Concrete Semantics
with Isabelle/HOL

Peter Lammich
(slides from Concrete Semantics by Nipkow)

2018-10-16
Chapter 1

Introduction
1 Background

2 This Course
1 Background

2 This Course
Organization Issues

Course Homepage: http://www21.in.tum.de/teaching/semantik/WS1819/
Organization Issues

Course Homepage: http://www21.in.tum.de/teaching/semantik/WS1819/

Book: Nipkow, Klein: Concrete Semantics
http://concrete-semantics.org/
Organization Issues

Course Homepage: http://www21.in.tum.de/teaching/semantik/WS1819/

Book: Nipkow, Klein: Concrete Semantics
http://concrete-semantics.org/

Homework: IMPORTANT! 40% of final grade
Organization Issues

Course Homepage: http://www21.in.tum.de/teaching/semantik/WS1819/

Book: Nipkow, Klein: Concrete Semantics
http://concrete-semantics.org/

Homework: IMPORTANT! 40% of final grade

Tutorials and Homework are the heart and soul of this course!
Why Semantics?

Without semantics, we do not really know what our programs mean.
Why Semantics?

Without semantics, we do not really know what our programs mean.

We merely have a good intuition and a warm feeling.
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
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 important!

• You need a good intuition to get your work done efficiently.
Intuition is important!

- You need a good intuition to get your work done efficiently.
- To understand the average accounting program, intuition suffices.
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!
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.
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?
Intuition is not sufficient!

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

- a deep understanding of language semantics,
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.
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?
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!!
Why Semantics??

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

- A compiler gives each individual program a semantics.
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.
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.
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.
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).
The sad facts of life

- Most languages have one or more compilers.
The sad facts of life

- Most languages have one or more compilers.
- Most compilers have bugs.
The sad facts of life

- Most languages have one or more compilers.
- Most compilers have bugs.
- Few languages have a (separate, abstract) semantics.
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”
Bugs

- Google “compiler bug”
- Google “hostile applet”
  Early versions of Java had various security holes.
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.
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.
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.
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:
The sad fact of life

SML semantics hardly used:
  - too difficult to read to answer simple questions quickly
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
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.
More sad facts of life

- Real programming languages are complex.
More sad facts of life

- Real programming languages are complex.
- Even if designed by academics, not industry.
More sad facts of life

- Real programming languages *are* complex.
- Even if designed by academics, not industry.
- Complex designs are error-prone.
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
The solution

Machine-checked language semantics and proofs

- Semantics at least type-correct
The solution

Machine-checked language semantics and proofs

- Semantics at least type-correct
- Maybe executable
The solution

Machine-checked language semantics and proofs

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

- You give the structure of the proof
Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step
Proof Assistants

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

This lecture course:

Formal = machine-checked
Verification = formal correctness proof

Traditionally:

Formal = mathematical
Two landmark verifications

C compiler
Two landmark verifications

C compiler
Competitive with gcc -01
Two landmark verifications

C compiler
Competitive with gcc -01

Xavier Leroy
INRIA Paris
using Coq
Two landmark verifications

C compiler
Competitive with gcc -01

Operating system
microkernel (L4)

Xavier Leroy
INRIA Paris
using Coq
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!
Why verification pays off

Short term:  *The software works!*

Long term:  

Tracking effects of changes by rerunning proofs
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
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:
Why verification pays off

Short term: \textit{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
What this course is not about

- Hot or trendy PLs
- Comparison of PLs or PL paradigms
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
What this course *is* about

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs
- Description techniques: *operational semantics*
What this course is about

- Techniques for the description and analysis of
  - PLs
  - PL tools
  - Programs
- Description techniques: *operational semantics*
- Proof techniques: *inductions*
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)
Our PA: Isabelle/HOL

- Started 1986 by Paulson (U of Cambridge)
- Later development mainly by Nipkow & Co (TUM) and Wenzel
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
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
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
Why I am so passionate about the PA part

- It is the future
Why I am so passionate about the PA part

- It is the future
- It is the only way to deal with complex languages reliably
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
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
Overview of course

- Introduction to Isabelle/HOL
- IMP (assignment and while loops) and its semantics
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 semantics part of the course is mostly traditional.

The use of a PA is leading edge.
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
What you learn in this course goes far beyond PLs

It has applications in compilers, security, software engineering etc.
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
At the end of the course . . .
At the end of the course . . .
Part I

Isabelle
Chapter 2

Programming and Proving
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification
Quiz

Which of the following formulas have the same meaning?

1. $A \implies (B \implies C)$
2. $(A \implies B) \implies C$
3. $(A \land B) \implies C$
Implication associates to the right:

\[ A \implies B \implies C \text{ means } A \implies (B \implies C) \]
Implication associates to the right:

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

Similarly for other arrows: \( \Rightarrow \), \( \rightarrow \)
Implication associates to the right:

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

Similarly for other arrows: \( \implies, \quad \quad \)

\[ \frac{A_1 \ldots A_n}{B} \] means \[ A_1 \implies \cdots \implies A_n \implies B \]
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: ∧, ∨, −→, ∀, . . .
HOL = Higher-Order Logic
HOL = Functional Programming + Logic
HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators
HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!
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 = 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 $term = term$
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. \( 1 + 2 = 4 \)
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: $\land$, $\lor$, $\rightarrow$, $\forall$, ...
Overview of Isabelle/HOL

Types and terms

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

Summary
Basic syntax:

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

Basic syntax:

$$\tau ::= (\tau)$$
Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \] base types
Types

Basic syntax:

$$\tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \quad \text{base types}$$
$$\quad \mid 'a \mid 'b \mid \ldots \quad \text{type variables}$$
Basic syntax:

$$
\tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \quad \text{base types} \\
\mid 'a \mid 'b \mid \ldots \quad \text{type variables} \\
\mid \tau \Rightarrow \tau \quad \text{functions}
$$
Types

Basic syntax:

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

| \tau \Rightarrow \tau |
| \tau \times \tau |

\tau ::=

| bool | nat | int | ... |
| 'a | 'b | ... |

base types

type variables

functions

pairs (ascii: *)
Types

Basic syntax:

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

| bool | nat | int | ... | base types |
| 'a | 'b | ... | type variables |
| \tau \Rightarrow \tau | functions |
| \tau \times \tau | \tau list |
| \tau list | |

pairs (ascii: *)
lists
### Types

#### Basic syntax:

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

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

#### Basic syntax:

\[
\tau ::= (\tau) \\
| \text{bool} \ |
| \text{nat} \ |
| \text{int} \ |
| \ldots \\
| \text{'}a\text{'} \ |
| \text{'}b\text{'} \ |
| \ldots \\
| \tau \Rightarrow \tau \\
| \tau \times \tau \\
| \tau \text{ list} \\
| \tau \text{ set} \\
| \ldots
\]

<table>
<thead>
<tr>
<th>base types</th>
<th>type variables</th>
<th>functions</th>
<th>pairs (ascii: *)</th>
<th>lists</th>
<th>sets</th>
<th>user-defined types</th>
</tr>
</thead>
</table>

#### 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

Terms can be formed as follows:

- *Function application*: \( f \, t \)

Examples:
- \( \sin \pi \)
- \( \text{plus} \, x \, y \)
Terms

Terms can be formed as follows:

- **Function application**: \( f t \)
  is the call of function \( f \) with argument \( t \).

- **Function abstraction**: \( \lambda x . t \)
  is the function with parameter \( x \) and result \( t \), i.e. \( x \mapsto t \).
Terms can be formed as follows:

- **Function application:** $f \ t$
  
  is the call of function $f$ with argument $t$.

