Concrete Semantics
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2023-1-31
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: \( \Rightarrow, \rightarrow \)

\[
\begin{array}{c}
\frac{A_1 \cdots A_n}{B} \\
\end{array}
\quad \text{means} \quad A_1 \implies \cdots \implies A_n \implies B
\]
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: \( \land, \lor, \rightarrow, \forall, \ldots \)
Overview of Isabelle/HOL

Types and terms

Interface

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

Summary
Types

Basic syntax:

\[ \tau ::= (\tau) \]

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

Convention: \( \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \equiv \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3) \)
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:

\[ t ::= (t) \]

\[ | \ a \ \text{constant or variable (identifier)} \]

\[ | \ t \ t \ \text{function application} \]

\[ | \ \lambda x. \ t \ \text{function abstraction} \]

\[ | \ \ldots \ \text{lots of syntactic sugar} \]

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 \textit{\lambda-calculus}.
The computation rule of the $\lambda$-calculus is the replacement of formal by actual parameters:

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

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

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

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

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

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

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

User can help with *type annotations* inside the term. Example: \( f \ (x::\text{nat}) \)
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*: +, −, *, #, @, ...
- *Mixfix*: if _ then _ else _, case _ of, ...

Prefix binds more strongly than infix:

\[
! f \ x + \ y \equiv (f \ x) + y \neq f (x + y) !
\]

Enclose if and case in parentheses:

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

Syntax:  
theory \textit{MyTh}  
imports \textit{T}_1 \ldots \textit{T}_n  
begin  
(definitions, theorems, proofs, ...)  
end  

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

Usually: \textit{imports} \textit{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
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 \textit{bool}, \textit{nat} and \textit{list}

Summary
**Type bool**

**datatype** \( bool = True \mid False \)

Predefined functions:
\( \land, \lor, \implies, \ldots \) :: \( bool \Rightarrow bool \Rightarrow bool \)

A *formula* is a term of type \( bool \)

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

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

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

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

! Numbers and arithmetic operations are overloaded:

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

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

unless the context is unambiguous: \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 \( Suc \, m \)** (the induction step):
  We assume \( add \, m \, 0 = m \),
  the induction hypothesis (IH).
  We need to show \( add \, (Suc \, m) \, 0 = Suc \, m \).
  The proof is as follows:
  \[
  add \, (Suc \, m) \, 0 \quad = \quad Suc \, (add \, m \, 0) \quad \text{by def. of \( add \)}
  \]
  \[
  = \quad Suc \, m \quad \text{by IH}
  \]
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")
• [x₁, ..., xₙ] = x₁ # ... xₙ # []
Structural Induction for lists

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

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

\[
\begin{align*}
P(\[]) \quad \land \quad x \quad xs. \quad P(xs) & \quad \implies \quad P(x\#xs) \\
\hline
P(xs) & \\
\end{align*}
\]
List_Demo.thy
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 $map$
Overview of Isabelle/HOL

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

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

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

• *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

\texttt{sorry}

“completes” any proof.

Allows top down development:

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

1. $\wedge x_1 \ldots x_p. \ A \implies B$

$x_1 \ldots x_p$ fixed local variables

$A$ local assumption(s)

$B$ actual (sub)goal
Multiple assumptions

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

abbreviates

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

; \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

\textbf{type\_synonym} \textit{name} = \tau

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

\textbf{Examples}

\textbf{type\_synonym} \textit{string} = \textit{char list}

\textbf{type\_synonym} (\textit{a,}\textit{b})\textit{foo} = \textit{a list} \times \textit{b list}

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

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

- **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:

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

Wildcards: \_ \\

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

Nested patterns:

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

Complicated patterns mean complicated proofs! 

