Proof Assistants

Thomas Bauereiss Meven Lennon-Bertrand

Department of Computer Science and Technology
University of Cambridge

Michaelmas 2024

Part I

Isabelle

Material

- Isabelle part of this course based on book "Concrete Semantics with Isabelle/HOL" (2014) by Tobias Nipkow and Gerwin Klein

Material

- Isabelle part of this course based on book "Concrete Semantics with Isabelle/HOL" (2014) by Tobias Nipkow and Gerwin Klein
- Slides shamelessly copied from Tobias Nipkow (errors are my own)

Chapter 1

Programming and Proving in Isabelle/HOL

1. Overview of Isabelle/HOL

2. Type and function definitions

3. Induction Heuristics

4. Simplification

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification

HOL = Higher-Order Logic

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:

- Equalities ($term = term$)

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:

- Equalities ($term = term$), 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:

- Equalities ($term = term$), e.g. $1 + 2 = 4$
- Later: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\ldots$

1. Overview of Isabelle/HOL
   - Types and terms
   - By example: types $bool$, nat and $list$

Types

Basic syntax:

$$\tau \quad ::=$$

Types

Basic syntax:

$$\tau \quad ::= \quad (\tau)$$

Types

Basic syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \ldots & \text{base types}
\end{array}
$$

Types

Basic syntax:

$$
\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & bool \ | \ nat \ | \ int \ | \ \ldots & \text{base types} \\
& | & 'a \ | \ 'b \ | \ \ldots & \text{type variables}
\end{array}
$$

Types

Basic syntax:

$$\begin{array}{rcll}
\tau & ::= & (\tau) & \\
& | & bool \mid nat \mid int \mid \ldots & \text{base types} \\
& | & 'a \mid 'b \mid \ldots & \text{type variables} \\
& | & \tau \Rightarrow \tau & \text{functions}
\end{array}$$

Types

Basic syntax:

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

Types

Basic syntax:

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

Types

Basic syntax:

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

Types

Basic syntax:

τ	$::=$	(τ)	
	$\mid$	$bool \mid nat \mid int \mid \ldots$	base types
	$\mid$	$'a \mid 'b \mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	$\tau\ list$	lists
	$\mid$	$\tau\ set$	sets
	$\mid$	$\ldots$	user-defined types

Types

Basic syntax:

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

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

Terms

Terms can be formed as follows:

Terms

Terms can be formed as follows:

- *Function application:* $f\ t$

Terms

Terms can be formed as follows:

- *Function application:* $f\ t$
  is the call of function f with argument 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: $f\ t_1\ t_2\ \ldots$

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: $sin\ \pi$, $plus\ x\ y$

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:* $\lambda x.\ t$

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: $sin\ \pi$, $plus\ x\ y$
- *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: $sin\ \pi$, $plus\ x\ y$
- *Function abstraction:* $\lambda x.\ t$
  is the function with parameter x and result t,
  i.e. "$x \mapsto t$".
  Example: $\lambda x.\ plus\ x\ x$

Terms

Basic syntax:

$$t \quad ::=$$

Terms

Basic syntax:

$$t \quad ::= \quad (t)$$

Terms

Basic syntax:

$$\begin{array}{rcll} t & ::= & (t) & \\ & | & a & \text{constant or variable (identifier)} \end{array}$$

Terms

Basic syntax:

$$
\begin{array}{rcll}
t & ::= & (t) & \\
& | & a & \text{constant or variable (identifier)} \\
& | & t\ t & \text{function application}
\end{array}
$$

Terms

Basic syntax:

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

Terms

Basic syntax:

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

Terms

Basic syntax:

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

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

Terms

Basic syntax:

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

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

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

Terms

Basic syntax:

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

Examples: $f\ (g\ x)\ y$
$h\ (\lambda x.\ f\ (g\ x))$

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

Terms

Basic syntax:

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

Examples: $f\ (g\ x)\ y$
$h\ (\lambda x.\ f\ (g\ x))$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Terms must be well-typed

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 τ".

