Concrete Semantics
with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2020-10-8
Chapter 1

Introduction
1 Background

2 This Course
1 Background

2 This Course
Why Semantics?

Without semantics, we do not really know what our programs mean. We merely have a good intuition and a warm feeling. Like the state of mathematics in the 19th century — before set theory and logic entered the scene.
Intuition is important!

- You need a good intuition to get your work done efficiently.
- To understand the average accounting program, intuition suffices.
- To write a bug-free accounting program may require more than intuition!
- I assume you have the necessary intuition.
- This course is about “beyond intuition”. 
Intuition is not sufficient!

Writing correct language processors (e.g. compilers, refactoring tools, ...) requires

- a deep understanding of language semantics,
- the ability to *reason* (= perform proofs) about the language and your processor.

Example:
What does the correctness of a type checker even mean? How is it proved?
Why Semantics??

We have a compiler — that is the ultimate semantics!!

• A compiler gives each individual program a semantics.
• It does not help with reasoning about the PL or individual programs.
• Because compilers are far too complicated.
• They provide the worst possible semantics.
• Moreover: compilers may differ!
The sad facts of life

• Most languages have one or more compilers.
• Most compilers have bugs.
• Few languages have a (separate, abstract) semantics.
• If they do, it will be informal (English).
Bugs

• Google “compiler bug”

• Google “hostile applet”
  Early versions of Java had various security holes. Some of them had to do with an incorrect bytecode verifier.

GI Dissertationspreis 2003:
Gerwin Klein: Verified Java Bytecode Verification
Standard ML (SML)

First real language with a mathematical semantics:
Milner, Tofte, Harper:
The Definition of Standard ML. 1990.

Robin Milner (1934–2010)
Turing Award 1991.

Main achievements: LCF (theorem proving)
SML (functional programming)
CCS, pi (concurrency)
The sad fact of life

SML semantics hardly used:
- too difficult to read to answer simple questions quickly
- too much detail to allow reliable informal proof
- not processable beyond \LaTeX{}, not even executable
More sad facts of life

- Real programming languages are complex.
- Even if designed by academics, not industry.
- Complex designs are error-prone.
- Informal mathematical proofs of complex designs are also error-prone.
The solution

Machine-checked language semantics and proofs

- Semantics at least type-correct
- Maybe executable
- Proofs machine-checked

The tool:

Proof Assistant (PA)

or

Interactive Theorem Prover (ITP)
Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step
- Can prove hard and huge theorems

Government health warnings:

- Time consuming
- Potentially addictive
- Undermines your naive trust in informal proofs
Terminology

This lecture course:

Formal = machine-checked
Verification = formal correctness proof

Traditionally:

Formal = mathematical
Two landmark verifications

C compiler
Competitive with gcc –01

Xavier Leroy
INRIA Paris
using Coq

Operating system microkernel (L4)

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle
A happy fact of life

Programming language researchers are increasingly using PAs
Why verification pays off

Short term: \textit{The software works!}

Long term:

\begin{itemize}
  \item Tracking effects of changes by rerunning proofs
  \item Incremental changes of the software typically require only incremental changes of the proofs
\end{itemize}

Long term much more important than short term:

\textit{Software Never Dies}
1 Background

2 This Course
What this course is *not* about

- Hot or trendy PLs
- Comparison of PLs or PL paradigms
- Compilers (although they will be one application)
What this course is about

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs
- Description techniques: operational semantics
- Proof techniques: inductions

Both informally and formally (PA!)
Our PA: Isabelle/HOL

- Started 1986 by Paulson (U of Cambridge)
- Later development mainly by Nipkow & Co (TUM) and Wenzel
- The logic HOL is ordinary mathematics

Learning to use Isabelle/HOL is an integral part of the course

All exercises require the use of Isabelle/HOL
Why I am so passionate about the PA part

• It is the future
• It is the only way to deal with complex languages reliably
• I want students to learn how to write correct proofs
• I have seen too many proofs that look more like LSD trips than coherent mathematical arguments
Overview of course

- Introduction to Isabelle/HOL
- IMP (assignment and while loops) and its semantics
- A compiler for IMP
- Hoare logic for IMP
- Type systems for IMP
- Program analysis for IMP
The semantics part of the course is mostly traditional.

