Concrete Semantics
A Proof Assistant Approach

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2014-1-26
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:

\[ A \implies B \implies C \] means \[ A \implies (B \implies C) \]

Similarly for other arrows: \[ \Rightarrow, \implies \]

\[ \frac{A_1 \ldots A_n}{B} \] means \[ A_1 \implies \ldots \implies A_n \implies B \]
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 \texttt{bool}, \texttt{nat} and \texttt{list}

Summary
Types

Basic syntax:

\[ \tau ::= (\tau) \]
\[ \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \] base types
\[ \mid 'a \mid 'b \mid \ldots \] type variables
\[ \mid \tau \Rightarrow \tau \] functions
\[ \mid \tau \times \tau \] pairs (ascii: *)
\[ \mid \tau \text{ list} \] lists
\[ \mid \tau \text{ set} \] sets
\[ \mid \ldots \] 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: \( \sin \; \pi \), \( \text{plus} \; x \; y \)

- **Function abstraction:**
  \[ \lambda x. \; t \]
  is the function with parameter \( x \) and result \( t \),
  i.e. \( \text{“} x \; \mapsto \; t \text{”} \).
  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 \( \tau \)”.

\[
\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2}
\]
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**: +, −, *, #, @, . . .
- **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:

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

Syntax: 

theory \textit{MyTh} 

imports \textit{ImpTh_1} \ldots \textit{ImpTh_n} 

begin 

(definitions, theorems, proofs, \ldots)^* 

end 

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

\textit{ImpTh_i}: name of imported theories. Import transitive. 

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

Summary
**Type** bool

```plaintext
datatype bool = True | False

Predefined functions:
∧, ∨, →, ... :: bool ⇒ bool ⇒ bool
```

A *formula* is a term of type `bool`

if-and-only-if: =
**Type nat**

```plaintext
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,... :: ′a,    + :: ′a ⇒ ′a ⇒ ′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 \( \text{add} \).

- 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:
  \[
  \text{add } (\text{Suc } m) \ 0 = \text{Suc } (\text{add } m \ 0) \quad \text{by def. of } \text{add}
  \]
  \[
  = \text{Suc } m \quad \text{by IH}
  \]
Type 'a list

Lists of elements of type 'a

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

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, ..., x_n] = x_1 # ... # 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 $P(x\#xs)$.

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

\[
P(xs)\]

28
List_Demo.thy
An informal proof

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

**Proof** by induction on \( xs \).

- **Case Nil:** \( \text{app} \ (\text{app} \ Nil \ ys) \ zs = \text{app} \ ys \ zs = \text{app} \ Nil \ (\text{app} \ ys \ zs) \) holds by definition of \( \text{app} \).

- **Case Cons \( x \ xs \):** We assume \( \text{app} \ (\text{app} \ xs \ ys) \ zs = \text{app} \ xs \ (\text{app} \ ys \ zs) \) (IH), and we need to show
  \[\text{app} \ (\text{app} \ (\text{Cons} \ x \ xs) \ ys) \ zs = \text{app} \ (\text{Cons} \ x \ xs) \ (\text{app} \ ys \ zs).\]

  The proof is as follows:

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

Included in Main.

Don’t reinvent, reuse!

Predefined: \( xs \odot ys \) (append), \( length \), and \( map \):

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

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

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

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

Types and terms

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!
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. $\land x_1 \ldots x_p. \ A \implies B$

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

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

abbreviates

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

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

2 Type and function definitions

3 Induction 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 × 'b list`

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

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

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

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

Datatype values can be taken apart with `case`:

\[(case \textit{xs} of \; [\;] \Rightarrow \ldots \; | \; \textit{y} \# \textit{ys} \Rightarrow \ldots \; \textit{y} \ldots \; \textit{ys} \ldots)\]

Wildcards: 

\[(case \textit{m} of \; 0 \Rightarrow \textit{Suc} \; 0 \; | \; \textit{Suc} \; _{\text{-}} \Rightarrow 0)\]

Nested patterns:

\[(case \textit{xs} of \; [0] \Rightarrow 0 \; | \; [\textit{Suc} \; \textit{n}] \Rightarrow \textit{n} \; | \; _{\text{-}} \Rightarrow 2)\]

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:

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

No pattern matching, just \quad 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:
• (Suc m, 0) > (m, Suc 0)
• (Suc m, Suc n) > (Suc m, n)
• (Suc m, Suc n) > (m, _)

51
A restrictive version of \texttt{fun}

Means \textit{primitive recursive}

Most functions are primitive recursive

Frequently found in Isabelle theories

The essence of primitive recursion:

\[
\begin{align*}
f(0) &= \ldots \\
f(Suc\ n) &= \ldots f(n) \ldots \\
g([]) &= \ldots \\
g(x\#xs) &= \ldots g(xs) \ldots
\end{align*}
\]
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:

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

A tail recursive version:

```ml
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**

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

\(\rightsquigarrow\) induction rule \(\text{div2}.\text{induct}:\)

\[
\begin{array}{c}
P(0) \quad P(\text{Suc} \ 0) \quad \land \ n. \ P(n) \implies P(\text{Suc}(\text{Suc} \ n)) \\
\hline
P(m)
\end{array}
\]
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 \texttt{f.induct}

If \( f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau' : \)

\begin{center}
\textit{(induction} \ a_1 \ldots \ a_n \text{ rule: f.induct)}
\end{center}

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 $\rightsquigarrow$ simplification rule

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

Equations:

\[ 0 + n = n \] (1)

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

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

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

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \] (1) \[ \equiv \]

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \] (2) \[ \equiv \]

\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \] (3) \[ \equiv \]

\[ 0 \leq 0 + x \] (4) \[ \equiv \]

\[ \text{True} \]
Conditional rewriting

Simplification rules can be conditional:

\[
\begin{bmatrix}
P_1; \ldots; P_k
\end{bmatrix} \implies l = r
\]

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

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

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

\[
n < m \implies (n < \text{Suc} \ m) = \text{True} \quad \text{YES}
\]

\[
\text{Suc} \ n < m \implies (n < m) = \text{True} \quad \text{NO}
\]
Proof method *simp*

**Goal:** 1. \[ [ P_1; \ldots; P_m ] \rightarrow C \]

**apply(\(simp \ add:: eq_1 \ldots eq_n\))**

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

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

**Variations:**

- \((simp \ldots del:: \ldots)\) removes *simp*-lemmas
- *add* and *del* are optional
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:
  \[
  \text{(auto simp add: \ldots simp del: \ldots)}
  \]
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 \textit{simp}

Automatic:

\[
P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))
\]

By hand:

\[
P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \ e = \text{Suc } n \rightarrow P(b))
\]

Proof method: \textit{(simp split: nat.split)}

Or \textit{auto}. Similar for any datatype \textit{t}: \textit{t.split}
Simp_Demo.thy
Chapter 3

Case Study: IMP Expressions
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.
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

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

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**

\[ aexp = N \text{ int} \mid V \ vname \mid \text{Plus} \ aexp \ aexp \]

<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:

\[
\begin{align*}
\text{type\_synonym} \quad \text{val} &= \text{int} \\
\text{type\_synonym} \quad \text{state} &= \text{vname} \Rightarrow \text{val}
\end{align*}
\]
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
5 Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
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.
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):

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

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: \(\leftrightarrow\) (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:

\(\forall\)  \(\langle\text{forall}\rangle\)  ALL
\(\exists\)  \(\langle\text{exists}\rangle\)  EX
\(\lambda\)  \(\langle\text{lambda}\rangle\)  %
\(\rightarrow\)  -->
\(\leftrightarrow\)  <--->
\(\land\)  \(\&\)
\(\lor\)  \(\mid\)
\(~\)  \(\langle\text{not}\rangle\)  ~
\(\neq\)  \(\langle\text{noteq}\rangle\)  ^=
Sets over type `'a`

`'a set`

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

\[\begin{align*}
\in & \ \ \langle \text{in} \rangle & : \\
\subseteq & \ \ \langle \text{subseteq} \rangle & \leq \\
\cup & \ \ \langle \text{union} \rangle & \text{Un} \\
\cap & \ \ \langle \text{inter} \rangle & \text{Int}
\end{align*}\]
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
• 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

\texttt{arith}:

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

- Goal & filtered library
- External ATPs

Characteristics:
- Sometimes it works,
- Sometimes it doesn't.

Do you feel lucky?