Terms must be well-typed

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

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

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

Type inference

Isabelle automatically computes the type of each variable in a term.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

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: $f\ (x{::}nat)$

`Overview_Demo.thy`

(including an example of how to define a simple function and prove a lemma about it)

1 Overview of Isabelle/HOL

Types and terms

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

Type $bool$

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

Type $bool$

datatype $bool = True \mid False$

Predefined functions:

$\wedge,\ \vee,\ \longrightarrow,\ \ldots\ :: bool \Rightarrow bool \Rightarrow bool$

Type $bool$

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

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

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

Type $bool$

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

Predefined functions:

$\wedge, \vee, \longrightarrow, \ldots \ :: bool \Rightarrow bool \Rightarrow bool$

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

if-and-only-if: $=$ or $\longleftrightarrow$

Type nat

datatype $nat = 0 \mid Suc\ nat$

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

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

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

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

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

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

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

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

You need type annotations: $1 :: nat$, $x + (y::nat)$

Type nat

datatype $nat = 0 \mid Suc\ nat$

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

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

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

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

Nat_Demo.thy

An informal proof

Lemma $add\ m\ 0 = m$

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 $add\ m\ 0 = m$
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).

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 $add\ m\ 0 = m$

Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:
  $add\ (Suc\ m)\ 0 \quad = \quad Suc\ (add\ m\ 0)$ by def. of add

An informal proof

Lemma $add\ m\ 0 = m$
Proof by induction on m.

- Case 0 (the base case):
  $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step):
  We assume $add\ m\ 0 = m$,
  the induction hypothesis (IH).
  We need to show $add\ (Suc\ m)\ 0 = Suc\ m$.
  The proof is as follows:
  $$\begin{array}{lcll} add\ (Suc\ m)\ 0 & = & Suc\ (add\ m\ 0) & \text{by def. of } add \\ & = & Suc\ m & \text{by IH} \end{array}$$

Induction on natural numbers

To prove $P(n)$ for all natural numbers n, prove

- $P(0)$ and
- for arbitrary but fixed n,
  $P(n)$ implies $P(Suc(n))$.

Induction on natural numbers

To prove $P(n)$ for all natural numbers n, prove

- $P(0)$ and
- for arbitrary but fixed n,
  $P(n)$ implies $P(Suc(n))$.

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

Type $'a\ list$

Lists of elements of type $'a$

Type $'a\ list$

Lists of elements of type $'a$

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

Type $'a\ list$

Lists of elements of type $'a$

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

Some lists: Nil,

Type $'a\ list$

Lists of elements of type $'a$

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

Some lists: Nil, $Cons\ 1\ Nil$,

Type $'a\ list$

Lists of elements of type $'a$

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

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

Type $'a\ list$

Lists of elements of type $'a$

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

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

Syntactic sugar:

- $[] = Nil$: empty list

Type $'a\ list$

Lists of elements of type $'a$

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

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

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"*)
- $[x_1, \ldots, x_n] = x_1\ \#\ \ldots\ x_n\ \#\ []$

Structural Induction for lists

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

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

Structural Induction for lists

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

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

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

List_Demo.thy

An informal proof

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

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

Large library: `HOL/List.thy`

Included in `Main`.

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: $xs \mathbin{@} ys$ (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

2 Type and function definitions

Type definitions

Function definitions

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples

type_synonym $string = char\ list$

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples

type_synonym $string = char\ list$

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

Type synonyms

type_synonym $name = \tau$

Introduces a *synonym* $name$ for type τ

Examples

type_synonym $string = char\ list$

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

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

datatype — the general case

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

datatype — the general case

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

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \dots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1,\dots,\alpha_n)t$

datatype — the general case

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

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1,\dots,\alpha_n)t$
- *Distinctness:* $C_i\ \dots \neq C_j\ \dots \qquad$ if $i \neq j$

datatype — the general case

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

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

datatype — the general case

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

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

Distinctness and injectivity are applied automatically

Induction must be applied explicitly

Case expressions

Datatype values can be taken apart with *case*:

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

Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards: _

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

Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards: _

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

Nested patterns:

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

Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards: _

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Case expressions

Datatype values can be taken apart with *case*:

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

Wildcards: _

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

The *option* type

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

The $option$ type

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

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

The $option$ type

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

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

Typical application:

fun $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**

The *option* type

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

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

Typical application:

fun $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**
$lookup\ []\ x = None \mid$

The $option$ type

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

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

Typical application:

fun $lookup :: ('a \times\ 'b)\ list \Rightarrow\ 'a \Rightarrow\ 'b\ option$ **where**
$lookup\ []\ x = None \mid$
$lookup\ ((a,\ b)\ \#\ ps)\ x =$

The $option$ type

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

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

Typical application:

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

2 Type and function definitions

Type definitions

Function definitions

Non-recursive definitions

Example

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

Non-recursive definitions

Example

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

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.

$\Longrightarrow 0 = 1$

The danger of nontermination

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

Subtract $f\,x$ on both sides.

$$\Longrightarrow 0 = 1$$

! 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

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

Example: Ackermann

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

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

Example: Ackermann

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

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

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

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

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:

fun $rev :: {}'a\ list \Rightarrow {}'a\ list$ **where**

$rev\ [] \quad = [] \quad |$

$rev\ (x\#xs) \quad = rev\ xs\ @\ [x]$

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

fun $itrev :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

A tail recursive reverse

Our initial reverse:

fun $rev :: 'a\ list \Rightarrow 'a\ list$ **where**

$$
\begin{aligned}
&rev\ [] &&= []\ \ | \\
&rev\ (x\#xs) &&= rev\ xs\ @\ [x]
\end{aligned}
$$

A tail recursive version:

fun $itrev :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$$
itrev\ []\qquad ys = ys\ \ |
$$

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

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

`Induction_Demo.thy`

Generalisation

Generalisation

- Replace constants by variables

Generalisation

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

So far, all proofs were by structural induction

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

Computation Induction

Example

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

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

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

Computation Induction

Example

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

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

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

Computation Induction

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

Computation Induction

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

for each defining equation

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

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

Computation Induction

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

for each defining equation

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

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

Induction follows course of (terminating!) computation

Computation Induction

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

for each defining equation

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

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

Induction follows course of (terminating!) computation

Motto: properties of f are best proved by rule $f.induct$

How to apply $f.induct$

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

How to apply $f.induct$

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

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

How to apply $f.induct$

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

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

Heuristic:

- there should be a call $f\ a_1\ \ldots\ a_n$ in your goal

How to apply $f.induct$

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

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

Heuristic:

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

Induction_Demo.thy

Computation Induction

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification

Simplification means . . .

Using equations $l = r$ from left to right

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 $\leadsto$ *simplification rule*

Simplification means ...

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *simplification rule*

Simplification = (Term) Rewriting

An example

Equations:

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

An example

Equations:

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

$$0 + Suc\ 0 \ \le \ Suc\ 0 + x$$

Rewriting:

An example

Equations:

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

$$\begin{array}{rcll} 0 + Suc\ 0 & \le & Suc\ 0 + x & \stackrel{(1)}{=} \\ Suc\ 0 & \le & Suc\ 0 + x & \end{array}$$

Rewriting:

An example

Equations:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

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

Conditional rewriting

Simplification rules can be conditional:

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Example

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Example

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

We can simplify $f(0)$ to $g(0)$

Conditional rewriting

Simplification rules can be conditional:

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

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

Example

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

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

Termination

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

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:

$$[\![\ P_1;\ \ldots;\ P_k\ ]\!] \Longrightarrow l = r$$

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:

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

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

$$n < m \Longrightarrow (n < Suc\ m) = True$$
$$Suc\ n < m \Longrightarrow (n < m) = True$$

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:

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

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

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

Proof method *simp*

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

apply$(simp\ add\colon\ eq_1\ \ldots\ \ eq_n)$

Proof method *simp*

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

apply$(simp\ add\colon\ eq_1\ \ldots\ \ eq_n)$

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

- lemmas with attribute $simp$

Proof method $simp$

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

apply$(simp\ add\colon\ eq_1\ \ldots\ eq_n)$

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

- lemmas with attribute $simp$
- rules from **fun** and **datatype**

Proof method *simp*

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

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

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

- lemmas with attribute $simp$
- rules from **fun** and **datatype**
- additional lemmas $eq_1\ \ldots\ eq_n$

Proof method *simp*

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

apply$(simp\ add\colon\ eq_1\ \ldots\ eq_n)$

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

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

Proof method *simp*

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

apply$(simp\ add\!: eq_1 \ \ldots \ eq_n)$

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

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

Variations:

- $(simp \ \ldots \ del\!: \ \ldots)$ removes $simp$-lemmas
- add and del are optional

$auto$ versus $simp$

- $auto$ acts on all subgoals
- $simp$ acts only on subgoal 1

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

Rewriting with definitions

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

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

Rewriting with definitions

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

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

f is the function whose definition is to be unfolded.

Case splitting with $simp/auto$

Automatic:

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

Case splitting with $simp/auto$

Automatic:

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

By hand:

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

Case splitting with *simp/auto*

Automatic:

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

By hand:

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

Proof method: $(simp\ split\colon nat.split)$

Case splitting with $simp/auto$

Automatic:

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

By hand:

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

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

Or $auto$.

Case splitting with *simp/auto*

Automatic:

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

By hand:

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

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

Simp_Demo.thy

Chapter 2

Case Study: IMP Expressions

5 Case Study: IMP Expressions

5 Case Study: IMP Expressions

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

This section introduces

arithmetic and boolean expressions

of our imperative language IMP.

IMP *commands* are introduced later.

5 Case Study: IMP Expressions

Arithmetic Expressions

Boolean Expressions

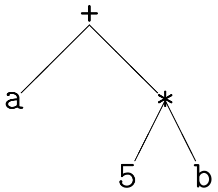
Concrete and abstract syntax

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

Concrete and abstract syntax

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

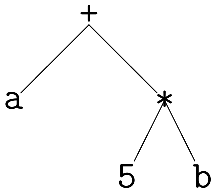
Abstract syntax: trees, eg



Concrete and abstract syntax

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

Abstract syntax: trees, eg

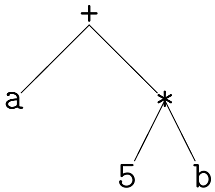


Parser: function from strings to trees

Concrete and abstract syntax

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

Abstract syntax: trees, eg



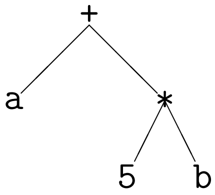
Parser: function from strings to trees

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

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

where n can be any natural number and x any variable.

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

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

where n can be any natural number and x any variable.

We focus on *abstract* syntax
which we introduce via datatypes.

Datatype $aexp$

Variable names are strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

Datatype $aexp$

Variable names are strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

Concrete	Abstract
5	$N\ 5$

Datatype $aexp$

Variable names are strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

Concrete	Abstract
5	$N\ 5$
x	$V\ ''x''$

Datatype $aexp$

Variable names are strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

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

Datatype $aexp$

Variable names are strings, values are integers:

type_synonym $vname = string$
datatype $aexp = N\ int \mid V\ vname \mid Plus\ aexp\ aexp$

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

Warning

This is syntax, not (yet) semantics!

Warning

This is syntax, not (yet) semantics!

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

The (program) state

What is the value of `x+1`?

The (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 *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 *state*.
- The state is a function from variable names to values:

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

Function update notation

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

$$f(a := b)$$

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.

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

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 defined in `AExp.thy`:

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

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 defined in `AExp.thy`:

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

Maps everything to 0, but $''a''$ to 5, $''x''$ to 3, etc.