The use of a PA is leading edge.

A growing number of universities offer related course.
What you learn in this course goes far beyond PLs

It has applications in compilers, security, software engineering etc.

It is a new approach to informatics
Part I

Isabelle
Chapter 2

Programming and Proving
Overview of Isabelle/HOL

Type and function definitions

Induction Heuristics

Simplification
Notation

Implication associates to the right:

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

Similarly for other arrows: \[ \Rightarrow, \quad \rightarrow \]

\[ \frac{A_1 \ldots A_n}{B} \] means \[ A_1 \implies \cdots \implies A_n \implies B \]
Overview of Isabelle/HOL

Type and function definitions

Induction Heuristics

Simplification
HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:
- For the moment: only $term = term$
  e.g. $1 + 2 = 4$
- Later: $\land$, $\lor$, $\rightarrow$, $\forall$, ...
Overview of Isabelle/HOL

Types and terms

Interface

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

Summary
Types

Basic syntax:

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

- base types
- type variables
- functions
- pairs (ascii: \( \ast \))
- lists
- sets
- 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. “$x \mapsto t$”.

  Example: $\lambda x. \ \text{plus} \ x \ x$
Terms

Basic syntax:

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

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 \( \lambda \text{-calculus} \).
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: \( f(x :: \text{nat}) \)
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*

$$f \ a_1 \ \text{where} \ a_1 :: \tau_1$$
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[
! f \ x + \ y \ \equiv \ (f \ x) + \ y \ \not\equiv \ f (x + y) \]

Enclose if and case in parentheses:

\[
! (if \ _ \ then \ _ \ else \ _) \]

44
Theory = Isabelle Module

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

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

Usually: imports Main
Concrete syntax

In .thy files:
Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides
Overview of Isabelle/HOL

Types and terms

Interface

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

Summary
Type \texttt{bool}

\textbf{datatype} \quad \texttt{bool} = \texttt{True} \mid \texttt{False}

Predefined functions:
\(\land, \lor, \rightarrow, \ldots\) :: \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool}

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

\textit{if-and-only-if:} \quad =
Type $nat$

**datatype** $nat = 0 \mid Suc \; nat$

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

Predefined functions: $+$, $\ast$, $\ldots$ :: $nat \Rightarrow nat \Rightarrow nat$

Numbers and arithmetic operations are overloaded:

$0, 1, 2, \ldots$ :: $'a$, $+$ :: $'a \Rightarrow 'a \Rightarrow 'a$

You need type annotations: $1 :: nat$, $x + (y :: nat)$ unless the context is unambiguous: $Suc\; z$
Nat_Demo.thy
An informal proof

**Lemma** \( \text{add } m \ 0 = m \)

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

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

- **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:
  \[
  \text{add } (\text{Suc } m) \ 0 = \text{Suc } (\text{add } m \ 0) \quad \text{by def. of add} \\
  = \text{Suc } m \quad \text{by IH}
  \]
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")
• [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 \( P(x\#xs) \).

\[
\frac{\neg \exists x \; xs. \; P(xs) \quad \land \; P ([])}{P(xs)}
\]
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 $map$
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 name}: "..."
apply (...)
apply (...)