  If $f$ has more arguments: $f \ t_1 \ t_2 \ldots$
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: $\sin \, \pi$, $\text{plus} \, x \, y$
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:** \( \lambda x. \, t \)

Examples: \( \sin \, \pi \), \( \text{plus} \, x \, y \)
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 \( t \)
Terms

Terms can be formed as follows:

- **Function application:** $f \ t$
  
is the call of function $f$ with argument $t$. If $f$ has more arguments: $f \ t_1 \ t_2 \ \ldots$

Examples: $\sin \ \pi$, $\text{plus} \ x \ y$

- **Function abstraction:** $\lambda \ x. \ t$
  
is the function with parameter $x$ and result $t$, i.e. “$x \mapsto t$”.
Terms

Terms can be formed as follows:

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

- **Function abstraction:** \( \lambda x. t \)
  is the function with parameter \( x \) and result \( t \),
  i.e. \( "x \mapsto t" \).
  Example: \( \lambda x. \text{plus} \ x \ x \)
Basic syntax:

\[ t ::= \]

Terms
Basic syntax:

\[ t ::= (t) \]
Terms

Basic syntax:

\[ t ::= (t) \]
\[ | \quad a \]

constant or variable (identifier)
Terms

Basic syntax:

\[ t ::= (t) \]
\[ \quad | \quad a \quad \text{constant or variable (identifier)} \]
\[ \quad | \quad t \ t \quad \text{function application} \]
Terms

Basic syntax:

\[ t ::= (t) \]
\[ | \quad a \quad \text{constant or variable (identifier)} \]
\[ | \quad t \\ t \quad \text{function application} \]
\[ | \quad \lambda x. \ t \quad \text{function abstraction} \]
Terms

Basic syntax:

\[ t ::= (t) \]

- \( a \): constant or variable (identifier)
- \( t t \): function application
- \( \lambda x. t \): function abstraction
- \( \ldots \): lots of syntactic sugar
Terms

Basic syntax:

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

Examples: \( f (g \; x) \; y \)
Basic syntax:

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

Examples:
\[ f \ (g \ x) \ y \]
\[ h \ (\lambda x. \ f \ (g \ x)) \]
Terms

Basic syntax:

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

Examples:

\[ f \ (g \ x) \ y \]
\[ h \ (\lambda x. \ f \ (g \ x)) \]

Convention:

\[ f \ t_1 \ t_2 \ t_3 \equiv ((f \ t_1) \ t_2) \ t_3 \]
Terms

Basic syntax:

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

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

\[
(\lambda x. \, t) \, u \ = \ t[u/x]
\]
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 $x$".
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 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.
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
Terms must be well-typed

(the argument of every function call must be of the right type)
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 \( \tau \)”. 
Terms must be well-typed
(the argument of every function call must be of the right type)

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

\[
\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2}
\]
Isabelle automatically computes the type of each variable in a term.
Isabelle automatically computes the type of each variable in a term. This is called *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.
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: \( f (x::\text{nat}) \)
Thou shalt Curry your functions
Currying

Thou shalt Curry your functions

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$
Currying

Thou shalt Curry your functions

- Curried: \( f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau \)
- Tupled: \( f' :: \tau_1 \times \tau_2 \Rightarrow \tau \)

Advantage:

Currying allows *partial application*

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

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

Prefix binds more strongly than infix:

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

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

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

Prefix binds more strongly than infix:

\[ f \ x + \ y \equiv (f \ x) + y \neq f (x + y) \]
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
Theory = Isabelle Module

Syntax:

theory \textit{MyTh}
imports \textit{T}_1 \ldots \textit{T}_n
begin
(definitions, theorems, proofs, ...)*
end
Theory = Isabelle Module

Syntax:

theory \textit{MyTh}
imports \(T_1 \ldots T_n\)
begin
(definitions, theorems, proofs, \ldots)*
end

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

\(T_i\) : names of imported theories. Import transitive.
Theory = Isabelle Module

Syntax:

```isabelle
theory MyTh
imports T_1 ... 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 "
Concrete syntax

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

Except for single identifiers
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
isabelle jedit

- Based on *jEdit* editor
- Processes Isabelle text automatically when editing `.thy` files
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 bool, nat and list

Summary
Type $bool$

datatype $bool = True \mid False$
Type \( bool \)

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

Predefined functions:
\( \land, \lor, \rightarrow, \ldots :: bool \Rightarrow bool \Rightarrow bool \)
Type `bool`

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

Predefined functions:
\( \wedge, \vee, \rightarrow, \ldots \) :: \( bool \Rightarrow bool \Rightarrow bool \)

A *formula* is a term of type `bool`
Type \textit{bool}

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

Predefined functions:
\&, \lor, \rightarrow, \ldots :: \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool}

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

if-and-only-if: =
**Type \texttt{nat}**

\begin{verbatim}
\textbf{datatype} \texttt{nat} = 0 | \texttt{Suc \, nat}
\end{verbatim}
**Type \textit{nat}**

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

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

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

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

Predefined functions: $+, \; *, \; \ldots :: nat \Rightarrow nat \Rightarrow nat$
Type `nat`

**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`
Type \textit{nat}

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

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

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

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

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

datatype \textit{nat} = 0 \mid \textit{Suc} \textit{nat}

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

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

Numbers and arithmetic operations are overloaded:

\begin{align*}
0, 1, 2, \ldots & : \ 'a, \\
+ & : \ 'a \Rightarrow \ 'a \Rightarrow \ 'a
\end{align*}

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

unless the context is unambiguous: \textit{Suc} \textit{z}
Nat_Demo.thy
An informal proof

**Lemma** \( add \ m \ 0 = m \)
An informal proof

Lemma \( add \ m \ 0 = m \)

Proof by induction on \( m \).
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 \( add \).
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).
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 \( add\ (Suc\ m)\ 0 = Suc\ m \).
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:
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:
  \( \text{add } (\text{Suc } m) \ 0 = \text{Suc } (\text{add } m \ 0) \) by def. of \( \text{add} \)
Lemma \( add \ m \ 0 = m \)

Proof by induction on \( m \).

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

- Case \( Suc \ m \) (the induction step):
  We assume \( add \ m \ 0 = m \),
  the induction hypothesis (IH).
  We need to show \( add \ (Suc \ m) \ 0 = Suc \ m \).
  The proof is as follows:
  \[
  add \ (Suc \ m) \ 0 = Suc \ (add \ m \ 0) \quad \text{by def. of } add
  = Suc \ m \quad \text{by IH}
  \]
Lists of elements of type 'a
Type 'a list

Lists of elements of type 'a

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

Lists of elements of type 'a

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

Some lists: Nil,
Type 'a list

Lists of elements of type 'a

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

Some lists: Nil, Cons 1 Nil,
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), ...
Type \texttt{'a list}

Lists of elements of type \texttt{'a}

\textbf{datatype} \quad \texttt{'a list} = \texttt{Nil} \mid \texttt{Cons 'a ('a list)}

Some lists: \texttt{Nil}, \; \texttt{Cons 1 Nil}, \; \texttt{Cons 1 (Cons 2 Nil)}, \ldots

Syntactic sugar:

- \texttt{[]} = \texttt{Nil}: empty 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”)
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 but fixed $x$ and $xs$, $P(xs)$ implies $P(x\#xs)$. 

Structural Induction for lists

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

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

\[
\begin{align*}
P([]) &\quad \land \ x \ xs. \ P(xs) \implies P(x\#xs) \\
P(xs) \quad \implies \quad P(xs)
\end{align*}
\]
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.
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!

Predefined: \( xs \@ ys \) (append),
Large library: HOL/List.thy

Included in Main.

Don’t reinvent, reuse!

Predefined: $xs @ ys$ (append), $length$, 
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 \texttt{bool}, \texttt{nat} and \texttt{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).
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):
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
```
Proofs

General schema:

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

\textbf{done}

If the lemma is suitable as a simplification rule:

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

Command

\texttt{sorry}

“completes” any proof.
Top down proofs

Command

\texttt{sorry}

“completes” any proof.

Allows top down development:

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

1. $\bigwedge x_1 \ldots x_p. \ A \implies B$
The proof state

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

$x_1 \ldots x_p$ fixed local variables
The proof state

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

$x_1 \ldots x_p$  fixed local variables
$A$  local assumption(s)
The proof state

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

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

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

abbreviates

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

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

abbreviates

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

\; \approx \; "and"
3 Overview of Isabelle/HOL

4 Type and function definitions

5 Induction Heuristics

6 Simplification
4 Type and function definitions

Type definitions

Function definitions
Type synonyms

```plaintext
type_synonym name = τ
```

Introduces a *synonym* `name` for type `τ`
Type synonyms

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

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

Examples

\texttt{type\_synonym \textit{string} = \textit{char list}}
**Type synonyms**

```
type_synonym name = τ
```

Introduces a *synonym name* for type $\tau$

**Examples**

```
type_synonym string = char list
```

```
type_synonym ('a,'b)foo = 'a list × 'b list
```
Type synonyms

\texttt{type\_synonym\ name = \tau}

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

\textbf{Examples}

\texttt{type\_synonym\ string = \texttt{char list}}

\texttt{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

datatype \((\alpha_1, \ldots, \alpha_n) t\) = \(C_1 \tau_{1,1} \ldots \tau_{1,n_1}\)
\[\vdash \ldots\]
\[\vdash C_k \tau_{k,1} \ldots \tau_{k,n_k}\]
**datatype — the general case**

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

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

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

- **Types**: $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t$
- **Distinctness**: $C_i \ldots \neq C_j \ldots$ if $i \neq j$
`datatype — the general case`

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

- **Types:** \( C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)t \)
- **Distinctness:** \( C_i \ldots \not= C_j \ldots \) 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
**datatype — the general case**

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

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

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

Datatype values can be taken apart with case:

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

Datatype values can be taken apart with case:

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

Wildcards: 

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

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

Nested patterns:

\[
\text{case } xs \text{ of } [0] \Rightarrow 0 \mid [Suc n] \Rightarrow n \mid _ \Rightarrow 2
\]
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!
Case expressions

Datatype values can be taken apart with case:

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

Wildcards: 

\[
(case \; m \; of \; 0 \; \Rightarrow \; Suc \; 0 \; | \; Suc \; _\; \Rightarrow \; 0)
\]

Nested patterns:

\[
(case \; xs \; of \; [0] \; \Rightarrow \; 0 \; | \; [Suc \; n] \; \Rightarrow \; n \; | \; _\; \Rightarrow \; 2)
\]

Complicated patterns mean complicated proofs!