AExp.thy

5. Case Study: IMP Expressions
   - Arithmetic Expressions
   - Boolean Expressions

`BExp.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 3

Logic and Proof Beyond Equality

6 Logical Formulas

7 Proof Automation

8 Single Step Proofs

9 Inductive Definitions

6 Logical Formulas

7 Proof Automation

8 Single Step Proofs

9 Inductive Definitions

Syntax (in decreasing precedence):

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

Syntax (in decreasing precedence):

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

Examples:

$$\neg\ A\ \wedge\ B \vee\ C\ \ \equiv\ \ ((\neg\ A)\ \wedge\ B)\ \vee\ C$$

Syntax (in decreasing precedence):

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

Examples:

$$\begin{array}{rcl} \neg\ A \wedge B \vee C & \equiv & ((\neg\ A) \wedge B) \vee C \\ s = t \wedge C & \equiv & (s = t) \wedge C \end{array}$$

Syntax (in decreasing precedence):

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

Examples:

$$
\begin{array}{rcl}
\neg\ A \wedge B \vee\ C & \equiv & ((\neg\ A)\ \wedge\ B)\ \vee\ C \\
s = t \wedge\ C & \equiv & (s = t)\ \wedge\ C \\
A\ \wedge\ B = B\ \wedge\ A & \equiv & A\ \wedge\ (B = B)\ \wedge\ A
\end{array}
$$

Syntax (in decreasing precedence):

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

Examples:

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

Syntax (in decreasing precedence):

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

Examples:

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

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

Variable binding convention:

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

Variable binding convention:

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

Similarly for $\exists$ and λ.

Warning

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

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

Mathematical symbols

and their ascii representations

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

Sets over type $'a$

$'a\ set$

Sets over type $'a$

$$'a\ set$$

- $\{\}, \quad \{e_1,\ldots,e_n\}$

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 $'a$

$$'a\ set$$

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

Sets over type $'a$

$'a\ set$

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

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

Set comprehension

- $\{x.\ P\}$ where x is a variable

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: $\{t\ |x\ y\ z.\ P\}$

Set comprehension

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

simp and *auto*

simp: rewriting and a bit of arithmetic

auto: rewriting and a bit of arithmetic, logic and sets

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.

fastforce

- 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

blast

- A complete proof search procedure for FOL ...

blast

- A complete proof search procedure for FOL ...
- ... but (almost) without "$=$"

blast

- 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

blast

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

Automating arithmetic

$arith$:

Automating arithmetic

$arith$:

- proves linear formulas (no "$*$")

Automating arithmetic

$arith$:

- proves linear formulas (no "$*$")
- complete for quantifier-free $real$ arithmetic

Automating arithmetic

$arith$:

- proves linear formulas (no "$*$")
- complete for quantifier-free $real$ arithmetic
- complete for first-order theory of nat and int (Presburger arithmetic)

Sledgehammer

Architecture:

$\boxed{\textbf{Isabelle}}$

external
ATPs[1]

[1] Automatic Theorem Provers

Architecture:

Isabelle

Goal
& filtered library $\downarrow$

external
ATPs[1]

---

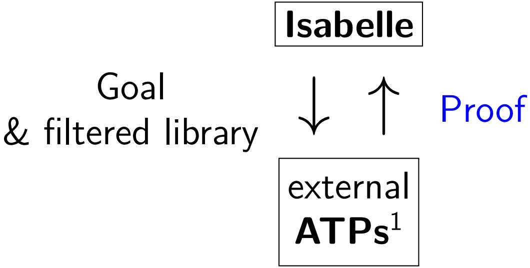
[1] Automatic Theorem Provers

Architecture:

$$\begin{array}{c} \boxed{\textbf{Isabelle}} \\ \text{Goal \& filtered library} \downarrow \quad \uparrow \text{Proof} \\ \boxed{\begin{array}{c}\text{external}\\ \textbf{ATPs}^{1}\end{array}} \end{array}$$

[1] Automatic Theorem Provers

Architecture:



Characteristics:

- Sometimes it works,

[1] Automatic Theorem Provers

Architecture:

$$\begin{array}{ccc} & \boxed{\textbf{Isabelle}} & \\ \begin{array}{c}\text{Goal}\\ \text{\& filtered library}\end{array} & \downarrow \quad \uparrow & \text{Proof} \\ & \boxed{\begin{array}{c}\text{external}\\ \textbf{ATPs}^{1}\end{array}} & \end{array}$$

Characteristics:

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

[1] Automatic Theorem Provers

Architecture:

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

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

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

Characteristics:

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

Do you feel lucky?