\textbf{done}

If the lemma is suitable as a simplification rule:

\textbf{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 \quad “and”
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification
Type and function definitions

Type definitions

Function definitions
Type synonyms

**type_synonym**  

\[ \text{name} = \tau \]

Introduces a *synonym* name for type \( \tau \)

**Examples**

**type_synonym**  

\[ \text{string} = \text{char list} \]

**type_synonym**  

\[ (\text{'a}, \text{'b})\text{foo} = \text{'a list} \times \text{'b list} \]

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 \quad = \quad C_1 \tau_{1,1} \cdots \tau_{1,n_1} \quad |
\quad \cdots \quad |
\quad 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:** \(C_i \ldots \neq C_j \ldots \text{ if } i \neq j\)
- **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:

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

Wildcards: 

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

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

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

If 'a has values \( a_1, a_2, \ldots \)  
then 'a option has values None, Some \( a_1 \), Some \( a_2 \), \ldots \)

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)
4 Type and function definitions

Type definitions

Function definitions
Non-recursive definitions

Example

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

\[
\text{fun } \text{sep} :: \ 'a \to \ 'a \ \text{list} \to \ 'a \ \text{list} \ \text{where}
\]

\[
\text{sep } a \ (x \# y \# zs) = x \# a \# \text{sep } a \ (y \# zs) \quad | \\
\text{sep } a \ xs = xs
\]
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:

\[
\begin{align*}
  f(0) & = \ldots \text{ no recursion} \\
  f(Suc \ n) & = \ldots f(n) \ldots \\
  g([],) & = \ldots \text{ no recursion} \\
  g(x\#xs) & = \ldots g(xs) \ldots 
\end{align*}
\]
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 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**

```
lemma itrev xs [] = rev xs
```
Induction_Demo.thy

Generalisation
Generalisation

• Replace constants by variables

• Generalize free variables
  • by \textit{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)

⇝ induction rule div2.induct:

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

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

\begin{center}
(induction $a_1 \ldots a_n$ rule: \textit{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
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 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:

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

Rewriting:

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

\text{True}
Conditional rewriting

Simplification rules can be conditional:

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

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 \textit{simp}-rules (almost) blindly from left to right.

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

Principle:

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

is suitable as a \textit{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 \textit{simp}

Goal: 1. \[
\left[ P_1; \ldots; P_m \right] \implies C
\]

\textbf{apply(}simp add: eq_1 \ldots eq_n\textbf{)}

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

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

Variations:

- \((simp \ldots del: \ldots)\) 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:
  
  \((auto\ simp\ add:\ \ldots\ simp\ del:\ \ldots)\)
Rewriting with definitions

Definitions \textbf{(definition)} must be used \textit{explicitly}:

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

\(f\) is the function whose definition is to be unfolded.
Case splitting with \textit{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: (\textit{simp split: nat.split})

Or \textit{auto}. Similar for any datatype \( t \): \textit{t.split}
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.
Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
Concrete and abstract syntax

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

Abstract syntax: trees, eg

Parser: function from strings to trees

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

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

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

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

We focus on abstract syntax which we introduce via datatypes.
## Datatype $aexp$

Variable names are strings, values are integers:

**type_synonym** $vname = string$

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

<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>$Plus (V &quot;x&quot;) (V &quot;y&quot;)$</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!

\[ N \; 0 \neq \; \textit{Plus} \; (N \; 0) \; (N \; 0) \]
The (program) state

What is the value of $x+1$?

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

```
type_synonym val = int
type_synonym state = vname => val
```
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
7 Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
BExp.thy
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
8 Logical Formulas
9 Proof Automation
10 Single Step Proofs
11 Inductive Definitions
8 Logical Formulas

9 Proof Automation

10 Single Step Proofs

11 Inductive Definitions
Syntax (in decreasing precedence):

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

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: \(\leftrightarrow\) (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 \( \lambda \).
Quantifiers have low precedence and need to be parenthesized (if in some context)

\[
\neg P \land \forall x. \ Q \ x \ \sim \ P \land (\forall x. \ Q \ x) \ \neg
\]
Mathematical symbols

... and their ascii representations:

\( \forall \)  \(<forall>\)  ALL
\( \exists \)  \(<exists>\)  EX
\( \lambda \)  \(<lambda>\)  %
\( \rightarrow \)  -->
\( \leftrightarrow \)  <-->
\( \wedge \)  \(/\)  &
\( \vee \)  \(/\)  |
\( \neg \)  \(<not>\)  ~
\( \neq \)  \(<noteq>\)  ~=
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 - A
- ...

