Concrete Semantics

with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2020-10-8

Chapter 1

Introduction

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 -O1`

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: *The software works!*

Long term:

Tracking effects of changes by rerunning proofs

Incremental changes of the software
typically require only incremental changes of the proofs

Long term much more important than short term:

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

3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification

Notation

Implication associates to the right:

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

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

$$\frac{A_1 \quad \ldots \quad A_n}{B} \quad \text{means} \quad A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$$

3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification

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

HOL has

- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:

- For the moment: only $term = term$, e.g. $1 + 2 = 4$
- Later: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, ...

3 Overview of Isabelle/HOL

- Types and terms
- Interface
- By example: types $bool$, nat and $list$
- Summary

Types

Basic syntax:

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

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

Terms

Terms can be formed as follows:

- *Function application:* $f\ t$
  is the call of function f with argument t.
  If f has more arguments: $f\ t_1\ t_2\ \ldots$
  Examples: $sin\ \pi$, $plus\ x\ y$
- *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:

$$
\begin{array}{rcll}
t & ::= & (t) & \\
 & | & a & \text{constant or variable (identifier)} \\
 & | & t\ t & \text{function application} \\
 & | & \lambda x.\ t & \text{function abstraction} \\
 & | & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

Examples: $f\ (g\ x)\ y$

$h\ (\lambda x.\ f\ (g\ x))$

Convention: $f\ t_1\ t_2\ t_3 \equiv ((f\ t_1)\ t_2)\ t_3$

This language of terms is known as the λ-*calculus*.

The computation rule of the λ-calculus is the replacement of formal by actual parameters:

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

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

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

- The step from $(\lambda x.\ t)\ u$ to $t[u/x]$ is called *β-reduction*.
- Isabelle performs β-reduction automatically.

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation:
$t :: \tau$ means "t is a well-typed term of type τ".

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

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:

! $f\ x + y \;\equiv\; (f\ x) + y \;\not\equiv\; f\ (x + y)$ **!**

Enclose *if* and *case* in parentheses:

! (*if* _ *then* _ *else* _) **!**

Theory = Isabelle Module

Syntax: **theory** $MyTh$
imports $T_1 \ldots T_n$
begin
(definitions, theorems, proofs, ...)*
end

$MyTh$: name of theory. Must live in file $MyTh$`.thy`
T_i: names of *imported* theories. Import transitive.

Usually: **imports** `Main`

Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in "

Except for single identifiers

" normally not shown on slides

isabelle jedit

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

Overview_Demo.thy

3 Overview of Isabelle/HOL

Types and terms

Interface

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

Summary

Type $bool$

datatype $bool \; = \; True \; | \; False$

Predefined functions:
$\wedge, \vee, \longrightarrow, \ldots \; :: \; bool \Rightarrow bool \Rightarrow bool$

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

if-and-only-if: $=$

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

Predefined functions: $+,\ *,\ ...\ ::\ nat \Rightarrow nat \Rightarrow nat$

! Numbers and arithmetic operations are overloaded:
$0,1,2,...\ ::\ 'a, \qquad +\ ::\ \ 'a \Rightarrow 'a \Rightarrow 'a$

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

Nat_Demo.thy

An informal proof

Lemma $add\ m\ 0 = m$

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

Type $'a$ *list*

Lists of elements of type $'a$

datatype $'a\ list\ =\ Nil\ \mid\ Cons\ 'a\ \ ('a\ list)$

Some lists: Nil, $Cons\ 1\ Nil$, $Cons\ 1\ (Cons\ 2\ Nil)$, ...

Syntactic sugar:

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

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 $P(x\#xs)$.

$$\frac{P([]) \qquad \bigwedge x\ xs.\ P(xs) \Longrightarrow P(x\#xs)}{P(xs)}$$

List_Demo.thy

An informal proof

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$
Proof by induction on xs.

- Case Nil: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of app.
- Case $Cons\ x\ xs$: We assume $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$ (IH), and we need to show
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs =$
  $app\ (Cons\ x\ xs)\ (app\ ys\ zs)$.
  The proof is as follows:
  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
  $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of app
  $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
  $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of app

Large library: `HOL/List.thy`

Included in `Main`.