[1] Automatic Theorem Provers

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

$$\approx$$

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

Auto_Proof_Demo.thy

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

What are these *?-variables* ?

What are these *?-variables* ?

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

What are these *?-variables* ?

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

Example: theorem `conjI`: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

What are these *?-variables* ?

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

Example: theorem `conjI`: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

What are these *?-variables* ?

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

Example: theorem `conjI`: $[\![?P;\ ?Q]\!] \Longrightarrow\ ?P \wedge\ ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$

What are these *?-variables* ?

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

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

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\leadsto$
  $\llbracket a = b;\ False \rrbracket \Longrightarrow a = b \wedge False$

What are these *?-variables* ?

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

Example: theorem `conjI`: $[\![?P; \ ?Q]\!] \Longrightarrow ?P \wedge ?Q$

These ?-variables can later be instantiated:

- By hand:
  `conjI[of "a=b" "False"]` $\rightsquigarrow$
  $[\![a = b; \ False]\!] \Longrightarrow a = b \wedge False$
- By unification:
  unifying $?P \wedge ?Q$ with $a{=}b \wedge False$

What are these *?-variables* ?

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

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

These ?-variables can later be instantiated:

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

Rule application

Rule application

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

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ldots \Longrightarrow B$

Rule application

Example: rule: $\llbracket ?P;\ ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ldots \Longrightarrow B$

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

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ldots \Longrightarrow B$

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

- Unify A and C

Rule application

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: 1. $\ldots \Longrightarrow A \wedge B$

Result: 1. $\ldots \Longrightarrow A$

2. $\ldots \Longrightarrow B$

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

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

Rule application

Example: rule: $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$

subgoal: $1.\ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ldots \Longrightarrow B$

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

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

apply($rule\ xyz$)

Rule application

Example: rule: $[\![?P;\ ?Q]\!] \Longrightarrow\ ?P \wedge\ ?Q$

subgoal: $1.\ \ \ldots \Longrightarrow A \wedge B$

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

$2.\ \ \ldots \Longrightarrow B$

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

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

apply($rule\ xyz$)

"Backchaining"

Typical backwards rules

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

Typical backwards rules

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

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ \texttt{impI}$$

Typical backwards rules

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

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

Typical backwards rules

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

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

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

Typical backwards rules

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

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

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

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

Automating intro rules

Automating intro rules

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

$$(\mathit{blast\ intro}\colon r)$$

allows blast to backchain on r during proof search.

Automating intro rules

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

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

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

Example:

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

Automating intro rules

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

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

allows blast to backchain on r during proof search.

Example:

theorem $\mathit{le\_trans}$: $[\![?x \le ?y;\ ?y \le ?z]\!] \Longrightarrow ?x \le ?z$

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

Automating intro rules

If r is a theorem $\llbracket A_1; \ldots; A_n \rrbracket \Longrightarrow A$ then

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

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

Example:

theorem le_trans: $\llbracket\ ?x \le\ ?y;\ ?y \le\ ?z\ \rrbracket \Longrightarrow\ ?x \le\ ?z$

goal 1. $\llbracket\ a \le b;\ b \le c;\ c \le d\ \rrbracket \Longrightarrow a \le d$

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

Automating intro rules

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

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

allows blast to backchain on r during proof search.