Need ( ) in context
The *option* type

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

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)
```
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\)
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**
The `option` type

```plaintext
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:

```plaintext
fun lookup :: ('a × 'b) list ⇒ 'a ⇒ 'b option where
lookup [] x = None |
```
The \textit{option} type

\textbf{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:

\textbf{fun} \textit{lookup} :: ('a × \textit{'}b\) list ⇒ \textit{'}a ⇒ \textit{'}b\ option \textbf{where}

\textit{lookup} [ ] \(x = None |\)
\textit{lookup} \(((a, b) \# ps) \ x =\)
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)
4 Type and function definitions

Type definitions

Function definitions
Non-recursive definitions

Example

definition sq :: nat ⇒ nat where sq n = n*n
Non-recursive definitions

Example
definition $sq :: \text{nat} \Rightarrow \text{nat}$ where $sq \ n = n \times n$

No pattern matching, just $f \ x_1 \ldots \ x_n = \ldots$
The danger of nontermination

How about \( f(x) = f(x) + 1 \)?
The danger of nontermination

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

Subtract $f \, x$ on both sides.

$$
\Rightarrow 0 = 1
$$
The danger of nontermination

How about \( f \ x = f \ x + 1 \) ?

Subtract \( f \ x \) on both sides.

\[
\implies 0 = 1
\]

All functions in HOL must be total!
Key features of `fun`

- Pattern-matching over datatype constructors
Key features of `fun`

- Pattern-matching over datatype constructors
- Order of equations matters
Key features of fun

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

```haskell
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)
Example: Ackermann

\[
\text{fun } \text{ack} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \text{ where }
\]

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

Terminates because the arguments decrease \textit{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 \texttt{fun}
primrec

- A restrictive version of fun
- Means primitive recursive
- A restrictive version of \texttt{fun}
- Means \textit{primitive recursive}
- Most functions are primitive recursive
• A restrictive version of **fun**
• Means *primitive recursive*
• Most functions are primitive recursive
• Frequently found in Isabelle theories
• A restrictive version of \textbf{fun}
• Means \textit{primitive recursive}
• Most functions are primitive recursive
• Frequently found in Isabelle theories

The essence of primitive recursion:

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

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

lemma

\[ \text{itrev} \; \text{xs} \; [] = \text{rev} \; \text{xs} \]
A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
  itrev [] ys = ys |

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

```otto
fun itrev :: 'a list ⇒ 'a list ⇒ 'a list where
    itrev [] ys = ys |
    itrev (x#xs) ys =
```
A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

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

Generalisation
Generalisation

- Replace constants by variables

arbitrary in induction proof
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 \textit{structural induction}
So far, all proofs were by structural induction because all functions were primitive recursive.
So far, all proofs were by structural induction because all functions were primitive recursive.

In each induction step, 1 constructor is added.
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.
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)
Computation Induction

Example

fun div2 :: nat ⇒ nat where

\[ \text{div2 } 0 = 0 \mid \]
\[ \text{div2 } (Suc \ 0) = 0 \mid \]
\[ \text{div2 } (Suc(Suc \ n)) = Suc(\text{div2 } n) \]

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

\[
\begin{array}{ccc}
P(0) & P(Suc \ 0) & P(n) \implies P(Suc(Suc \ n)) \\
\hline
& & \hline
& & P(m)
\end{array}
\]
Computation Induction

Example

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

\[ \Rightarrow \text{induction rule div2.induct:} \]
\[
\begin{align*}
P(0) \quad P(\text{Suc } 0) \quad \wedge n. \quad P(n) \quad \Rightarrow \quad P(\text{Suc}(\text{Suc } n)) \\
P(m)
\end{align*}
\]
Computation Induction

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

\[
\text{prove } P(e) \text{ assuming } P(r_1), \ldots, P(r_k).
\]

Induction follows course of (terminating!) computation.

Motto: properties of \( f \) are best proved by rule \textbf{f.induct}.
If $f : \tau \Rightarrow \tau'$ is defined by \textbf{fun}, a special induction schema is provided to prove $P(x)$ for all $x : \tau$:

for each defining equation

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

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

\textbf{Motto:} properties of $f$ are best proved by rule $f.induct$. 

\textbf{Computation Induction}
Computation Induction

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

for each defining equation

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

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

Induction follows course of (terminating!) computation
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 \( f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau' \):
How to apply \( f \text{.induct} \)

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

\[ (\text{induction } a_1 \ldots a_n \text{ rule: } f \text{.induct}) \]
How to apply \textit{f.induct}

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

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

Heuristic:

- there should be a call $f \ a_1 \ldots \ a_n$ in your goal
How to apply \texttt{f.induct}

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

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

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
Simplification means . . .

Using equations $l = r$ from left to right

As long as possible
Simplification means . . .

Using equations $l = r$ from left to right
As long as possible

Terminology: equation $\rightsquigarrow$ simplification rule
Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ simplification rule

Simplification = (Term) Rewriting
An example

Equations:

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

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

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

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

\begin{align*}
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{align*}

\[ 0 + Suc \ 0 \leq Suc \ 0 + x \]

Rewriting:
An example

Equations:

\[
\begin{align*}
0 + n &= n \quad (1) \\
(Suc \ m) + n &= Suc \ (m + n) \quad (2) \\
(Suc \ m \leq Suc \ n) &= (m \leq n) \quad (3) \\
(0 \leq m) &= True \quad (4)
\end{align*}
\]

Rewriting:

\[
\begin{align*}
0 + Suc \ 0 &\leq Suc \ 0 + x \quad (1) \\
Suc \ 0 &\leq Suc \ 0 + x
\end{align*}
\]
An example

Equations:

\[ 0 + n = n \]  \hspace{1cm} (1)

\[ (\text{Suc } m) + n = \text{Suc } (m + n) \]  \hspace{1cm} (2)

\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \]  \hspace{1cm} (3)

\[ (0 \leq m) = \text{True} \]  \hspace{1cm} (4)

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{1cm} (1) \implies \hspace{1cm} (1)

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{1cm} (2) \implies \hspace{1cm} (2)

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

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

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

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

\[ 0 \leq 0 + x \]
An example

Equations:

\[ 0 + n = n \]  \hspace{1cm} (1)

\[ (\text{Suc } m) + n = \text{Suc } (m + n) \]  \hspace{1cm} (2)

\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \]  \hspace{1cm} (3)

\[ (0 \leq m) = \text{True} \]  \hspace{1cm} (4)

Rewriting:

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

\[ \text{Suc } 0 \leq \text{Suc } 0 + x \]  \hspace{1cm} (2) 

\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \]  \hspace{1cm} (3) 

\[ 0 \leq 0 + x \]  \hspace{1cm} (4)

\[ \text{True} \]
Conditional rewriting

Simplification rules can be conditional:

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

Example

\( p(0) = True \)

\( p(x) = \Rightarrow f(x) = g(x) \)

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

Simplification rules can be conditional:

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

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

\[
p(0) = True
\]

\[
p(x) \implies f(x) = g(x)
\]
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 \(g(0)\)
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.
Termination

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

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

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

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

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

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

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

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

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

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

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

Goal: 1. \( [ P_1; \ldots; P_m ] \Rightarrow C \)

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

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

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

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

- lemmas with attribute \textit{simp}
Proof method \textit{simp}

Goal: \hspace{1em} 1. \hspace{1em} \[ P_1; \ldots; P_m \] \Longrightarrow C

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

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

- lemmas with attribute \textit{simp}
- rules from \textit{fun} and \textit{datatype}
Proof method \textit{simp}

Goal: 1. \([ P_1; \ldots; P_m ] \implies C\)

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

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

- lemmas with attribute \textit{simp}
- rules from \textbf{fun} and \textbf{datatype}
- additional lemmas \textit{eq}_1 \ldots \textit{eq}_n
Proof method $\textit{simp}$

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

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

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

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

Goal: \textcolor{red}{1. \left[ P_1; \ldots; P_m \right] \implies C}

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

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

- lemmas with attribute \textit{simp}
- rules from \textit{fun} and \textit{datatyper}
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

Variations:

- $(\textit{simp} \ldots \textit{del:} \ldots)$ removes \textit{simp}-lemmas
- \textit{add} and \textit{del} are optional
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
auto versus simp

- `auto` acts on all subgoals
- `simp` acts only on subgoal 1
- `auto` applies `simp` and more
auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:
  \((\text{auto simp add: } \ldots \text{ simp del: } \ldots)\)
Definitions (**definition**) must be used **explicitly**:

\[(\text{simp add: } f\_def \ldots)\]
Rewriting with definitions

Definitions (definition) must be used explicitly:

\[(\text{simp add: } f	ext{-def ...})\]

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

Automatic:

\[ P \ (\text{if } A \text{ then } s \text{ else } t) \]

\[ = \]

\[ (A \rightarrow P(s)) \land (\neg A \rightarrow P(t)) \]
Case splitting with \textit{simp/auto}

Automatic:

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

By hand:

\[
P \ (\text{case} \ e \ of \ 0 \Rightarrow a \ | \ Suc \ n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \ e = Suc \ n \rightarrow P(b))
\]
Case splitting with *simp/auto*

Automatic:

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

By hand:

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

Proof method: (*simp split: nat.split*)
Case splitting with simp/auto

Automatic:

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

By hand:

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

Proof method: (simp split: nat.split)
Or auto.
Case splitting with \texttt{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:

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

Proof method: \texttt{(simp split: nat.split)}
Or \texttt{auto}. Similar for any datatype \texttt{t}: \texttt{t.split}
Simp_Demo.thy
Chapter 3

Case Study: IMP Expressions
Case Study: IMP Expressions
Case Study: IMP Expressions
This section introduces 

*arithmetic and boolean expressions* 

of our imperative languageIMP.
This section introduces

\textit{arithmetic and boolean expressions}

of our imperative language IMP.

IMP \textit{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"
Concrete and abstract syntax

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

Abstract syntax: trees, eg