Don't reinvent, reuse!

Predefined: $xs \mathbin{@} ys$ (append), $length$, and map

- **datatype** defines (possibly) recursive data types.
- **fun** defines (possibly) recursive functions by pattern-matching over datatype constructors.

Proof methods

- $induction$ performs structural induction on some variable (if the type of the variable is a datatype).

- $auto$ solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

  "$=$" is used only from left to right!

Proofs

General schema:

lemma *name*: `"..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]:  "..."`

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

The proof state

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

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

Multiple assumptions

$$[\![\ A_1; \ \ldots \ ; \ A_n \ ]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ \ldots \ \Longrightarrow A_n \Longrightarrow B$$

$$; \quad \approx \quad \text{“and”}$$

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples

type_synonym $string = char\ list$

type_synonym $('a,'b)foo = 'a\ list \times\ 'b\ list$

Type synonyms are expanded after parsing
and are not present in internal representation and output

datatype — the general case

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)t \quad = \quad C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \quad | \quad \ldots \quad | \quad C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- *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:* $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i})$

Distinctness and injectivity are applied automatically

Induction must be applied explicitly

Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards: _

$$(\textit{case}\ m\ \textit{of}\ \ 0 \Rightarrow Suc\ 0\ \ |\ \ Suc\ \_ \Rightarrow 0)$$

Nested patterns:

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

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

The *option* type

datatype $'a\ option = None \mid Some\ 'a$

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 $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**
$lookup\ []\ x = None \mid$
$lookup\ ((a,\ b)\ \#\ ps)\ x =$
$\quad (\textit{if}\ a = x\ \textit{then}\ Some\ b\ \textit{else}\ lookup\ ps\ x)$

4 Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example

definition $sq :: nat \Rightarrow nat$ **where** $sq\ n\ =\ n{*}n$

No pattern matching, just $f\, x_1\, \dots\, x_n\ =\ \dots$

The danger of nontermination

How about $f\,x = f\,x + 1$?

! All functions in HOL must be total **!**

Key features of **fun**

- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema

Example: separation

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

Example: Ackermann

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**

$$
\begin{array}{lll}
ack\ 0\ n & = Suc\ n & | \\
ack\ (Suc\ m)\ 0 & = ack\ m\ (Suc\ 0) & | \\
ack\ (Suc\ m)\ (Suc\ n) & = ack\ m\ (ack\ (Suc\ m)\ n) &
\end{array}
$$

Terminates because the arguments decrease *lexicographically* with each recursive call:

- $(Suc\ m,\ 0) > (m,\ Suc\ 0)$
- $(Suc\ m,\ Suc\ n) > (Suc\ m,\ n)$
- $(Suc\ m,\ Suc\ n) > (m,\ \_)$

primrec

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

The essence of primitive recursion:

$$\begin{aligned} f(0) &= \dots && \text{no recursion} \\ f(Suc\ n) &= \dots f(n) \dots \end{aligned}$$

$$\begin{aligned} g([]) &= \dots && \text{no recursion} \\ g(x\#xs) &= \dots g(xs) \dots \end{aligned}$$

Basic induction heuristics

Theorems about recursive functions
are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

lemma $itrev\ xs\ [] = rev\ xs$

Induction_Demo.thy

Generalisation

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $arbitrary$ in induction proof
  - (or by universal quantifier in formula)

So far, all proofs were by structural induction
because all functions were primitive recursive.

In each induction step, 1 constructor is added.
In each recursive call, 1 constructor is removed.

Now: induction for complex recursion patterns.

Computation Induction

Example

fun $div2 :: nat \Rightarrow nat$ **where**
$div2\ 0 = 0 \ \ |$
$div2\ (Suc\ 0) = 0 \ \ |$
$div2\ (Suc(Suc\ n)) = Suc(div2\ n)$

$\leadsto$ induction rule `div2.induct`:

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

Computation Induction

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

for each defining equation

$$f(e) \;=\; \ldots f(r_1) \ldots f(r_k) \ldots$$

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

Induction follows course of (terminating!) computation
Motto: properties of f are best proved by rule $f.induct$

How to apply $f.induct$

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

$$(induction\ a_1\ \ldots\ a_n\ rule\colon f.induct)$$

Heuristic:

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

`Induction_Demo.thy`

Computation Induction

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 = (Term) Rewriting

An example

Equations:

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

Rewriting:

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Example

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

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

Termination

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

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

Principle:

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

is suitable as a $simp$-rule only
if l is "bigger" than r and each P_i

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

Proof method $simp$

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

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:
  $(auto\ simp\ add\colon\ \ldots\ \ simp\ del\colon\ \ldots)$

Rewriting with definitions

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

$$(simp\ add\colon f\_def \dots)$$

f is the function whose definition is to be unfolded.