\( \in \) \quad \text{\textbackslash <in>}: \\
\subseteq \quad \text{\textbackslash <subseteq> \text{\textbackslash subseteqeq} \leq} \\
\cup \quad \text{\textbackslash <union> Un} \\
\cap \quad \text{\textbackslash <inter> 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 \( t \)
8 Logical Formulas

9 Proof Automation

10 Single Step Proofs

11 Inductive Definitions
**simp and auto**

- **simp**: rewriting and a bit of arithmetic
- **auto**: rewriting and a bit of arithmetic, logic and sets

- Show you where they got stuck
- highly incomplete
- Extensible with new simp-rules

**Exception:** auto acts on all subgoals
fastforce

- rewriting, logic, sets, relations and a bit of arithmetic.
- incomplete but better than auto.
- Succeeds or fails
- Extensible with new simp-rules
blast

- A complete proof search procedure for FOL . . .
- . . . but (almost) without “=”
- Covers logic, sets and relations
- Succeeds or fails
- Extensible with new deduction rules
Automating arithmetic

**arith:**

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

Isabelle

Goal & filtered library

↓                      ↑

Proof

external ATPs

Characteristics:

• Sometimes it works,
• sometimes it doesn’t.

Do you feel lucky?

---

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

\(\approx\)

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

done
Auto_Proof_Demo.thy
8 Logical Formulas

9 Proof Automation

10 Single Step Proofs

11 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 \texttt{conjI}: \([?P; ?Q] \implies ?P \land ?Q\)

These ?-variables can later be instantiated:

- By hand:
  \[
  \text{conjI[of "a=b" "False"]} \quad \leadsto \quad \[a = b; \ False\] \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: $[ ?P; ?Q ] \implies ?P \land ?Q$
subgoal: 1. \ldots $\implies A \land B$

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

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

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

apply($rule \ xyz$)

“Backchaining”
Typical backwards rules

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

\[
\frac{?P \iff ?Q}{?P \rightarrow ?Q} \quad \text{impI}
\]

\[
\frac{\forall x. ?P x}{\wedge x. ?P x} \quad \text{allI}
\]

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

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

Example:

\begin{align*}
\text{theorem } \texttt{le\_trans: } & \quad [ \ ?x \leq \ ?y; \ ?y \leq \ ?z \ ] \implies \ ?x \leq \ ?z \\
\text{goal } \textit{1. } & \quad [ \ a \leq b; \ b \leq c; \ c \leq d \ ] \implies \ a \leq d \\
\text{proof } & \quad \texttt{apply(blast intro: le\_trans)}
\end{align*}

Also works for \textit{auto} and \textit{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: \( ?t = ?t \)

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

\[
    \rightsquigarrow
\]

\[
    ?Q \implies a = a \land ?Q
\]
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: $?t = ?t$