```
+  
/\ 
/  \ 
a * 5 b
```
Concrete and abstract syntax

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

Abstract syntax:  trees, eg

Parser:  function from strings to trees
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 $Plus \ a \ (Times \ 5 \ b)$
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

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

where \( n \) can be any natural number and \( x \) any variable.
Concrete syntax is defined by a context-free grammar, eg

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

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

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

Variable names are strings, values are integers:

```
# type_synonym
vname = string

# datatype
aexp = N int | V vname | Plus aexp aexp
```
Datatype \( aexp \)

Variable names are strings, values are integers:

- **type_synonym** \( vname = \text{string} \)
- **datatype** \( aexp = N \text{ int} \mid V \ vname \mid Plus \ aexp \ aexp \)

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>( N \ 5 )</td>
</tr>
</tbody>
</table>
### Datatype $aexp$

Variable names are strings, values are integers:

- **type_synonym** $vname = string$
- **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>
</tbody>
</table>
Datatype $aexp$

Variable names are strings, values are integers:

- **type_synonym** $vname = string$
- **datatype** $aexp = N \text{ int} \mid V \ vname \mid Plus \ aexp \ aexp$

<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>
</tbody>
</table>
Datatype \texttt{aexp}

Variable names are strings, values are integers:

\begin{itemize}
\item \texttt{type\_synonym} \quad \texttt{vname} = \texttt{string}
\item \texttt{datatype} \quad \texttt{aexp} = N \texttt{int} \mid V \texttt{vname} \mid \texttt{Plus aexp aexp}
\end{itemize}

\begin{tabular}{|c|c|}
\hline
Concrete & Abstract \\
\hline
5 & \texttt{N 5} \\
x & \texttt{V "x"} \\
x+y & \texttt{Plus (V "x") (V "y")} \\
2+(z+3) & \texttt{Plus (N 2) (Plus (V "z") (N 3))} \\
\hline
\end{tabular}
Warning

This is syntax, not (yet) semantics!
Warning

This is syntax, not (yet) semantics!

\[ N \circ 0 \neq \text{Plus} \ (N \circ 0) \ (N \circ 0) \]
The (program) state

What is the value of \( x+1 \)?
The (program) state

What is the value of $x+1$?

- The value of an expression depends on the value of its variables.
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 (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:
The (program) state

What is the value of \( x+1 \)?

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

\begin{verbatim}
  type_synonym val = int
  type_synonym state = vname ⇒ val
\end{verbatim}
Function update notation

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

$$f(a := b)$$
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 \( a \).
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:

- $\lambda x. 0$
How to write down a state

Some states:

- \( \lambda x. 0 \)
- \( (\lambda x. 0)("a" := 3) \)
How to write down a state

Some states:

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

\(<"a" := 5, "x" := 3, "y" := 7>\)
How to write down a state

Some states:

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

Nicer notation:

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

Maps everything to 0, but $\"a\"$ to 5, $\"x\"$ to 3, etc.
AExp.thy
Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
BExp.thy
Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

Stack Machine and Compilation
ASM.thy
This was easy.
This was easy.
Because evaluation of expressions always terminates.
This was easy. Because evaluation of expressions always terminates. But execution of programs may *not* terminate.
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.
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):

\[
\begin{align*}
\text{form} & ::= (\text{form}) \\
& \mid \text{term} = \text{term} \\
& \mid \neg \text{form} \\
& \mid \text{form} \land \text{form} \\
& \mid \text{form} \lor \text{form} \\
& \mid \forall x. \text{form} \\
& \mid \exists x. \text{form}
\end{align*}
\]
Syntax (in decreasing precedence):

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

Examples:

\[ \neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \]
Syntax (in decreasing precedence):

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

Examples:

\[ \neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \]
\[ s = t \land C \equiv (s = t) \land C \]
Syntax (in decreasing precedence):

\[
\begin{align*}
form & ::= (form) \mid term = term \mid \neg form \\
& \mid form \land form \mid form \lor form \\
& \mid \forall x. \, form \mid \exists x. \, form
\end{align*}
\]

Examples:

\[
\begin{align*}
\neg A \land B \lor C & \equiv ((\neg A) \land B) \lor C \\
s = t \land C & \equiv (s = t) \land C \\
A \land B = B \land A & \equiv A \land (B = B) \land A
\end{align*}
\]
Syntax (in decreasing precedence):

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

Examples:

\[
\neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \\
\equiv (s = t) \land C
\]

\[
A \land B = B \land A \equiv A \land (B = B) \land A
\]

\[
\forall x. P x \land Q x \equiv \forall x. (P x \land Q x)
\]
Syntax (in decreasing precedence):

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

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:

\[ \forall x \; y. \; P \; x \; y \equiv \forall x. \; \forall y. \; P \; x \; y \]
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 \text{ ALL}
\exists \text{ EX}
\lambda \% 
\rightarrow \text{ -->}
\leftrightarrow \text{ <->}
\&
| 
\sim 
\neq
Sets over type \( 'a \)

\( 'a \) set
Sets over type 'a

'a set

• \{\}, \{e_1, \ldots, e_n\}
Sets over type 'a

'a set

- \{\}, \{e_1, \ldots, e_n\}
- e \in A, A \subseteq B
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
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
- ...
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 \; \text{\textbackslash\textless in\textgreater} : \quad \subseteq \; \text{\textbackslash\textless subseteq\textgreater} \quad \leq \quad \cup \; \text{\textbackslash\textless union\textgreater} \quad \text{Un} \quad \cap \; \text{\textbackslash\textless inter\textgreater} \quad \text{Int}
Set comprehension

- \( \{ x. \, P \} \) where \( x \) is a variable

- But not \( \{ t. \, P \} \) where \( t \) is a proper term

- Instead: \( \{ t | x \, y \, z. \, P \} \) is short for \( \{ \forall v. \, \exists x \, y \, z. \, v = t \land P \} \) where \( x, y, z \) are the free variables in \( t \).
Set comprehension

- \{x. P\} where \(x\) is a variable
- But not \{t. P\} where \(t\) is a proper term
Set comprehension

- \( \{ x. \ P \} \) where \( x \) is a variable
- But not \( \{ t. \ P \} \) where \( t \) is a proper term
- Instead: \( \{ t \mid x \ y \ z. \ P \} \)
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 \( t \)
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
**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
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
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
**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.
• rewriting, logic, sets, relations and a bit of arithmetic.
• incomplete but better than *auto*.
fastforce

- rewriting, logic, sets, relations and a bit of arithmetic.
- incomplete but better than auto.
- Succeeds or fails
fastforce

- rewriting, logic, sets, relations and a bit of arithmetic.
- **incomplete** but better than auto.
- Succeeds or fails
- Extensible with new simp-rules
A complete proof search procedure for FOL . . .
• A complete proof search procedure for FOL . . .
• . . . but (almost) without “=”
• A complete proof search procedure for FOL . . .
• . . . but (almost) without “=”
• Covers logic, sets and relations
blast

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

**arith**:
- proves linear formulas (no “*”)

133
Automating arithmetic

\textit{arith}:

- proves linear formulas (no “\(*\)"")
- complete for quantifier-free \textit{real} arithmetic
Automating arithmetic

$\text{arith}$:

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

Isabelle

external ATPs\(^1\)

\(^1\)Automatic Theorem Provers
Architecture:

Isabelle

Goal

& filtered library

↓

external

ATPs

1

\(^1\)Automatic Theorem Provers
Architecture:

Isabelle

Goal & filtered library

Proof

external ATPs\textsuperscript{1}

\textsuperscript{1}Automatic Theorem Provers
Architecture:

Goal & filtered library \[\downarrow\]

Isabelle \[\downarrow \uparrow\]

Proof

external ATPs\(^1\)

Characteristics:

- Sometimes it works,
Architecture:

Goal & filtered library  \rightarrow  Proof

Isabelle

external ATPs

Characteristics:

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

1 Automatic Theorem Provers
Architecture:

Isabelle

Goal & filtered library

Proof

external

ATPs\(^1\)

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
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 \textit{?}-\textit{variables}?
What are these ?-variables?

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

Example: theorem `conjI`:

```plaintext
```

These ?-variables can later be instantiated:

- By hand: `conjI[of "a=b" "False"]` $\Rightarrow$ `[ [ a = b ; False ] ] = ⇒ a = b ∧ False`
What are these $?-$variables ?

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

Example: theorem conjI: $[?P; ?Q] \implies ?P \land ?Q$
What are these ?-variables?

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

Example: theorem conjI: $[?P; ?Q] \rightarrow ?P \land ?Q$

These ?-variables can later be instantiated:
What are these \(?\)-variables \(\text{?}\) ?

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

Example: theorem \text{conjI}: \[\[\text{?P; ?Q}\] \implies \text{?P \land ?Q}\]

These \(?\)-variables can later be instantiated:

- By hand:
  \begin{verbatim}
  conjI[of "a=b" "False"] \implies
  \end{verbatim}
What are these \(?\)-variables? 

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

Example: theorem conjI: \([\text{?}P; \text{?}Q] \implies \text{?}P \land \text{?}Q\)

These \(?\)-variables can later be instantiated:

- By hand:
  \[\text{conjI[of "a=b" "False"] ⇝ [a = b; False] \implies a = b \land False}\]
What are these ?-variables?

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

Example: theorem conjI: $[?P; ?Q] \Rightarrow ?P \land ?Q$

These ?-variables can later be instantiated:

- By hand:
  \[
  \text{conjI[of "a=b" "False"]} \mapsto
  [a = b; False] \Rightarrow a = b \land False
  \]

- By unification:
  
  unifying $?P \land ?Q$ with $a = b \land False$
What are these \texttt{?}-variables? \\

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

Example: theorem \texttt{conjI}: \([ ?P; ?Q] \Longrightarrow ?P \land ?Q \)

These \texttt{?}-variables can later be instantiated:

- By hand:
  \[
  \texttt{conjI[of "a=b" "False"]} \mapsto
  [ a = b; False] \Longrightarrow a = b \land False
  \]

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

Example: rule:

\[ \left[ \[ \?P; \?Q \right\] \right] = \Rightarrow \?P \land \?Q

Subgoal:

1. \ldots = \Rightarrow \text{A} \land \text{B}

Result:

1. \ldots = \Rightarrow \text{A}
2. \ldots = \Rightarrow \text{B}

The general case: applying rule

\[ \left[ \[ \text{A}_1; \ldots; \text{A}_n \right\] \right] = \Rightarrow \text{A}

to subgoal

\ldots = \Rightarrow \text{C}:

- Unify \text{A} and \text{C}
- Replace \text{C} with \( n \) new subgoals \text{A}_1 \ldots \text{A}_n

apply (rule xyz)

"Backchaining"
Rule application

Example: rule: \([ ?P; ?Q ] \implies ?P \land ?Q \)
Rule application

Example:

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

subgoal: 1. \ldots \implies A \land B
Rule application

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

subgoal: 1. \( \ldots \implies A \land B \)

Result: 1. \( \ldots \implies A \)
        2. \( \ldots \implies B \)
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:
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 \(C\)
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
Rule application

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

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

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

2. ... $\implies B$

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

to subgoal ... $\implies C$:

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

apply($rule\ xyz$)
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 \( A_1 \ldots A_n \)

\textbf{apply}(\textit{rule xyz})

“Backchaining”
Typical backwards rules

\[ \begin{array}{c} ?P \\ ?Q \end{array} \quad \text{conjI} \]

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

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

\[
\frac{?P \iff ?Q}{?P \implies ?Q \implies ?Q} \text{ impI}
\]
Typical backwards rules

\[
\begin{align*}
\frac{?P \quad ?Q}{?P \land ?Q} & \text{ conjI} \\
\frac{?P \iff ?Q}{?P \quad \iff ?Q} & \text{ impI} \\
\frac{?P \quad \rightarrow ?Q}{?P \quad \rightarrow ?Q} & \text{ allI}
\end{align*}
\]
Typical backwards rules

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

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

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

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

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

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

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

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

If \( r \) is a theorem \( [A_1; ...; A_n] = \Rightarrow A \) then \( (\text{blast intro: } r) \) allows blast to backchain on \( r \) during proof search.

Example: theorem le 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 apply (blast intro: le trans)

Also works for auto and fastforce

Can greatly increase the search space!
Automating 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.
Automating intro rules

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

\[
(blast \ intro: r)
\]

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

Example:

\textbf{theorem} \( le\_trans: \left[ ?x \leq ?y; ?y \leq ?z \right] \Rightarrow ?x \leq ?z \)
Automating intro rules

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

$$(\text{blast intro: } r)$$

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

Example:

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

goal 1. $[ a \leq b; \ b \leq c; \ c \leq d ] \Rightarrow a \leq d$
Automating intro rules

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

$$(\text{blast intro: } r)$$

allows $\text{blast}$ to backchain on $r$ during proof search.

Example:

theorem $\text{le_trans}$: $[\ ?x \leq \ ?y; \ ?y \leq \ ?z ] \implies \ ?x \leq \ ?z$

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

proof $\text{apply(blast intro: le_trans)}$
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:

\textbf{theorem} \textit{le_trans}: $[\ ?x \leq \ ?y; \ ?y \leq \ ?z \ ] \implies \ ?x \leq \ ?z$

\textbf{goal} 1. $[a \leq b; \ b \leq c; \ c \leq d \ ] \implies a \leq d$

\textbf{proof} \textit{apply}(\text{blast intro: le_trans})

Also works for \textit{auto} and \textit{fastforce}
Automating intro rules

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

$$(\text{blast intro: } r)$$

allows $\text{blast}$ to backchain on $r$ during proof search.

Example:

**Theorem** $\text{le_trans}$: $[\ ?x \leq \ ?y; \ ?y \leq \ ?z ] \Rightarrow \ ?x \leq \ ?z$

**Goal** 1. $[\ a \leq \ b; \ b \leq \ c; \ c \leq \ d \ ] \Rightarrow \ a \leq \ d$

**Proof** apply($\text{blast intro: le_trans}$)

Also works for $\text{auto}$ and $\text{fastforce}$

Can greatly increase the search space!
Forward proof: OF

If \( r \) is a theorem \( A \rightarrow B \)
If \( r \) is a theorem \( A \implies B \)
and \( s \) is a theorem that unifies with \( A \)
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: $?t = ?t \implies a = a \land ?Q$
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: \(?t = ?t\)
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: $?t = ?t$

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

If \( r \) is a theorem \( A \implies B \) and \( s \) is a theorem that unifies with \( A \) then

\[
\text{\( r[\text{OF } s] \)}
\]

is the theorem obtained by proving \( A \) with \( s \).

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

\[
\text{conjI}[\text{OF } \text{refl[of "a"]}] \\
\implies \\
?Q \implies a = a \land ?Q
\]
The general case:

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

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

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

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

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

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

Example: theorem refl: $?t = ?t$
The general case:

If $r$ is a theorem $\left[ A_1; \ldots; A_n \right] \implies A$ 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: $?t = ?t$

$$\text{conjI}[\text{OF refl[of } "a"] \text{ refl[of } "b"]]$$
The general case:

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

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

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

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

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

$$\implies a = a \land b = b$$
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 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 \( A \).
is part of the Isabelle framework. It structures theorems and proof states: \([ A_1; \ldots; A_n ] \Rightarrow A\)

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

Phrase theorems like this \([ A_1; \ldots; A_n ] \Rightarrow A\)
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:

\texttt{inductive ev :: \texttt{nat} \Rightarrow \texttt{bool}}
\texttt{where}
\texttt{ev 0 | ev n = \Rightarrow ev (n + 2)}
Example: even numbers

Informally:

- 0 is even
Example: even numbers

Informally:

- 0 is even
- If $n$ is even, so is $n + 2$
Example: even numbers

Informally:

- 0 is even
- If \( n \) is even, so is \( n + 2 \)
- These are the only even numbers
Example: even numbers

Informally:

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

In Isabelle/HOL:

```isabelle
inductive ev :: nat ⇒ bool
```
Example: even numbers

Informally:

- 0 is even
- If \( n \) is even, so is \( n + 2 \)
- These are the only even numbers

In Isabelle/HOL:

```isabelle
inductive ev :: nat ⇒ bool
where
```

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

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

\[\textbf{fun evn :: nat } \Rightarrow \textbf{ bool where}\]
\[\text{evn } 0 = \text{True } | \]
\[\text{evn } (\text{Succ } 0) = \text{False } | \]
\[\text{evn } (\text{Succ } (\text{Succ } n)) = \text{evn } n\]
Consider

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

A trickier proof: \( ev m \iff evn m \)
Consider

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

A trickier proof: ev m ⇒ evn m

By induction on the structure of the derivation of ev m
Consider

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

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

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

Two cases: \( ev m \) is proved by
  - rule \( ev 0 \)
Consider

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

A trickier proof: \( \text{ev} \ m \ \Rightarrow \ evn \ m \)

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

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

- rule \text{ev} 0

\( \Rightarrow m = 0 \Rightarrow evn \ m = True \)
Consider

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

A trickier proof: \(\text{ev } m \implies \evn m\)

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

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

- rule \text{ev } 0

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

- rule \text{ev } n \implies \text{ev } (n+2)
Consider

**fun** evn :: nat \(\Rightarrow\) bool **where**

\[
evn 0 = \text{True} \\
\text{evn (Suc 0) = False} \\
\text{evn (Suc (Suc n)) = evn n}
\]

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

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

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

- **rule** \(ev 0\)
  \[\Rightarrow m = 0 \Rightarrow evn m = \text{True}\]

- **rule** \(ev n \Rightarrow ev (n+2)\)
  \[\Rightarrow m = n+2 \text{ and } evn n (\text{IH})\]
Consider

**fun evn :: nat \to bool where**

\[
evn 0 = \text{True} \mid \\
evn (\text{Suc } 0) = \text{False} \mid \\
evn (\text{Suc } (\text{Suc } n)) = evn n
\]

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

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

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

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

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

To prove

\[
ev n \implies P n
\]

by *rule induction* on \( ev n \) we must prove
Rule induction for $ev$

To prove

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

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

- $P \ 0$
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 induction for \( ev \)

To prove

\[
ev n \implies P n
\]

by \textit{rule induction} on \( ev n \) we must prove

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

Rule \texttt{ev.induct}:

\[
\begin{array}{c}
ev n \quad P 0 \quad \land n. \quad [\ ev n; \ P n \ ] \\
\end{array} \implies P(n+2)
\]

\[
P \ n
\]
Format of inductive definitions

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

Note:
- $I$ may have multiple arguments.
- Each rule may also contain side conditions not involving $I$. 
Format of inductive definitions

\begin{align*}
\textbf{inductive} \quad & I :: \tau \Rightarrow \text{bool where} \\
[ I a_1; \ldots ; I a_n ] & \Rightarrow I a
\end{align*}
Format of inductive definitions

\texttt{inductive } I :: \tau \Rightarrow bool \texttt{ where}

\[
[ I \ a_1 ; \ldots ; I \ a_n ] \Rightarrow I \ a
\]
Format of inductive definitions

\textbf{inductive } I :: \tau \Rightarrow \textit{bool} \textbf{ where }
\begin{align*}
[ I \ a_1 ; \ldots ; I \ a_n ] & \Rightarrow I \ a \\
\vdots
\end{align*}

\textbf{Note:}

\begin{itemize}
  \item \textit{I} may have multiple arguments.
\end{itemize}
Format of inductive definitions

\[ \text{inductive } I :: \tau \Rightarrow \text{bool where} \]
\[ [ I a_1 ; \ldots ; I a_n ] \implies I a \]
\[ \vdots \]

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 \( I \ x \)
Rule induction in general

To prove

\[ I \ x \ \Longrightarrow \ P \ x \]

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

\[ \left[ I \ a_1; \ldots ; I \ a_n \right] \ \Longrightarrow \ I \ a \]

that \( P \) is preserved:
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

\[ \text{inductive_set } I :: \tau \text{ set where} \]
Inductively defined sets

\begin{verbatim}
inductive_set I :: τ set where
  \[ a_1 \in I; \ldots ; a_n \in I \] ⇒ a ∈ I
\end{verbatim}
Inductively defined sets

\texttt{inductive\_set} \ I :: \ \tau \ set \ \textbf{where}

\[ \begin{array}{l}
\left[ \ a_1 \in I; \ldots ; \ a_n \in I \right] \Rightarrow a \in I \\
\vdots
\end{array} \]
Inductively defined sets

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

Difference to \texttt{inductive}: 
- arguments of \( I \) are tupled, not curried
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
Apply scripts

- unreadable
- hard to maintain
Apply scripts

- unreadable
- hard to maintain
- do not scale
Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!
Apply scripts versus Isar proofs

Apply script = assembly language program
Apply scripts versus Isar proofs

Apply script = assembly language program
Isar proof = structured program with assertions
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 \( \text{formula}_1 \) \textbf{by simp} 
  
  
  have \( \text{formula}_n \) \textbf{by blast} 
  show \( \text{formula}_{n+1} \) \textbf{by } \ldots 

qed
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

\[
\text{proof} \; = \; \text{proof} \; [\text{method}] \; \text{step}^* \; \text{qed} \\
\mid \; \text{by} \; \text{method}
\]
Isar core syntax

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

method = (simp ...) | (blast ...) | (induction ...) | ...
Isar core syntax

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

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

step = fix variables (\land)
   | assume prop (\equiv)
   | [from fact^+] (have | show) prop proof
Isar core syntax

\[
\text{proof} = \text{proof} \ [\text{method}] \ \text{step}^* \ \text{qed}
\]

| \text{by} \ \text{method} |

\[
\text{method} = (\text{simp} \ldots) | (\text{blast} \ldots) | (\text{induction} \ldots) | \ldots
\]

\[
\text{step} = \text{fix} \ \text{variables} \quad (\wedge)
\]

| \text{assume} \ \text{prop} \quad (\implies) |

| \text{[from fact]}^+ \ \text{(have} \ | \ \text{show}) \ \text{prop} \ \text{proof} |

\[
\text{prop} = \ [\text{name:}] \ "\text{formula}" 
\]
Isar core syntax

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

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

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

prop = [name:] ”formula”

fact = name | . . .
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Example: Cantor’s theorem

lemma \( \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set}) \)
Example: Cantor’s theorem

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

proof
Example: Cantor’s theorem

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

proof default proof: assume \text{surj}, show \text{False}
Example: Cantor's theorem

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

proof  
  default proof: assume \( \text{surj} \), show \( \text{False} \)

  assume \( a :: \text{surj} f \)
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 \( b: \forall A. \exists a. A = f a \)
Example: Cantor’s theorem

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

**Proof**

- Default proof: assume \( \text{surj} \), show \( \text{False} \)

- Assume \( a: \text{surj } f \)

- From \( a \) have \( b: \forall A. \exists a. A = f a \)

- By (simp add: surj_def)
Example: Cantor’s theorem

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

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

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


Example: Cantor’s theorem

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

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

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

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

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

    from \( c \) show \text{False}
**Example: Cantor’s theorem**