Case splitting with $simp/auto$

Automatic:

$$P\ (\textit{if}\ A\ \textit{then}\ s\ \textit{else}\ t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

By hand:

$$P\ (\textit{case}\ e\ \textit{of}\ 0 \Rightarrow a \mid Suc\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall\, n.\ e = Suc\ n \longrightarrow P(b))$$

Proof method: $(simp\ split{:}\ nat.split)$
Or $auto$. Similar for any datatype t: $t.split$

Simp_Demo.thy

Chapter 3

Case Study: IMP Expressions

7 Case Study: IMP Expressions

7 Case Study: IMP Expressions

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

IMP *commands* are introduced later.

7 Case Study: IMP Expressions

Arithmetic Expressions

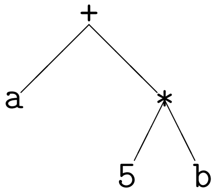
Boolean Expressions

Stack Machine and Compilation

Concrete and abstract syntax

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

Abstract syntax: trees, eg



Parser: function from strings to trees

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

Abstract syntax trees/terms are datatype values!

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

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

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 $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

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

Warning

This is syntax, not (yet) semantics!

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

The (program) state

What is the value of `x+1`?

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

  type_synonym $val = int$
  type_synonym $state = vname \Rightarrow val$

Function update notation

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

$$f(a := b)$$

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

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

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

BExp.thy

7 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

Syntax (in decreasing precedence):

$$
\begin{array}{lclcl}
\mathit{form} & ::= & (\mathit{form}) & \mid & \mathit{term} = \mathit{term} & \mid & \neg\mathit{form} \\
& \mid & \mathit{form} \wedge \mathit{form} & \mid & \mathit{form} \vee \mathit{form} & \mid & \mathit{form} \longrightarrow \mathit{form} \\
& \mid & \forall x.\ \mathit{form} & \mid & \exists x.\ \mathit{form}
\end{array}
$$

Examples:

$$
\begin{array}{rcl}
\neg\ A \wedge B \vee C & \equiv & ((\neg\ A) \wedge B) \vee C \\
s = t \wedge C & \equiv & (s = t) \wedge C \\
A \wedge B = B \wedge A & \equiv & A \wedge (B = B) \wedge A \\
\forall\, x.\ P\ x \wedge Q\ x & \equiv & \forall\, x.\ (P\ x \wedge Q\ x)
\end{array}
$$

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

Variable binding convention:

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

Similarly for $\exists$ and λ.

Warning

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

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

Mathematical symbols

... and their ascii representations:

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

Sets over type $'a$

$'a\ set$

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

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

Set comprehension

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

$simp$ 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:

$$\boxed{\textbf{Isabelle}}$$

Goal & filtered library $\downarrow$ $\uparrow$ Proof

$$\boxed{\text{external } \textbf{ATPs}^{1}}$$

Characteristics:

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

Do you feel lucky?

---

[1] Automatic Theorem Provers

by($proof\text{-}method$)

$$\approx$$

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

Auto_Proof_Demo.thy

Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.

What are these *?-variables* ?

After you have finished a proof, Isabelle turns all free variables V in the theorem into $?V$.

Example: theorem `conjI`: $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$
  $\llbracket a = b; False \rrbracket \Longrightarrow a = b \wedge False$
- By unification:
  unifying $?P \wedge ?Q$ with $a{=}b \wedge False$
  sets $?P$ to $a{=}b$ and $?Q$ to $False$.