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

$$\implies$$

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

\( A_1 \land \ldots \land A_n \rightarrow A \)
8 Logical Formulas

9 Proof Automation

10 Single Step Proofs

11 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 \Rightarrow bool
where
  ev 0 | ev n \Rightarrow ev (n + 2)
```
An easy proof: \( ev \ 4 \)

\[
ev 0 \implies ev 2 \implies ev 4
\]
Consider

\textbf{fun} \textit{evn :: nat} \Rightarrow \textit{bool} \ \textbf{where}

\textit{evn 0} = \textit{True} \ |
\textit{evn (Suc 0)} = \textit{False} \ |
\textit{evn (Suc (Suc n))} = \textit{evn n}

A trickier proof: \textit{ev m} \implies \textit{evn m}

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

Two cases: \textit{ev m} is proved by

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

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

To prove

$$ev \ n \ \Rightarrow \ P \ n$$

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

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

Rule $ev.induct$:

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

\textbf{inductive} \ I :: \ \tau \Rightarrow \ bool \ \textbf{where} \\
\left[ \ I \ a_1; \ldots ; \ I \ a_n \ \right] \Rightarrow \ I \ a \\
::

\textbf{Note:}

- \( I \) may have multiple arguments.
- Each rule may also contain \textit{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

\[
\text{inductive_set } I :: \tau \text{ set where } \\
\left[ a_1 \in I; \ldots; a_n \in I \right] \implies a \in I
\]

Difference to \textbf{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
12. Isar by example

13. Proof patterns

14. Streamlining Proofs

15. 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 (\land)
  | assume prop (\iff)
  | [from fact^{+}] (have | show) prop proof

prop = [name:] "formula"

fact = name | ...
12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction
Example: Cantor’s theorem

lemma \neg \text{surj}(f :: \forall 'a. \exists 'a set)

proof
  default proof: assume \text{surj}, show 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 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

\[(\text{have} | \text{show}) \text{ prop } \text{using facts} = \]

\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 \( y :: \tau_2 \) . . .
\textbf{assumes} \( a :: P \) and \( b :: Q \) . . .
\textbf{shows} \( R \)

- \textbf{fixes} and \textbf{assumes} sections optional
- \textbf{shows} optional if no \textbf{fixes} and \textbf{assumes}
12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 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⟩
show \( \neg P \)
proof
  assume \( P \)
  
  show \( \text{False} \) ⟨proof⟩
qed

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

proof

assume $P$

\[ \vdots \]

show $Q \langle proof \rangle$

next

assume $Q$

\[ \vdots \]

\[ \vdots \]

show $P \langle proof \rangle$

qed
∀ and ∃ introduction

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

show \( \exists x. P(x) \)
proof
  :  
  :  
  :  show \( P(\text{witness}) \)  \( \langle \text{proof} \rangle \)
qed
\exists \text{ elimination: obtain}

\begin{align*}
\text{have } \exists x. \ P(x) \\
\text{then obtain } x \text{ where } p: P(x) \text{ by blast} \\
\vdash \ x \text{ fixed local variable}
\end{align*}

Works for one or more \( x \)
lemma ¬ surj(f :: 'a ⇒ 'a set)
proof
  assume surj f
  hence ∃ a. {x. x ∉ f x} = f a by (auto simp: surj_def)
  then obtain a where {x. x ∉ f x} = f a by blast
  hence a ∉ f a ⟷ a ∈ f a by blast
  thus False by blast
qed
Set equality and subset

\[ \text{show } A = B \]
\[ \text{proof} \]
\[ \text{show } A \subseteq B \langle \text{proof} \rangle \]
\[ \text{next} \]
\[ \text{show } B \subseteq A \langle \text{proof} \rangle \]
\[ \text{qed} \]

\[ \text{show } A \subseteq B \]
\[ \text{proof} \]
\[ \text{fix } x \]
\[ \text{assume } x \in A \]
\[ : \]
\[ \text{show } x \in B \langle \text{proof} \rangle \]
\[ \text{qed} \]
Isar_Demo.thy

Exercise
12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 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 formula \((is \ ?thesis)\)
proof -
  :
    show \(?thesis \langle proof\rangle\)
qed

Every show implicitly defines \(?thesis\)
Introducing local abbreviations in proofs:

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

By name:

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

By value:

```lisp
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**

\[ \exists y s \; z s. \; x s = y s \; @ \; z s \land \]

\[(\text{length } y s = \text{length } z s \lor \text{length } y s = \text{length } z s + 1)\]

**Proof** ???

189


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 $P_1$ ... 
morerover 
have $P_2$ ... 
morerover 
: 
morerover 
have $P_n$ ... 
ultimately 
have $P$ ... 

have $lab_1$: $P_1$ ... 
have $lab_2$: $P_2$ ... 
: 
have $lab_n$: $P_n$ ... 
from $lab_1$ $lab_2$ ... 
have $P$ ... 