Example:

theorem $\mathit{le\_trans}$: $[\![\, ?x \le ?y;\ ?y \le ?z \,]\!] \Longrightarrow ?x \le ?z$

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

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

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

Automating intro rules

If r is a theorem $\llbracket\ A_1; \ldots; A_n\ \rrbracket \Longrightarrow A$ then

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

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

Example:

theorem le_trans: $\llbracket\ ?x \le\ ?y;\ ?y \le\ ?z\ \rrbracket \Longrightarrow\ ?x \le\ ?z$
goal $1.\ \llbracket\ a \le b;\ b \le c;\ c \le d\ \rrbracket \Longrightarrow a \le d$
proof **apply**$(blast\ intro{:}\ le\_trans)$

Also works for $auto$ and $fastforce$

Can greatly increase the search space!

Forward proof: OF

If r is a theorem $A \Longrightarrow B$

Forward proof: OF

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

Forward proof: OF

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

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

Forward proof: OF

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

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

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

Forward proof: OF

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

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

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

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

Forward proof: OF

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

$$r[OF\ s]$$

is the theorem obtained by proving A with s.

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

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

$$\leadsto$$

$$?Q \Longrightarrow a = a \land ?Q$$

The general case:

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

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

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

The general case:

If r is a theorem $\llbracket\ A_1; \ldots; A_n\ \rrbracket \Longrightarrow A$
and $r_1, \ldots, r_m$ $(m{\leq}n)$ are theorems then

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

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

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

The general case:

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

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

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

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

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

The general case:

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

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

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

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

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

$$\rightsquigarrow$$

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

From now on: $?$ mostly suppressed on slides

`Single_Step_Demo.thy`

$\Longrightarrow$ versus $\longrightarrow$

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

$\Longrightarrow$ versus $\longrightarrow$

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

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

$\Longrightarrow$ versus $\longrightarrow$

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

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

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

Example: even numbers

Informally:

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:

inductive $ev :: nat \Rightarrow bool$

Example: even numbers

Informally:

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

In Isabelle/HOL:

inductive $ev :: nat \Rightarrow bool$
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 $ev :: nat \Rightarrow bool$
where

$$ev\ 0 \quad |$$
$$ev\ n \Longrightarrow ev\ (n + 2)$$

An easy proof: $ev\ 4$

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

Consider

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

Consider

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

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

Consider

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

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

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

Consider

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

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

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

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$

Consider

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

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

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

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow evn\ m = True$

Consider

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

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

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

Two cases: $ev\ m$ is proved by

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

Consider

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

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

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

Two cases: $ev\ m$ is proved by

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

Consider

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

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

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

Two cases: $ev\ m$ is proved by

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

Rule induction for ev

To prove

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

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

Rule induction for ev

To prove

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

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

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

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

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

Rule `ev.induct`:

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

Format of inductive definitions

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

Format of inductive definitions

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

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

Format of inductive definitions

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

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

$\vdots$

Format of inductive definitions

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

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

$\vdots$

Note:

- I may have multiple arguments.

Format of inductive definitions

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

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

$\vdots$

Note:

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

Rule induction in general

To prove

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

by *rule induction* on $I\ x$

Rule induction in general

To prove

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

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

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

that P is preserved:

Rule induction in general

To prove

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

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

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

that P is preserved:

$$\llbracket\ I\ a_1;\ P\ a_1;\ \ldots\ ;\ I\ a_n;\ P\ a_n\ \rrbracket \Longrightarrow P\ a$$

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

Inductive_Demo.thy

Inductively defined sets

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

Inductively defined sets

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

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

Inductively defined sets

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

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

$\vdots$

Inductively defined sets

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

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

$\vdots$

Difference to **inductive**: I can later be used with set theoretic operators, eg $I \cup \ldots$

Chapter 4

Isar: A Language for Structured Proofs

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

A typical Isar proof

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

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

Isar core syntax

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

Isar core syntax

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

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

Isar core syntax

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

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

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

Isar core syntax

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

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

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

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

Isar core syntax

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

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

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

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

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

`Isar_Demo.thy`

Isar by example

Further reading

- More detailed Isar introduction in Chapter 5 of "Concrete Semantics"
- Isabelle/Isar reference manual (`isar-ref.pdf`), in particular Chapter 6