Rule application

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

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

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

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

apply$(rule\ xyz)$

"Backchaining"

Typical backwards rules

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

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

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

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

Automating intro rules

If r is a theorem $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ then

$$(blast\ intro:\ r)$$

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

Example:

theorem le_trans: $[\![?x \leq ?y;\ ?y \leq ?z]\!] \Longrightarrow ?x \leq ?z$

goal 1. $[\![a \leq b;\ b \leq c;\ c \leq d]\!] \Longrightarrow a \leq d$

proof **apply**$(blast\ intro:\ le\_trans)$

Also works for $auto$ and $fastforce$

Can greatly increase the search space!

Forward proof: OF

If r is a theorem $A \Longrightarrow B$
and s is a theorem that unifies with A then

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

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

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

$$\rightsquigarrow$$

$$?Q \Longrightarrow a = a \wedge \,?Q$$

The general case:

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

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

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

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

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

$$\leadsto$$

$$a = a \wedge b = b$$

From now on: ? mostly suppressed on slides

Single_Step_Demo.thy

$\Longrightarrow$ versus $\longrightarrow$

$\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states: $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$

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

Phrase theorems like this	$[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$
not like this	$A_1 \wedge \ldots \wedge A_n \longrightarrow A$

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive $ev :: nat \Rightarrow bool$
where
$ev\ 0 \quad |$
$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: $ev\ 4$

$$ev\ 0 \Longrightarrow ev\ 2 \Longrightarrow ev\ 4$$

Consider

fun $evn :: nat \Rightarrow bool$ **where**
$evn\ 0 = True\ |$
$evn\ (Suc\ 0) = False\ |$
$evn\ (Suc\ (Suc\ n)) = evn\ n$

A trickier proof: $ev\ m \Longrightarrow evn\ m$

By induction on the *structure* of the derivation of $ev\ m$

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow evn\ m = True$
- rule $ev\ n \Longrightarrow ev\ (n{+}2)$
  $\Longrightarrow m = n{+}2$ and $evn\ n$ (IH)
  $\Longrightarrow evn\ m = evn\ (n{+}2) = evn\ n = True$

Rule induction for ev

To prove

$$ev\ n \Longrightarrow P\ n$$

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

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

Rule `ev.induct`:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ [\![\ ev\ n;\ P\ n\ ]\!] \Longrightarrow P(n+2)}{P\ n}$$

Format of inductive definitions

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

$\llbracket\ I\ a_1;\ \ldots\ ;\ I\ a_n\ \rrbracket \Longrightarrow I\ a \quad |$

$\vdots$

Note:

- I may have multiple arguments.
- Each rule may also contain *side conditions* not involving I.

Rule induction in general

To prove

$$I\ x \Longrightarrow P\ x$$

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

$$[\![\ I\ a_1;\ \dots\ ;\ I\ a_n\ ]\!] \Longrightarrow I\ a$$

that P is preserved:

$$[\![\ I\ a_1;\ P\ a_1;\ \dots\ ;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

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

Inductive_Demo.thy

Inductively defined sets

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

$\llbracket\ a_1 \in I;\ \ldots\ ;\ a_n \in I\ \rrbracket \Longrightarrow a \in I\ \ |$

$\vdots$

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 assertions

But: **apply** still useful for proof exploration

A typical Isar proof

proof
  assume $formula_0$
  have $formula_1$ **by** $simp$
  $\vdots$
  have $formula_n$ **by** $blast$
  show $formula_{n+1}$ **by** $\ldots$
qed

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

Isar core syntax

$$
\begin{array}{rcl}
\text{proof} & = & \textbf{proof}\ [\text{method}]\ \ \text{step}^*\ \ \textbf{qed} \\
& | & \textbf{by}\ \text{method}
\end{array}
$$

$$
\text{method} \ = \ (\mathit{simp}\ \ldots) \mid (\mathit{blast}\ \ldots) \mid (\mathit{induction}\ \ldots) \mid \ldots
$$

$$
\begin{array}{rcll}
\text{step} & = & \textbf{fix}\ \text{variables} & (\bigwedge) \\
& | & \textbf{assume}\ \text{prop} & (\Longrightarrow) \\
& | & [\textbf{from}\ \text{fact}^+]\ \ (\textbf{have} \mid \textbf{show})\ \text{prop}\ \ \text{proof} &
\end{array}
$$

$$
\text{prop} \ = \ [\text{name:}]\ \text{"formula"}
$$

$$
\text{fact} \ = \ \text{name} \mid \ldots
$$

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction

Example: Cantor's theorem

lemma $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof default proof: assume *surj*, show *False*
  assume a: $surj\ f$
  from a **have** b: $\forall\ A.\ \exists\ a.\ A = f\ a$
    by$(simp\ add\!: surj\_def)$
  from b **have** c: $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$
    by $blast$
  from c **show** $False$
    by $blast$
qed

