  
/  
/   
/    
a    *
/     
/      
5      b
```

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

\[ a ::= n \mid x \mid (a) \mid a + a \mid a \times a \mid \ldots \]

where \( n \) can be any natural number and \( x \) any variable.

We focus on abstract syntax which we introduce via datatypes.
Datatype \emph{aexp}

Variable names are strings, values are integers:

\begin{align*}
\text{type\_synonym} & \quad vname = \text{string} \\
\text{datatype} & \quad aexp = N \text{ int} \mid V \text{ vname} \mid \text{Plus } aexp \ aexp
\end{align*}

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$N \ 5$</td>
</tr>
<tr>
<td>$x$</td>
<td>$V \ &quot;x&quot;$</td>
</tr>
<tr>
<td>$x + y$</td>
<td>Plus ($V \ &quot;x&quot;$) ($V \ &quot;y&quot;$)</td>
</tr>
<tr>
<td>$2 + (z + 3)$</td>
<td>Plus ($N \ 2$) ($Plus (V \ &quot;z&quot; ) (N \ 3)$)</td>
</tr>
</tbody>
</table>
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 \textit{state}.
- The state is a function from variable names to values:

\begin{verbatim}
type_synonym val = int
type_synonym state = vname ⇒ val
\end{verbatim}
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:

- \( \lambda x. 0 \)
- \( (\lambda x. 0)("a" := 3) \)
- \( ((\lambda x. 0)("a" := 5))("x" := 3) \)

Nicer notation:

\[ <"a" := 5, "x" := 3, "y" := 7> \]

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

$$form ::= (form) \mid term = term \mid \neg form$$
$$\mid form \land form \mid form \lor form \mid \forall x. \ form \mid \exists x. \ form$$

Examples:

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

Input syntax: $$\longleftrightarrow$$ (same precedence as $$\rightarrow$$)
Variable binding convention:

\[ \forall x \ y. \ P \ x \ y \equiv \ \forall x. \ \forall y. \ P \ x \ y \]

Similarly for \( \exists \) and \( \lambda \).
Warning

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

\[ P \land \forall x. \ Q \ x \ \leadsto \ P \land (\forall x. \ Q \ x) \]

\[ P \land \forall x. \ Q \ x \ \leadsto \ P \land (\forall x. \ Q \ x) \]
Mathematical symbols
and their ascii representations

∀ \forall ALL
∃ \exists EX
λ \lambda %
\rightarrow -->
\leftrightarrow <->
∧ \wedge &
∨ \vee |
¬ \text{not} ~
\neq \text{noteq} \neq
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,  - A
- ...

\in \ \langle\text{in}\rangle\quad : \quad \subseteq \ \langle\text{subseteq}\rangle \quad \leq\quad \cup \ \langle\text{union}\rangle \quad \text{Un}\quad \cap \ \langle\text{inter}\rangle \quad \text{Int}
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 \( \{ v. \ \exists x \ y \ z. \ v = t \land P \} \) 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
• 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

\textit{arith:}

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

- Isabelle
- Goal & filtered library → Proof
- external ATPs

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

Do you feel lucky?

---

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

≈

apply\((proof\text{-}method)\)

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: \([?P; ?Q] \Rightarrow ?P \land ?Q\)

These ?-variables can later be instantiated:

- **By hand:**
  
  ```
  conjI[of "a=b" "False"] \leadsto 
  [a = b; False] \Rightarrow 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: \[
\begin{array}{c}
?P; ?Q \\
\end{array}
\] \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 \[
\begin{array}{c}
A_1; \ldots ; A_n
\end{array}
\] \implies A
to subgoal \ldots \implies C:

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

apply(\textit{rule xyz})

“Backchaining”
Typical backwards rules

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

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

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

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

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

\[
(blast \ intro: \ r)
\]

allows \texttt{blast} to backchain on \( r \) during proof search.