With names
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
Local lemmas

\[ \text{have } B \text{ if name: } A_1 \ldots A_m \text{ for } x_1 \ldots x_n \]

\langle \text{proof} \rangle

proves \[ \lbrack A_1; \ldots; A_m \rbrack \implies B \]

where all \( x_i \) have been replaced by ?\( x_i \).
Proof state and Isar text

In general: proof **method**

Applies *method* and generates subgoal(s):

\[ \forall x_1 \ldots x_n. \[ \[ A_1; \ldots ; A_m \] \] \Rightarrow B \]

How to prove each subgoal:

fix \( x_1 \ldots x_n \)
assume \( A_1 \ldots A_m \)

: 

show \( B \)

Separated by **next**
12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction
Isar_Induction_Demo.thy

Proof by cases
Datatype case analysis

```verbatim
datatype t = C_1 \bar{\tau} | \ldots
```

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

where

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

```
fix \ x_1 \ldots \ x_k
assume \ C_i: \ label \quad term = (C_i \ x_1 \ldots \ x_k) \quad formula
```
Isar_Induction_Demo.thy

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

\begin{verbatim}
show \( P(n) \)
proof \((\text{induction } n)\)
  case 0
  :
  show \( ?\text{case} \)
next
  case \((\text{Suc } n)\)
  :
  show \( ?\text{case} \)
  let \( ?\text{case} = P(0) \)
  let \( ?\text{case} = P(\text{Suc } n) \)
qed
\end{verbatim}
Structural induction with $\implies$

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

proof (induction $n$)
  case 0
    : 
    show $\ ? case$
  next
  case (Suc $n$)
    : 
    : 
    show $\ ? case$

qed
```

$$\equiv$$
```
assume 0: $A(0)$
let $\ ? case = P(0)$

fix $n$
assume Suc: $A(n) \implies P(n)$
$A(Suc \ n)$
let $\ ? case = P(Suc \ n)$
```
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

\textit{C.IH} the induction hypotheses

\textit{C.prems} the premises \(A_i\)

\textbf{C} \textit{C.IH + C.prems}
A remark on style

- **case** \((\text{Suc } n) \ldots \text{show } ?\text{case}\)
  is easy to write and maintain
- **fix** \(n \text{ assume } \text{formula} \ldots \text{show } \text{formula}'\)
  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** $I :: \tau \Rightarrow \sigma \Rightarrow \text{bool}$

where

$\text{rule}_1$: ...

\[ ...
\]

$\text{rule}_n$: ...

\[ ...
\]

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

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

*case* $\text{rule}_1$

\[ ...
\]

**show** $?\text{case}$

next

\[ ...
\]

next

*case* $\text{rule}_n$

\[ ...
\]

**show** $?\text{case}$

qed
Fixing your own variable names

\textbf{case} \( (\textit{rule}_i \ x_1 \ldots \ x_k) \)

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

In a proof of

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

by rule induction on \( I \ldots \):

In the context of

\textbf{case} \( R \)

we have

\begin{align*}
\textit{R.IH} & \quad \text{the induction hypotheses} \\
\textit{R.hyps} & \quad \text{the assumptions of rule} \ R \\
\textit{R.prems} & \quad \text{the premises} \ A_i \\
R & \quad R.IH + R.hyps + R.prems
\end{align*}
Proof by Cases and Induction

Rule Induction

Rule Inversion
Rule inversion

\textbf{inductive} \ ev :: nat \Rightarrow \ bool \ where

\textit{ev0}: \ ev \ 0 \ |

\textit{evSS}: \ ev \ n \implies ev(Suc(Suc \ n))

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

\( ev \ n \implies n = 0 \lor (\exists k. \ n = Suc(Suc \ k) \land ev \ k) \)

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