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

datatype 'a option = None | Some 'a

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

Typical application:

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** \( sq :: \text{nat} \Rightarrow \text{nat} \) **where** \( sq \ n = n \times n \)

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

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:

• \((\text{Suc } m, 0) > (m, \text{Suc } 0)\)
• \((\text{Suc } m, \text{Suc } n) > (\text{Suc } m, n)\)
• \((\text{Suc } m, \text{Suc } n) > (m, _)\)
primrec

- 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
```
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 ⇒ nat where
  div2 0 = 0 |
  div2 (Suc 0) = 0 |
  div2 (Suc(Suc n)) = Suc(div2 n)
```

\[ \Rightarrow \text{ induction rule } \texttt{div2.induct}: \]

\[
\begin{align*}
P(0) & \quad P(Suc 0) \quad \land n. \ P(n) \implies P(Suc(Suc n)) \\
\end{align*}
\]

\[ P(m) \]
Computation Induction

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

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 $f.\text{induct}$

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

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

Heuristic:

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

\[ 0 + n = n \quad (1) \]
\[ (\text{Suc } m) + n = \text{Suc } (m + n) \quad (2) \]
\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \quad (3) \]
\[ (0 \leq m) = \text{True} \quad (4) \]

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \quad (1) \equiv \]
\[ \text{Suc } 0 \leq \text{Suc } 0 + x \quad (2) \equiv \]
\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \quad (3) \equiv \]
\[ 0 \leq 0 + x \quad (4) \equiv \]
\[ \text{True} \]
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

\[
p(0) = True
\]

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

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

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

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

Principle:

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

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

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

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

\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 \textbf{fun} and \textbf{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:
  
  \[
  \text{(auto simp add: \ldots simp del: \ldots)}
  \]
Rewriting with definitions

Definitions (definition) must be used explicitly:

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

\(f\) is the function whose definition is to be unfolded.
Case splitting with *simp/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
5 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.
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 \( \text{Plus} \ a \ (\text{Times} \ 5 \ b) \)

Abstract syntax trees/terms are datatype values!
Concrete syntax is defined by a context-free grammar, eg

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

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

We focus on abstract syntax which we introduce via datatypes.
Datatype \( aexp \)

Variable names are strings, values are integers:

**type_synonym** \( vname = \text{string} \)

**datatype** \( aexp = N \text{ int} | V vname | \text{Plus } aexp \ aexp \)

<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>( \text{Plus} \ (V \ &quot;x&quot;) \ (V \ &quot;y&quot;) )</td>
</tr>
<tr>
<td>( 2+(z+3) )</td>
<td>( \text{Plus} \ (N \ 2) \ (\text{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 \Rightarrow 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.
AExp.thy
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):

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

Examples:

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

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

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

∀ \<forall> ALL
∃ \<exists> EX
λ \<lambda> %
\rightarrow -->
\leftrightarrow <->
∧ \&
∨ |
¬ \<not> ~
\neq \<noteq> ~=
Sets over type 'a

'a set

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

\in \quad \textless\text{in}\textgreater \quad : \quad \subseteq \quad \textless\text{subseteq}\textgreater \quad \leq \quad \cup \quad \textless\text{union}\textgreater \quad Un \quad \cap \quad \textless\text{inter}\textgreater \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?

1Automatic 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:
  
  ```isar`
  `conjI[of "a=b" "False"]` ~→
  
  $[a = b; False] \Rightarrow a = b \land False$
  ```

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

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

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

Result: 1. ... $\implies A$

2. ... $\implies B$

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

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

apply(rule xy)

“Backchaining”
Typical backwards rules

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

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

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

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

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

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

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

Example:

- \texttt{theorem \textit{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 \textit{apply}(\texttt{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

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

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

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

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

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

Phrase theorems like this $\{ A_1; \ldots; A_n \} \implies A$
not like this $A_1 \land \ldots \land A_n \implies A$
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

fun evn :: nat ⇒ bool where
  evn 0 = True |
  evn (Suc 0) = False |
  evn (Suc (Suc n)) = evn n

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

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

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

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

- rule \( ev \ n \implies ev \ (n+2) \)
  \[ \implies m = n+2 \text{ and } evn \ n \text{ (IH)} \]
  \[ \implies evn \ m = evn \ (n+2) = evn \ n = True \]
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 $ev\.induct$:

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

\textbf{inductive} \( I :: \tau \Rightarrow bool \) \textbf{where} \\
[ \quad I \ a_1; \ldots ; \ I \ a_n \ ] \Rightarrow \ I \ a \ |

\textbf{Note:}

\begin{itemize}
  \item \( I \) may have multiple arguments.
  \item Each rule may also contain \textit{side conditions} not involving \( I \).
\end{itemize}
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

```haskell
inductive_set I :: τ set where
  \[ a_1 \in I; \ldots ; a_n \in I \] \implies a \in I
```