Example:

\texttt{theorem le\_trans: [ \( ?x \leq ?y \); \( ?y \leq ?z \) ] \implies ?x \leq ?z}

\texttt{goal 1. [ \( a \leq b \); \( b \leq c \); \( c \leq d \) ] \implies a \leq d}

\texttt{proof apply(blast intro: le\_trans)}

Also works for \texttt{auto} and \texttt{fastforce}

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

$$r[\text{OF } s]$$

is the theorem obtained by proving $A$ with $s$.

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

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

$$\implies$$

$$?Q \implies a = a \land ?Q$$
The general case:

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

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

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

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

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

\[ \implies \]

\[ a = a \land b = b \]
From now on: ? mostly suppressed on slides
Single_Step_Demo.thy
\( \iff \) is part of the Isabelle framework. It structures theorems and proof states: \([ A_1; \ldots; A_n \] \( \iff \) \( A \)

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

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

```isabelle
inductive ev :: nat ⇒ bool
where
  ev 0 |
  ev n ⟷ ev (n + 2)
```
An easy proof: $ev\ 4$

$ev\ 0 \implies ev\ 2 \implies ev\ 4$
Consider

\textbf{fun} \ evn :: nat \Rightarrow bool \ \textbf{where}
\begin{align*}
\text{evn 0} &= \text{True} \ |
\text{evn} \ (\text{Suc 0}) &= \text{False} \ |
\text{evn} \ (\text{Suc} \ (\text{Suc} \ n)) &= \text{evn} \ n
\end{align*}

A trickier proof: \ ev m \implies evn m

By induction on the \textit{structure} of the derivation of \ ev m

Two cases: \ ev m is proved by

- rule \ ev 0
  \[ \implies m = 0 \implies evn m = \text{True} \]

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

To prove

$$ev \ n \ \implies \ P \ n$$

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

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

Rule $ev.induct$:

$$\frac{ev \ n \quad P \ 0 \quad \bigwedge n. \ [ev \ n; \ P \ n] \ \implies \ P(n+2)}{P \ n}$$
Format of inductive definitions

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

\[
\begin{align*}
& \left[ I \ a_1 ; \ldots ; I \ a_n \right] \implies I \ a \\
& \vdots
\end{align*}
\]

**Note:**

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

To prove

$$I \ x \ \implies \ P \ x$$

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

$$[[ I \ a_1; \ldots; I \ a_n ]] \ \implies \ I \ a$$

that $P$ is preserved:

$$[[ I \ a_1; \ P \ a_1; \ldots; I \ a_n; \ P \ a_n ]] \ \implies \ P \ a$$
Rule induction is absolutely central to (operational) semantics and the rest of this lecture course.
Inductive_Demo.thy
Inductively defined sets

\[
\text{inductive_set } I :: \tau \ set \ \text{where} \\
\begin{array}{l}
\left[ a_1 \in I; \ldots ; a_n \in I \right] \Rightarrow a \in I \\
\end{array}
\]

Difference to \texttt{inductive}:

\begin{itemize}
\item arguments of \( I \) are tupled, not curried
\item \( I \) can later be used with set theoretic operators, eg \( I \cup \ldots \)
\end{itemize}
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 assertions

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

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

qed

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

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

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

step = fix variables (∧)
| assume prop (⇒)
| [from fact+] (have | show) prop proof

prop = [name:] ”formula”

fact = name | ...
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Example: Cantor’s theorem

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

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 \((\text{simp add: surj_def})\)

  from \(b\) have \(c: \exists a. \{x. x \notin f\ x\} = f\ a\)
    by \text{blast}

  from \(c\) show \(\text{False}\)
    by \text{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
  \textbf{fixes} \ f :: 'a \Rightarrow 'a set
  \textbf{assumes} \ s: \textit{surj} \ f
  \textbf{shows} \ False

\textbf{proof} \textemdash \textit{no automatic proof step}

\textbf{have} \ \exists \ a. \ \{x. \ x \notin f \ x\} = f \ a \ \textbf{using} \ s
  \ \textbf{by} (\textit{auto simp: surj_def})