```
lemma ¬ surj(f :: 'a ⇒ 'a set)
proof  default proof: assume surj, show False
  assume a: surj f
  from a have b: ∀ A. ∃ a. A = f a
    by (simp add: surj_def)
  from b have c: ∃ a. {x. x ∉ f x} = f a
    by blast
  from c show False
    by blast
```
Example: Cantor’s theorem

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

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

  assume a: \text{surj } f

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

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

  from c show 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
using and with

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

\[= \]

\[\text{from } \text{facts} \ (\text{have} \mid \text{show}) \text{ prop} \]
using and with

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

**Lemma**

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

**assumes** $s :: \text{surj } f$

**shows** $False$
lemma

fixes \( f :: \text{'}a \Rightarrow \text{'}a \text{ set} \)

assumes \( s: \text{surj } f \)

shows \( \text{False} \)

proof —

Structured lemma statement

lemma
  fixes $f :: 'a \Rightarrow 'a \text{ set}$
  assumes $s : \text{surj } f$
  shows $False$
proof — no automatic proof step
Structured lemma statement

lemma

  fixes  \( f :: 'a \rightarrow 'a \Rightarrow 'a \ \text{set} \)
  assumes \( s: \text{surj} \ f \)
  shows \( \text{False} \)

proof — no automatic proof step

have \( \exists \ a. \ \{ x. x \notin f \ x \} = f \ a \ \text{using} \ s \)
  by (auto simp: \text{surj\_def})

Proves \( \text{surj} \ f = \Rightarrow \text{False} \)
but \( \text{surj} \ f \) becomes local fact \( s \) in proof.
Structured lemma statement

lemma

  fixes \( f :: 'a \Rightarrow 'a \Rightarrow \text{set} \)

  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
Structured lemma statement

lemma
  fixes f :: 'a ⇒ 'a set
  assumes s: surj f
  shows False
proof — no automatic proof step
  have ∃ a. {x. x ∉ f x} = f a using s
    by (auto simp: surj_def)
thus False by blast
qed

Proves surj f ⇒ False
Structured lemma statement

lemma
  fixes f :: 'a ⇒ 'a set
  assumes s: surj f
  shows False
proof — no automatic proof step
  have ∃ a. {x. x ∉ f x} = f a using s
    by (auto simp: surj_def)
thus False by blast
qed

Proves surj f ⟷ 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 \( y :: \tau_2 \) 
assumes \( a: P \) and \( b: Q \) 
shows \( R \)
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
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
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 \langle \text{proof} \rangle$
next
  assume $\neg P$
  :$
  show R \langle \text{proof} \rangle$
qed
Case distinction

show \( R \)
proof cases
  assume \( P \)
  : 
  show \( R \) \( \langle \text{proof} \rangle \)
next
  assume \( \neg P \)
  : 
  show \( R \) \( \langle \text{proof} \rangle \)
qed

have \( P \lor Q \) \( \langle \text{proof} \rangle \)
then show \( R \)
proof
  assume \( P \)
  : 
  show \( R \) \( \langle \text{proof} \rangle \)
next
  assume \( Q \)
  : 
  show \( R \) \( \langle \text{proof} \rangle \)
qed
show \neg P
proof
  assume \neg P
  :
  show False ⟨proof⟩
qed
Contradiction

\[
\text{show } \neg P \\
\text{proof} \\
\text{assume } P \\
\vdash \\
\text{show } False \langle \text{proof} \rangle \\
\text{qed}
\]

\[
\text{show } P \\
\text{proof } (\text{rule ccontr}) \\
\text{assume } \neg P \\
\vdash \\
\text{show } False \langle \text{proof} \rangle \\
\text{qed}
\]
show $P \leftrightarrow Q$
proof
  assume $P$
  :
  :
  show $Q$ ⟨proof⟩
next
  assume $Q$
  :
  :
  show $P$ ⟨proof⟩
qed
∀ and ∃ introduction

show \( \forall x. \ P(x) \)
proof
  \begin{align*}
  \text{fix } x & \quad \text{local fixed variable} \\
  \text{show } P(x) & \quad \langle proof \rangle
  \end{align*}
qed
∀ and ∃ introduction

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

show $\exists x. \ P(x)$
proof
  : 
  : 
  show $P(witness) \langle proof \rangle$
qed
\exists \text{ elimination: obtain}
for one or more $x$.

\[ \exists \quad \text{have } \exists x. \ P(x) \]
\[ \text{then obtain } x \quad \text{where } p: \ P(x) \quad \text{by blast} \]
\[ : \quad x \quad \text{fixed local variable} \]
\( \exists \) 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 \neg \text{surj}(f :: \ 'a \Rightarrow \ 'a \text{ set})

proof
  assume \text{surj } f
  hence \exists a. \{ x. \ x \notin f \ x \} = f \ a \text{ by (auto simp: surj_def)}
lemma \( \neg \text{surj}(f :: 'a \Rightarrow 'a\;\text{set}) \)

proof

assume \( \text{surj}\;f \)

hence \( \exists a. \{x. x \not\in f\;x\} = f\;a \) by (auto simp: surj_def)

then obtain \( a \) where \( \{x. x \not\in f\;x\} = f\;a \) by blast
lemma \( \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set}) \)

proof

assume \( \text{surj } f \)

hence \( \exists a. \{x. x \notin f\ x\} = f\ a \) by (auto simp: surj_def)

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

hence \( a \notin f\ a \leftrightarrow a \in f\ a \) by blast

thus False by blast

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

proof

assume \text{surj } f

hence \( \exists a. \{ x. x \notin f x \} = f a \) by (auto simp: surj_def)

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

hence \( a \notin f a \leftarrow\rightarrow a \in f a \) by blast

thus \( \text{False} \) by blast

qed
Set equality and subset

\begin{proof}
\begin{enumerate}
\item \textbf{show} $A = B$
\item \textbf{proof}
\begin{enumerate}
\item \textbf{show} $A \subseteq B$ \langle \textit{proof} \rangle
\end{enumerate}
\item \textbf{next}
\begin{enumerate}
\item \textbf{show} $B \subseteq A$ \langle \textit{proof} \rangle
\end{enumerate}
\end{enumerate}
\end{proof}
Set equality and subset

\[ A = B \]

**Proof**

\[ A \subseteq B \]

\[ \text{proof} \]

**Proof**

\[ B \subseteq A \]

**Next**

\[ \text{proof} \]

**Proof**

\[ x \in A \]

**Assume**

\[ x \in B \]

**Show**

\[ x \in A \]

**Show**

\[ x \in B \]

**Q.E.D.**
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 $formula_1 \leftrightarrow formula_2$ (is $?L \leftrightarrow ?R$)
Example: pattern matching

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

proof

\begin{align*}
\text{assume } & ?L \\
\vdots & \\
\text{show } & ?R \langle \text{proof} \rangle \\
\text{next} & \\
\text{assume } & ?R \\
\vdots & \\
\text{show } & ?L \langle \text{proof} \rangle \\
\text{qed} &
\end{align*}
show $\text{formula}$

proof -

::

  show $\text{thesis} \langle \text{proof} \rangle$

qed
show formula (is ?thesis)
proof -
  :
  show ?thesis ⟨proof⟩
qed
show \textit{formula} \ (is \ \textit{thesis})

proof -

\begin{itemize}
  \item show \textit{thesis} \langle \textit{proof}\rangle
\end{itemize}

\textit{qed}

Every show implicitly defines \textit{thesis}
Introducing local abbreviations in proofs:

```plaintext
let ?t = "some-big-term"

have "...?t..."
```
Quoting facts by value

By name:

have \( x_0: "x > 0" \ldots \)

; ;;
from \( x_0 \ldots \)
Quoting facts by value

By name:

```
have x0: ”x > 0” ...
:\nfrom x0 ...
```

By value:

```
have ”x > 0” ...
:\nfrom ‘x>0‘ ...
Quoting facts by value

By name:

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

By value:

```plaintext
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

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

**Lemma**

$$\exists\ ys\ zs.\ xs = ys \mathbin{@} zs \land$$

$$(\text{length}\ ys = \text{length}\ zs \lor \text{length}\ ys = \text{length}\ zs + 1)$$

**Proof** ???
Isar_Demo.thy

Top down proof development
When automation fails

Split proof up into smaller steps.
When automation fails

Split proof up into smaller steps.

Or explore by **apply**: 
When automation fails

Split proof up into smaller steps.

Or explore by apply:

have ... using ...
When automation fails

Split proof up into smaller steps.

Or explore by apply:

  have \ldots using \ldots
  apply - to make incoming facts
  part of proof state
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
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 ...
When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

- **have** . . . **using** . . .
- apply -
- apply *auto*
- apply . . .

At the end:

- done
- Better: convert to structured proof

- to make incoming facts part of proof state
- or whatever
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**
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 $P_1$ . . .

moreover

have $P_2$ . . .

moreover

::

moreover

have $P_n$ . . .

ultimately

have $P$ . . .
moreover—ultimately

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

have $\text{lab}_1$: $P_1$ \ldots
have $\text{lab}_2$: $P_2$ \ldots
\vvdots
have $\text{lab}_n$: $P_n$ \ldots
from $\text{lab}_1$ $\text{lab}_2$ \ldots
have $P$ \ldots

With names
Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

moreover

Local lemmas
Local lemmas

\begin{proof}
\begin{align*}
\text{have } B & \text{ if name: } A_1 \ldots A_m \text{ for } x_1 \ldots x_n \\
\end{align*}
\end{proof}
Local lemmas

\( \textbf{have } B \quad \textbf{if name: } A_1 \ldots A_m \quad \textbf{for } x_1 \ldots x_n \quad \langle \text{proof} \rangle \)

proves \[
\left[ A_1; \ldots ; A_m \right] \implies B
\]
have $B$ if name: $A_1 \ldots A_m$ for $x_1 \ldots x_n$

(proof)

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

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

proof method
Proof state and Isar text

In general: \textbf{proof} \textit{method}

Applies \textit{method} and generates subgoal(s):

\[ \bigwedge x_1 \ldots x_n. \ [ A_1; \ldots ; A_m ] \implies B \]
Proof state and Isar text

In general: \textbf{proof }method

Applies \textit{method} and generates subgoal(s):

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

How to prove each subgoal:
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:

- **fix** \( x_1 \ldots x_n \)
- **assume** \( A_1 \ldots A_m \)
- : 
- **show** \( B \)
Proof state and Isar text

In general: proof \textit{method}

Applies \textit{method} and generates subgoal(s):

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

How to prove each subgoal:

\begin{itemize}
  \item \textbf{fix} \; x_1 \ldots x_n
  \item \textbf{assume} \; A_1 \ldots A_m
  \item : 
  \item \textbf{show} \; B
\end{itemize}

Separated by \textbf{next}
Isar by example

Proof patterns

Streamlining Proofs

Proof by Cases and Induction
Isar_Induction_Demo.thy

Proof by cases
Datatype case analysis

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

datatype \( t = C_1 \tau \mid \ldots \)

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

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

next
:
:
qed
Datatype case analysis

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

proof \((\text{cases "term"})\)
  \begin{align*}
  \text{case } (C_1 x_1 \ldots x_k) \\
  \ldots x_j \ldots
  \end{align*}

next

: 

qed

where \begin{align*}
\text{case } (C_i x_1 \ldots x_k) & \equiv \\
\text{fix } x_1 \ldots x_k \\
\text{assume } C_i: & \begin{cases}
\text{label} & \text{term} = (C_i x_1 \ldots x_k) \\
\text{formula} & 
\end{cases}
\end{align*}
Isar\_Induction\_Demo.thy

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

\begin{verbatim}
show $P(n)$
proof (induction n)
  case 0
  ...
  show $?\text{case}$
next
  case (Suc n)
  ...
  ...
  show $?\text{case}$
qed
\end{verbatim}
Structural induction for $\textit{nat}$

\begin{align*}
\text{show} & \quad P(n) \\
\text{proof} & \quad (\text{induction } n) \\
\text{case} & \quad 0 \quad \equiv \quad \text{let} \quad ?\text{case} = P(0) \\
\vdots & \\
\text{show} & \quad ?\text{case} \\
\text{next} & \\
\text{case} & \quad (\textit{Suc } n) \\
\vdots & \\
\vdots & \\
\text{show} & \quad ?\text{case} \\
\text{qed} &
\end{align*}
Structural induction for \( \text{nat} \)

\[
\begin{align*}
\text{show } & P(n) \\
\text{proof } & (\text{induction } n) \\
\text{ case } & 0 \\
\quad & \equiv \text{ let } \ ?\text{case} = P(0) \\
\quad & \vdash \text{ show } \ ?\text{case} \\
\text{next } \\
\text{ case } & (\text{Suc } n) \\
\quad & \equiv \text{ fix } n \text{ assume Suc: } P(n) \\
\quad & \text{ let } \ ?\text{case} = P(\text{Suc } n) \\
\quad & \vdash \text{ show } \ ?\text{case} \\
\text{qed}
\end{align*}
\]
show $A(n) \Rightarrow P(n)$

proof (induction $n$)

    case 0
    
    :  

    show ?case

next

    case ($Suc\ n$)
    
    :  

    :  

    :  

    show ?case

qed
Structural induction with \[ \Rightarrow \]

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

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

\begin{itemize}
  \item \text{case } 0 \\
  \quad \equiv \text{assume } 0: A(0) \\
  \quad \text{let } \ ?\text{case } = P(0) \\
  \quad \text{show } ?\text{case}
\end{itemize}

\text{next}

\begin{itemize}
  \item \text{case } (\text{Suc } n) \\
  \quad \equiv \text{assume } \text{Suc } 0: A(0) = \Rightarrow P(0) \\
  \quad \text{let } \ ?\text{case } = P(0) \\
  \quad \text{show } ?\text{case}
\end{itemize}

\text{qed}
Structural induction with \( \Rightarrow \)

\[
\begin{align*}
\text{show } & A(n) \Rightarrow P(n) \\
\text{proof (induction } n) & \\
\text{case } 0 & \equiv \text{ assume } 0: A(0) \\
: & \\
\text{show } ?\text{case} & \\
\text{let } ?\text{case} = P(0) \\
\text{next} & \\
\text{case } (Suc \ n) & \equiv \text{ fix } n \\
: & \\
\text{let } ?\text{case} = P(Suc \ n) \\
\text{show } ?\text{case} & \\
\text{qed}
\end{align*}
\]
Named assumptions

In a proof of

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

by structural induction:
Named assumptions

In a proof of

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

by structural induction:

In the context of

**case** \( C \)
Named assumptions

In a proof of
\[ A_1 \implies \ldots \implies A_n \implies B \]
by structural induction:

In the context of
\textbf{case } C

we have
\textbf{C.IH} the induction hypotheses
Named assumptions

In a proof of

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

by structural induction:

In the context of

\textbf{case } C

we have

\textbf{C.IH} the induction hypotheses
\textbf{C.prems} the premises \( A_i \)
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** \((\text{Suc } n) \ldots \text{show } ?\text{case}\)
  is easy to write and maintain
A remark on style

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

Rule Induction

Rule Inversion
Isar\_Induction\_Demo.thy

Rule induction
Rule induction

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

rule$_1$: 

:::

rule$_n$: 

show $I \; x \; y = \Rightarrow \; P \; x \; y$
proof (induction rule: $I$. induct)
case rule$_1$. . . 
?case next

next case rule$_n$. . . 
?case qed

208
Rule induction

**inductive** $I :: \tau \Rightarrow \sigma \Rightarrow \text{bool}$

**where**

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

$
\vdots
$

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

**show** $I \, x \, y \implies P \, x \, y$
**Rule induction**

**inductive** \( I :: \tau \Rightarrow \sigma \Rightarrow \text{bool} \)

**where**

\[ \text{rule}_1 : \ldots \]

\[ \vdots \]

\[ \text{rule}_n : \ldots \]

**show** \( I \ x \ y \ \Rightarrow \ P \ x \ y \)

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

\[ \text{inductive } I :: \tau \Rightarrow \sigma \Rightarrow \text{bool} \]

where

\[ \text{rule}_1: \ldots \]

\[ \vdots \]

\[ \text{rule}_n: \ldots \]

\[ \text{show } I \ x \ y \ \Rightarrow \ P \ x \ y \]

\[ \text{proof (induction rule: } I.\text{induct}) \]

\[ \text{case } \text{rule}_1 \]

\[ \ldots \]

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

next

\[ \vdots \]

\[ \vdots \]

\[ \text{case } \text{rule}_n \]

\[ \ldots \]

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

qed
Fixing your own variable names

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

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

In a proof of

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

by rule induction on \( I \ldots \):
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} \( R \)
Named assumptions

In a proof of

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

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

In the context of

\textit{case} \( R \)

we have

\( R.IH \) the induction hypotheses
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

\textbf{R.IH} the induction hypotheses

\textbf{R.hyps} the assumptions of rule \( R \)
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

\textbf{R.IH} the induction hypotheses

\textbf{R.hyps} the assumptions of rule \( R \)

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

Rule Induction

Rule Inversion
Rule inversion

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

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

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

What can we deduce from \textit{ev} \ n \ ?
Rule inversion

**inductive** \( ev :: \text{nat} \Rightarrow \text{bool} \) where

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

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

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

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

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

\( \text{ev } n \Rightarrow n = 0 \lor (\exists k. n = \text{Suc} (\text{Suc } k) \land \text{ev } k) \)
Rule inversion

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

\begin{itemize}
  \item \texttt{ev0}: \ ev \ 0 \\
  \item \texttt{evSS}: \ ev \ n \ \Rightarrow \ ev(\text{Suc}(\text{Suc} \ n))
\end{itemize}

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

\[ ev \ n \ \Rightarrow \ n = 0 \vee (\exists k. \ n = \text{Suc} (\text{Suc} \ k) \land ev \ k) \]

\textbf{Rule inversion = case distinction over rules}
Rule inversion
from 'ev n' have P
proof cases
  case ev0
  : show ?thesis ...
next
  case (evSS k)
  : show ?thesis ...
qed

n = 0

n = Suc (Suc k), ev k
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