Isar_Demo.thy

Cantor and abbreviations

Abbreviations

$$
\begin{array}{rcl}
\textit{this} & = & \text{the previous proposition proved or assumed} \\
\text{then} & = & \textbf{from } \textit{this} \\
\text{thus} & = & \textbf{then show} \\
\text{hence} & = & \textbf{then have}
\end{array}
$$

using and **with**

(**have**|**show**) prop **using** facts

=

from facts (**have**|**show**) prop

with facts

=

from facts *this*

Structured lemma statement

lemma
  fixes $f :: 'a \Rightarrow 'a\ set$
  assumes s: $surj\ f$
  shows $False$
proof $-$ no automatic proof step
  have $\exists\ a.\ \{x.\ x \notin f\ x\} = f\ a$ **using** s
    by$(auto\ simp\text{: } surj\_def)$
  thus $False$ **by** $blast$
qed

Proves $surj\ f \Longrightarrow False$
but $surj\ f$ *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** $y :: \tau_2 \ \ldots$
assumes *a:* P **and** *b:* $Q \ \ldots$
shows R

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

Case distinction

show R
proof *cases*
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume $\neg$ P
  $\vdots$
  show R $\langle proof \rangle$
qed

have $P \vee Q$ $\langle proof \rangle$
then show R
proof
  assume P
  $\vdots$
  show R $\langle proof \rangle$
next
  assume Q
  $\vdots$
  show R $\langle proof \rangle$
qed

Contradiction

show $\neg\ P$
proof
  assume P
  $\vdots$
  show $False\ \langle proof\rangle$
qed

show P
proof $(rule\ ccontr)$
  assume $\neg P$
  $\vdots$
  show $False\ \langle proof\rangle$
qed

$\longleftrightarrow$

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
  show Q $\langle proof \rangle$
next
  assume Q
  $\vdots$
  show P $\langle proof \rangle$
qed

$\forall$ and $\exists$ 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
  $\vdots$
  show $P(witness)$ $\langle proof \rangle$
qed

$\exists$ elimination: **obtain**

have $\exists\, x.\ P(x)$
then obtain x **where** *p:* $P(x)$ **by** *blast*
$\vdots$ $\quad x$ fixed local variable

Works for one or more x

obtain example

lemma $\neg\ surj(f :: 'a \Rightarrow 'a\ set)$
proof
  assume $surj\ f$
  hence $\exists\, a.\ \{x.\ x \notin f\ x\} = f\ a$ **by**$(auto\ simp\text{:}\ surj\_def)$
  then obtain a **where** $\{x.\ x \notin f\ x\} = f\ a$ **by** $blast$
  hence $a \notin f\ a \longleftrightarrow a \in f\ a$ **by** $blast$
  thus $False$ **by** $blast$
qed

Set equality and subset

show $A = B$
proof
  show $A \subseteq B$ $\langle proof \rangle$
next
  show $B \subseteq A$ $\langle proof \rangle$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\langle proof \rangle$
qed

Isar_Demo.thy

Exercise

Example: pattern matching

show $formula_1 \longleftrightarrow formula_2$ (**is** $?L \longleftrightarrow ?R$)
proof
  assume $?L$
  $\vdots$
  show $?R$ $\langle proof \rangle$
next
  assume $?R$
  $\vdots$
  show $?L$ $\langle proof \rangle$
qed

$?thesis$

show $formula$ (is $?thesis$)
proof -
$\vdots$
show $?thesis$ $\langle proof \rangle$
qed

Every show implicitly defines $?thesis$

let

Introducing local abbreviations in proofs:

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

$\vdots$

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

Quoting facts by value

By name:

have *x0:* $''x > 0''$ $\ldots$
$\vdots$
from *x0* $\ldots$

By value:

have $''x > 0''$ $\ldots$
$\vdots$
from `x>0` $\ldots$
$\uparrow$ $\uparrow$
back quotes

Isar_Demo.thy

Pattern matching and quotations

14 Streamlining Proofs

Example

lemma

$$\exists\, ys\ zs.\ xs = ys\ @\ zs \land (length\ ys = length\ zs \lor length\ ys = length\ zs + 1)$$

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* or whatever
apply ...