\textbf{thus} \ False \ \textbf{by} \ \textit{blast}

\textbf{qed}

Proves \ \textit{surj} \ f \ \Rightarrow \ False

but \ \textit{surj} \ f \ \textit{becomes local fact} \ s \ \textit{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 :: τ₁ and y :: τ₂ . . .
assumes a: P and b: Q . . .
shows R

• fixes and assumes sections optional
• shows optional if no fixes and assumes
```
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Case distinction

```plaintext
show R
proof cases
  assume P
  : show R ⟨proof⟩
next
  assume ¬ P
  : show R ⟨proof⟩
qed

have P ∨ Q ⟨proof⟩
then show R
proof
  assume P
  : show R ⟨proof⟩
next
  assume Q
  : show R ⟨proof⟩
qed
```
Contradiction

\begin{align*}
\text{show } & \neg P \\
\text{proof } & \\
\text{assume } & P \\
\text{::} & \\
\text{show } & \text{False} \langle \text{proof} \rangle \\
\text{qed} & \\
\text{show } & P \\
\text{proof } & (\text{rule ccontr}) \\
\text{assume } & \neg P \\
\text{::} & \\
\text{show } & \text{False} \langle \text{proof} \rangle \\
\text{qed} &
\end{align*}
show \( P \leftrightarrow Q \)
proof
  assume \( P \)
  : 
  : 
  show \( Q \) \(\langle \text{proof} \rangle\)
next
  assume \( Q \)
  : 
  : 
  : 
  show \( P \) \(\langle \text{proof} \rangle\)
qed
∀ and ∃ introduction

show \( \forall x. P(x) \)
proof
\begin{array}{l}
\text{fix } x \quad \text{local fixed variable} \\
\text{show } P(x) \langle proof \rangle
\end{array}
qed

show \( \exists x. P(x) \)
proof
\begin{array}{l}
\vdots \\
\text{show } P(\text{witness}) \langle proof \rangle
\end{array}
qed
∃ elimination: obtain

have \( \exists x. P(x) \)
then obtain \( x \) where \( p: P(x) \) by blast
\[ x \text{ fixed local variable} \]

Works for one or more \( x \)
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 $a \notin f\ a \iff a \in f\ a$ by blast
thus False by blast
qed
Set equality and subset

show \( A = B \)
proof
  show \( A \subseteq B \) ⟨proof⟩
next
  show \( B \subseteq A \) ⟨proof⟩
qed

show \( A \subseteq B \)
proof
  fix \( x \)
  assume \( x \in A \)
  ⟨proof⟩
  |
  show \( x \in B \) ⟨proof⟩
qed
Isar_Demo.thy

Exercise
10 Isar by example

11 Proof patterns

12 Streamlining Proofs

13 Proof by Cases and Induction
Streamlining Proofs

Pattern Matching and Quotations
Top down proof development
moreover
Local lemmas
Example: pattern matching

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

\begin{proof}
  \begin{align*}
    \text{assume } & \textit{?L} \\
    \text{\vdots} \\
    \text{show } & \textit{?R} \langle \textit{proof} \rangle \\
  \end{align*}
\end{proof}

\begin{proof}
  \begin{align*}
    \text{assume } & \textit{?R} \\
    \text{\vdots} \\
    \text{show } & \textit{?L} \langle \textit{proof} \rangle \\
  \end{align*}
\end{proof}

qed
Every show implicitly defines $\text{thesis}$
Introducing local abbreviations in proofs:

```ml
let ?t = "some-big-term"
;
have "... ?t ..."
```
Quoting facts by value

By name:

```prolog
have x0: "x > 0" ...
: from x0 ...
```

By value:

```prolog
have "x > 0" ...
: from 'x>0' ...
```

↑ ↑

*back quotes*
Isar_Demo.thy

Pattern matching and quotations
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
Example

**lemma**

\[ \exists y_1 z_1. x_1 = y_1 @ z_1 \land (\text{length } y_1 = \text{length } z_1 \vee \text{length } y_1 = \text{length } z_1 + 1) \]

**proof ???**
Isar_Demo.thy

Top down proof development
When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

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

**apply** - 

**apply** *auto*  

**apply** . . .

At the end:

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

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
moreover—ultimately

have $P_1 \ldots$
moreover
have $P_2 \ldots$
moreover

\vdots

moreover
have $P_n \ldots$
ultimately
have $P \ldots$

have $\text{lab}_1$: $P_1 \ldots$
have $\text{lab}_2$: $P_2 \ldots$

\vdots

have $\text{lab}_n$: $P_n \ldots$
from $\text{lab}_1 \text{ lab}_2 \ldots$
have $P \ldots$

With names
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
Local lemmas

have $B$ if name: $A_1 \ldots A_m$ for $x_1 \ldots x_n$

⟨proof⟩

proves $\left[ A_1; \ldots ; A_m \right] \implies B$

where all $x_i$ have been replaced by $?x_i$. 
Proof state and Isar text

In general: \textbf{proof method}

Applies \textit{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:

\begin{align*}
\text{fix } x_1 \ldots x_n \\
\text{assume } A_1 \ldots A_m \\
\vdots \\
\text{show } B
\end{align*}

Separated by \texttt{next}
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Isar_Induction_Demo.thy

Proof by cases
Datatype case analysis

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

proof \((\text{cases } "\text{term}"))

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

\ldots x_j \ldots

next

\vdots

qed

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

\text{fix } x_1 \ldots x_k

\text{assume } C_i:\ 

\underbrace{\text{term} = (C_i x_1 \ldots x_k)}_{\text{formula}}
Isar_Induction_Demo.thy

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

\[
\text{show } P(n) \\
\text{proof } (\text{induction } n) \\
\quad \text{case } 0 \\
\quad \quad \vdots \\
\quad \quad \text{show } ?\text{case} \\
\text{next} \\
\quad \text{case } (\text{Suc } n) \\
\quad \quad \vdots \\
\quad \quad \vdots \\
\quad \quad \text{show } ?\text{case} \\
\text{qed}
\]

\[
\equiv \quad \text{let } ?\text{case} = P(0) \\
\equiv \quad \text{fix } n \text{ assume } \text{Suc: } P(n) \\
\quad \text{let } ?\text{case} = P(\text{Suc } n)
\]
Structural induction with \( \Rightarrow \)

show \( A(n) \Rightarrow P(n) \)

proof (induction \( n \))

\begin{align*}
\text{case } 0 & \quad \equiv \quad \text{assume } 0: A(0) \\
\text{let } ?\text{case} = P(0) & \\
\text{show } ?\text{case} & \\
\text{qed}
\end{align*}

next

\begin{align*}
\text{case } (\text{Suc } n) & \quad \equiv \quad \text{fix } n \\
\text{assume } \text{Suc}: A(n) \Rightarrow P(n) & \\
A(\text{Suc } n) & \\
\text{let } ?\text{case} = P(\text{Suc } n) & \\
\text{show } ?\text{case} & \\
\text{qed}
\end{align*}
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** \( C.IH + C.prems \)
A remark on style

- **case** \((\text{Suc } n) \ldots \text{ show } ?\text{case}\)
  - is easy to write and maintain
- **fix** \(n\) **assume** \(\text{formula} \ldots \text{ show } \text{formula}'\)
  - is easier to read:
    - all information is shown locally
    - no contextual references (e.g. \(?\text{case}\))
Proof by Cases and Induction

Rule Induction

Rule Inversion
Isar_Induction_Demo.thy

Rule induction
**Rule induction**

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

**where**

$rule_1: \ldots$

$\vdots$

$rule_n: \ldots$

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

**proof** (induction rule: $I.induct$)

case $rule_1$

$\ldots$

show $?case$

next

$\vdots$

next

case $rule_n$

$\ldots$

show $?case$

qed
Fixing your own variable names

\texttt{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
\[ I \ldots \Rightarrow A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B \]
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 \( A_i \)

\( R \) \( R.IH + R.hyps + R.prems \)
Proof by Cases and Induction

Rule Induction

Rule Inversion
Rule inversion

\textbf{inductive} \ ev :: \ nat \Rightarrow \ bool \ where \\
\texttt{ev0}: \ ev \ 0 \ |
\texttt{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