Difference to `inductive`:
- arguments of $I$ are tupled, not curried
- $I$ can later be used with set theoretic operators, eg $I \cup \ldots$
Chapter 5

Isar: A Language for Structured Proofs
Isar by example

Proof patterns

Streamlining Proofs

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 \( formula_0 \)
  have \( formula_1 \) by simp
  
  have \( formula_n \) by blast
  show \( formula_{n+1} \) by \ldots

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  (\&)
       |  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 \text{surj}, show \text{False}

  assume \( a: \text{surj} f \)

  from \( a \) have \( b: \forall A. \exists a. A = f a \)
  
    by (simp add: \text{surj\_def})

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

  from \( c \) show \text{False}
  
    by blast

qed
Isar_Demo.thy

Cantor and abbreviations
Abbreviations

\textit{this} = the previous proposition proved or assumed

\textit{then} = \textbf{from} \textit{this}

\textit{thus} = \textbf{then show}

\textit{hence} = \textbf{then have}
using and with

\[(\text{have} | \text{show}) \text{ prop } \text{using facts} \]
\[= \]
\[\text{from facts (have} | \text{show}) \text{ prop with facts} \]
\[= \]
\[\text{from facts this} \]
Structured lemma statement

**Lemma**

- **fixes** $f :: 'a \Rightarrow 'a \text{ set}$
- **assumes** $s : \text{surj } f$
- **shows** $\text{False}$

**Proof** — no automatic proof step

- **have** $\exists \ a. \ \{x. \ x \notin f \ x\} = f \ a$ using $s$
  - by (auto simp: surj_def)
- **thus** $\text{False}$ by blast

**Qed**

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

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

\textbf{fixes} \( x :: \tau_1 \) and \( y :: \tau_2 \) \ldots

\textbf{assumes} \( a: P \) and \( b: Q \) \ldots

\textbf{shows} \( R \)

\begin{itemize}
  \item \textbf{fixes} and \textbf{assumes} sections optional
  \item \textbf{shows} optional if no \textbf{fixes} and \textbf{assumes}
\end{itemize}
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 $R$ ⟨proof⟩
next
  assume $\neg P$
  :
  show $R$ ⟨proof⟩
qed

have $P \lor Q$ ⟨proof⟩
then show $R$
proof
  assume $P$
  :
  show $R$ ⟨proof⟩
next
  assume $Q$
  :
  show $R$ ⟨proof⟩
qed
show \neg P
proof
  assume P
  : show False ⟨proof⟩
qed

show P
proof (rule ccontr)
  assume \neg P
  : show False ⟨proof⟩
qed
show $P \iff Q$

**proof**

assume $P$

... 

show $Q \langle proof \rangle$

**next**

assume $Q$

... 

show $P \langle proof \rangle$

**qed**
∀ and ∃ introduction

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

show $\exists x. P(x)$
proof
  show $P(witness) \langle proof \rangle$
qed
∃ elimination: obtain

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

: x fixed local variable

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

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

qed
Set equality and subset

\[
\begin{align*}
\text{show} & \quad A = B \\
\text{proof} & \\
& \quad \text{show} \; A \subseteq B \langle \text{proof} \rangle \\
\text{next} & \\
& \quad \text{show} \; B \subseteq A \langle \text{proof} \rangle \\
\text{qed} & \\
\end{align*}
\]

\[
\begin{align*}
\text{show} & \quad A \subseteq B \\
\text{proof} & \\
& \quad \text{fix} \; x \\
& \quad \text{assume} \; x \in A \\
& \quad : \\
& \quad \text{show} \; x \in B \langle \text{proof} \rangle \\
\text{qed} & \\
\end{align*}
\]
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

show $\text{formula}_1 \leftrightarrow \text{formula}_2$ (is $?L \leftrightarrow ?R$)
proof
  assume $?L$
  :
  show $?R \langle \text{proof} \rangle$
next
  assume $?R$
  :
  show $?L \langle \text{proof} \rangle$
qed
show \( \text{formula} \ (is \ \text{thesis}) \)
proof -
  :
    show \( \text{thesis} \ \langle \text{proof} \rangle \)
qed

Every show implicitly defines \( \text{thesis} \)
Introducing local abbreviations in proofs:

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

By name:

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

By value:

```plaintext
have "x > 0" ... :
from \{x > 0\} ...
```

\([\text{open}] \quad \text{\textclose}\)
Isar_Demo.thy

Pattern matching and quotations
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
Example

Lemma

\[ \exists ys\ zs.\ xs = ys \ @\ zs \land\ (\text{length}\ ys = \text{length}\ zs \lor \text{length}\ ys = \text{length}\ zs + 1) \]

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
- **apply** *auto* or whatever
- **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
: 
moreover
have $P_n \ldots$
ultimately
have $P \ldots$

$\approx$

have $\text{lab}_1: P_1 \ldots$
have $\text{lab}_2: P_2 \ldots$
: 
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 $[ A_1; \ldots ; A_m ] \implies B$

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):
\[ \bigwedge_{x_1 \ldots x_n} \, \left[ A_1; \ldots ; A_m \right] \implies B \]

How to prove each subgoal:
\begin{align*}
\text{fix} & \quad x_1 \ldots x_n \\
\text{assume} & \quad A_1 \ldots A_m \\
\text{show} & \quad B \\
\end{align*}

Separated by \texttt{next}
Isar_Induction_Demo.thy

Proof by cases
Datatype case analysis

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

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

where

\[
\text{case } (C_i x_1 \ldots x_k) \equiv \\
\text{fix } x_1 \ldots x_k
\]

\[
\text{assume } C_i: \quad \text{term} = (C_i x_1 \ldots x_k)
\]

label formula
Isar_Induction_Demo.thy

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

show \( P(n) \)

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

\[
\text{case } 0 \\
\vdots \\
\text{show } \ ?\text{case}
\]

\[
\equiv \quad \text{let } \ ?\text{case} = P(0)
\]

next

\[
\text{case } (\text{Suc } n) \\
\vdots \\
\vdots \\
\vdots \\
\text{show } \ ?\text{case}
\]

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

qed
Structural induction with \( \Rightarrow \)

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

\textbf{proof} \ (\textit{induction } n)

\begin{align*}
\text{case } 0 & \quad \equiv \quad \text{assume } 0: A(0) \\
\vdots & \\
\text{show } \text{?case} & \\
\text{next} & \\
\text{case } (\text{Suc } n) & \quad \equiv \quad \text{fix } n \\
\vdots & \\
\vdots & \\
\text{show } \text{?case} & \\
\text{qed}
\end{align*}

\[
\text{let } \text{?case } = P(0)
\]

\[
\text{assume } \text{Suc}: A(n) \Rightarrow P(n) \\
A(\text{Suc } n)
\]

\[
\text{let } \text{?case } = P(\text{Suc } n)
\]
Named assumptions

In a proof of

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

by structural induction:

In the context of

\textbf{case} \( C \)

we have

\textbf{C.IH} \quad \text{the induction hypotheses}

\textbf{C.prems} \quad \text{the premises} \( A_i \)

\( C \quad C.IH + C.prems \)
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)
### Proof by Cases and Induction

- Rule Induction
- Rule Inversion
Isar Induction Demo.thy

Rule induction


Rule induction

\textbf{inductive} \ I :: \ \tau \Rightarrow \sigma \Rightarrow \text{bool} \n
\textbf{where}

\text{rule}_1: \ldots

\vdots

\text{rule}_n: \ldots

\textbf{show} \ I \ x \ y \Rightarrow \ P \ x \ y

\textbf{proof} (\textit{induction rule: \ I.induct})

\textbf{case} \ \text{rule}_1

\vdots

\textbf{show} \ ?\text{case}

\textbf{next}

\vdots

\textbf{next}

\textbf{case} \ \text{rule}_n

\vdots

\textbf{show} \ ?\text{case}

\textbf{qed}
Fixing your own variable names

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

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

In a proof of

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

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 \( A_i \)

\[ R \quad R.IH + R.hyps + R.prems \]
Proof by Cases and Induction

Rule Induction

Rule Inversion
Rule inversion

**inductive** \( ev :: nat \Rightarrow bool \) **where**

- \( ev0: \ ev \ 0 \ |
- \( evSS: \ ev \ n \ \Rightarrow \ ev(Suc(Suc \ n)) \)

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

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

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

Rule inversion
Rule inversion template

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

Impossible cases disappear automatically