At the end:

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

14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas

moreover—ultimately

have $P_1 \quad \ldots$
moreover
have $P_2 \quad \ldots$
moreover
$\vdots$
moreover
have $P_n \quad \ldots$
ultimately
have $P \quad \ldots$

$\approx$

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

With names

14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas

Local lemmas

have B **if** *name:* $A_1 \ \dots \ A_m$ **for** $x_1 \ \dots \ x_n$
$\langle proof \rangle$

proves $[\![\ A_1; \ \dots \ ; A_m \ ]\!] \Longrightarrow B$
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):

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

How to prove each subgoal:

fix $x_1 \ \ldots \ x_n$
assume $A_1 \ \ldots \ A_m$
$\vdots$
show B

Separated by **next**

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
$\vdots$
qed

where **case** $(C_i \ x_1 \ \ldots \ x_k) \quad \equiv$

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

Isar_Induction_Demo.thy

Structural induction for nat

Structural induction for nat

show $P(n)$
proof $(induction\ n)$
    case 0 $\qquad\equiv\qquad$ **let** $?case = P(0)$
    $\vdots$
    show $?case$
next
    case $(Suc\ n)$ $\qquad\equiv\qquad$ **fix** n **assume** Suc: $P(n)$
    $\vdots$ $\qquad\qquad$ **let** $?case = P(Suc\ n)$
    $\vdots$
    show $?case$
qed

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$
proof $(induction\ n)$
  case 0 $\equiv$ **assume** 0: $A(0)$
  $\vdots$ **let** $?case = P(0)$
  show $?case$
next
  case $(Suc\ n)$ $\equiv$ **fix** n
  $\vdots$ **assume** Suc: $A(n) \Longrightarrow P(n)$
  $A(Suc\ n)$
  $\vdots$ **let** $?case = P(Suc\ n)$
  show $?case$
qed

Named assumptions

In a proof of

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by structural induction:

In the context of

case C

we have

$C.IH$	the induction hypotheses
$C.prems$	the premises A_i
C	$C.IH + C.prems$

A remark on style

- **case** $(Suc\ n)$ ... **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. $?case$)

15 Proof by Cases and Induction

Rule Induction

Rule Inversion

Isar_Induction_Demo.thy

Rule induction

Rule induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: ...
$\vdots$
$rule_n$: ...

show $I\ x\ y \Longrightarrow P\ x\ y$
proof (*induction rule: I.induct*)
  case $rule_1$
  ...
  show *?case*
next
$\vdots$
next
  case $rule_n$
  ...
  show *?case*
qed

Fixing your own variable names

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

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

Named assumptions

In a proof of

$$I \ldots \Longrightarrow A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by rule induction on $I \ldots$:

In the context of

case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

15 Proof by Cases and Induction

Rule Induction

Rule Inversion

Rule inversion

inductive $ev :: nat \Rightarrow bool$ **where**
$ev0$: $ev\ 0$ |
$evSS$: $ev\ n \Longrightarrow ev(Suc(Suc\ n))$

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

$$ev\ n \Longrightarrow n = 0 \lor (\exists\, k.\ n = Suc\ (Suc\ k) \land ev\ k)$$

Rule inversion = case distinction over rules

`Isar_Induction_Demo.thy`

Rule inversion

Rule inversion template

from $`ev\ n`$ **have** P
proof $cases$
  case $ev0$ $\qquad\qquad n = 0$
  $\vdots$
  show $?thesis\ \ \ldots$
next
  case $(evSS\ k)$ $\qquad\qquad n = Suc\ (Suc\ k),\ ev\ k$
  $\vdots$
  show $?thesis\ \ \ldots$
qed

Impossible cases disappear automatically