---

1 Automatic 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 \( \text{conjI} \): \[
\begin{align*}
[?P; ?Q] & \implies ?P \land ?Q
\end{align*}
\]

These ?-variables can later be instantiated:

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

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

Example: rule:  \[ [?P; ?Q] \Rightarrow ?P \land ?Q \]

subgoal:  1. \ldots \Rightarrow A \land B

Result:  1. \ldots \Rightarrow A

2. \ldots \Rightarrow B

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

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

apply(\textit{rule xyz})

“Backchaining”
Typical backwards rules

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

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

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

They are known as introduction rules because they introduce a particular connective.
Teaching \textit{blast} new intro rules

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

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

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

Example:

\begin{verbatim}
theorem \texttt{trans}: [ ?x \leq ?y; ?y \leq ?z ] \Rightarrow ?x \leq ?z

goal 1. [ a \leq b; b \leq c; c \leq d ] \Rightarrow a \leq d

proof \texttt{apply}(\text{blast intro: } \texttt{trans})
\end{verbatim}

Can greatly increase the search space!
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: \( ?t = ?t \)

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

\[
\therefore
\]

\[
?Q \implies a = a \land ?Q
\]
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: \texttt{theorem refl}: $?t = ?t$

\begin{verbatim}
conjI[OF refl[of "a"] refl[of "b"]]
\end{verbatim}

$$\leadsto$$

$$a = a \land b = b$$
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:

\begin{verbatim}
inductive ev :: nat ⇒ bool
where
  ev 0  | ev n ⇒ ev (n + 2)
\end{verbatim}
An easy proof: $ev 4$

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

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

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

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

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

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

- rule \( ev \ n \implies ev \ (n+2) \)
  \( \implies m = n+2 \) and \( even \ n \) (IH)
  \( \implies even \ m = even \ (n+2) = even \ n = True \)
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:

$$ev \ n \ \ P \ 0 \ \ \bigwedge n. \ [ ev \ n; \ P \ n ] \ \Longrightarrow \ P(n+2)$$
Format of inductive definitions

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

$\left[ I \ a_1; \ldots; I \ a_n \right] \Rightarrow I \ a \ |

\vdots

Note:

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

To prove

\[ I \ x \ \implies P \ x \]

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

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

```plaintext
[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
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
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  (\forall)
  | assume prop        (\equiv)
  | [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 \( a: \text{surj } f \)
  from \( a \) have \( b: \forall A. \exists a. A = f a \)
    by (simp add: surj_def)
  from \( b \) have \( c: \exists a. \{x. x \notin f x\} = f a \)
    by blast
  from \( c \) show False
    by blast
qed
Isar_Demo.thy

Cantor and abbreviations
Abbreviations

\( this \) = the previous proposition proved or assumed
\( then \) = from \( this \)
\( thus \) = then show
\( hence \) = then have
using and with

(have|show) prop using facts
  =
from facts (have|show) prop

with facts
  =
from facts this
Structured lemma statement

lemma
  fixes $f :: 'a \Rightarrow 'a \text{ set}$
  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

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

- fixes and assumes sections optional
- shows optional if no fixes and assumes
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 \( R \) . . .
next
  assume \( \neg P \)
  : 
  show \( R \) . . .
qed

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

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

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

proof

  assume $P$

  : 

  show $Q$ \ldots

next

  assume $Q$

  : 

  : 

  show $P$ \ldots

qed
∀ and ∃ introduction

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

show \( \exists x. \ P(x) \)
proof
  \begin{align*}
  & \text{show} \ P(witness) \  . . . \\
  \end{align*}
qed
∃ elimination: obtain

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

: \( x \) fixed local variable

Works for one or more \( x \)
lemma ⋾ \textit{surj}(\texttt{f :: 'a ⇒ 'a set})

proof
  assume \textit{surj} \texttt{f}
  hence \( \exists \texttt{a}. \{ \texttt{x}. \texttt{x} \notin \texttt{f x} \} = \texttt{f a} \)
  by (auto simp: \texttt{surj_def})

then obtain \texttt{a} where \( \{ \texttt{x}. \texttt{x} \notin \texttt{f x} \} = \texttt{f a} \)
by blast

hence \texttt{a} \notin \texttt{f a} \iff \texttt{a} \in \texttt{f a}
by blast

thus \texttt{False} by blast

qed
Set equality and subset

show \( A = B \)
proof
  show \( A \subseteq B \) \ldots
next
  show \( B \subseteq A \) \ldots
qed

show \( A \subseteq B \)
proof
  fix \( x \)
  assume \( x \in A \)
  \vdots
  show \( x \in B \) \ldots
qed
Isar_Demo.thy

Exercise
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks
Example: pattern matching

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

\textbf{proof}

\hspace{1em} \textbf{assume } ?L
\hspace{1em} \vdots
\hspace{1em} \textbf{show } ?R \ldots

\textbf{next}

\hspace{1em} \textbf{assume } ?R
\hspace{1em} \vdots
\hspace{1em} \textbf{show } ?L \ldots

\textbf{qed}
show formula (is thesis)
proof -
  :
  : show thesis ...
qed

Every show implicitly defines thesis
Introducing local abbreviations in proofs:

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

By name:

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

By value:

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

↑  ↑

back quotes
Isar_Demo.thy

Pattern matching and quotation
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover and raw proof blocks
Example

lemma

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

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
- **apply** *auto*
- **apply** . . .

At the end:

- **done**
- Better: **convert to structured proof**
Streamlining Proofs
Pattern Matching and Quotations
Top down proof development
moreover and raw proof blocks
moreover—ultimately

have $P_1 \ldots$
mmoreover
have $P_2 \ldots$
mmoreover
:  
mmoreover
have $P_n \ldots$
ultimately
have $P \ldots$

have $lab_1$: $P_1 \ldots$
have $lab_2$: $P_2 \ldots$
: 
have $lab_n$: $P_n \ldots$
from $lab_1$ $lab_2$ \ldots
have $P \ldots$

With names
Raw proof blocks

{ fix $x_1 \ldots x_n$
  assume $A_1 \ldots A_m$
  ;
  have $B$
}

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

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):

\[ \bigwedge x_1 \ldots x_n \left[ A_1; \ldots ; A_m \right] \implies B \]

How to prove each subgoal:

\begin{verbatim}
  fix  x_1 \ldots x_n
  assume A_1 \ldots A_m
  : show  B
\end{verbatim}

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

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

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

fix \( x_1 \ldots x_k \)
assume \( C_i: \)

\( \begin{aligned}
\text{label} & \quad \text{term} = (C_i x_1 \ldots x_k) \\
\text{formula} & \\
\end{aligned} \)
Isar_Induction_Demo.thy

Structural induction for \textit{nat}
Structural induction for $nat$

show $P(n)$

proof (induction $n$)

  case 0  
  \vdots  
  show ?case

next

  case ($Suc\ n$)  
  \vdots  
  \vdots

  show ?case

qed
show \( A(n) \implies P(n) \)
proof (induction \( n \))
  case \( 0 \)
  :
  show \(?case\)
next
  case \((\text{Suc } n)\)
  :
  :
  show \(?case\)
qed

\begin{align*}
\equiv \quad & \text{assume } 0: A(0) \\
\quad & \text{let } ?case = P(0) \\
\equiv \quad & \text{fix } n \\
\quad & \text{assume } \text{Suc}: A(n) \implies P(n) \\
\quad & \hspace{1cm} A(\text{Suc } n) \\
\quad & \text{let } ?case = P(\text{Suc } n)
\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 \( A_i \)

\[ C \quad C.IH + C.prems \]
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
inductive \( I :: \tau \Rightarrow \sigma \Rightarrow \text{bool} \)
where
\textit{rule}_1: \ldots

\vdots

\textit{rule}_n: \ldots

\begin{align*}
\text{show } & I \ x \ y \Rightarrow \ P \ x \ y \\
\text{proof (induction rule: } & I \text{.induct)} \\
\text{case } & \text{rule}_1 \\
\ldots \\
\text{show } & \text{?case} \\
\text{next} \\
\vdots \\
\text{next} \\
\text{case } & \text{rule}_n \\
\ldots \\
\text{show } & \text{?case} \\
\text{qed}
\end{align*}
Fixing your own variable names

\begin{align*}
\textbf{case} \ (rul_e_i \ x_1 \ldots \ x_k) \\
\end{align*}

Renames the first $k$ variables in $rul_e_i$ (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

\textbf{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

\[
\text{inductive } \ ev :: \ \text{nat} \Rightarrow \ \text{bool} \ \text{where}
\]

\[
\begin{align*}
\text{ev0: } & \ ev \ 0 \ | \\
\text{evSS: } & \ ev \ n \ \Rightarrow \ ev(Suc(Suc \ n))
\end{align*}
\]

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

\[
\begin{align*}
ev \ n \ \Rightarrow \ n = 0 \ \lor \ (\exists \ k. \ n = Suc(Suc \ k) \ \land \ ev \ k)
\end{align*}
\]

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