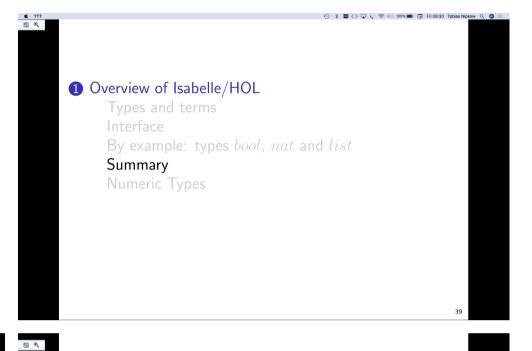
Script generated by TTT

Title: FDS (03.05.2019)

Date: Fri May 03 08:30:34 CEST 2019

Duration: 89:08 min

Pages: 101



•

- 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):

"=" is used only from left to right!

Proofs

General schema:

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

42

(

Proofs

General schema:

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

If the lemma is suitable as a simplification rule:

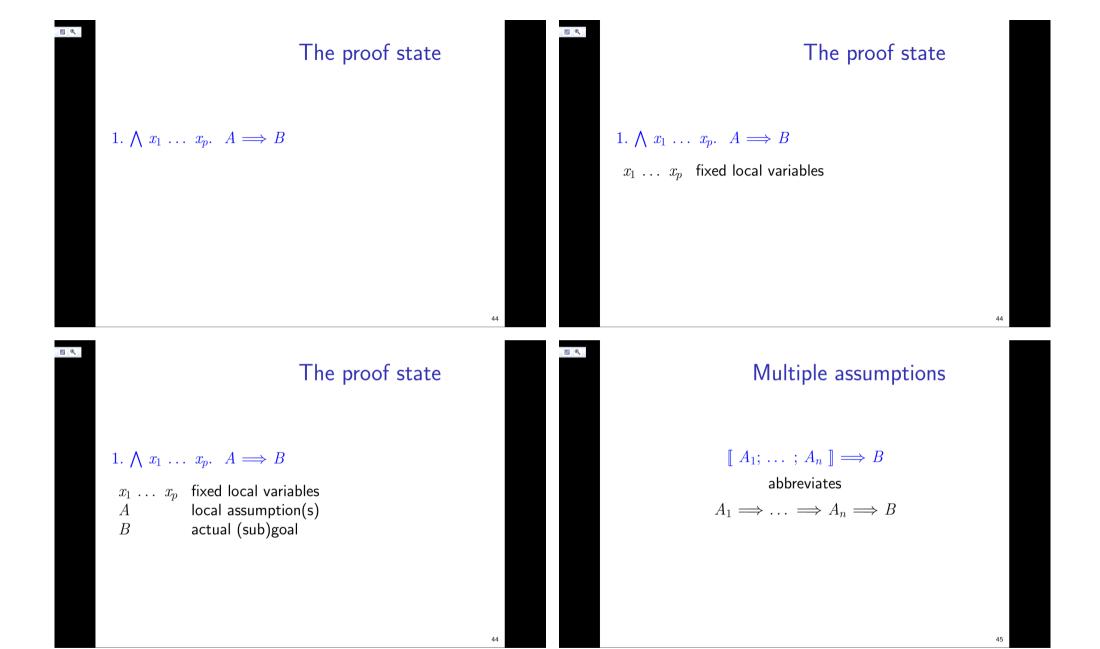
```
lemma name[simp]: "..."
```

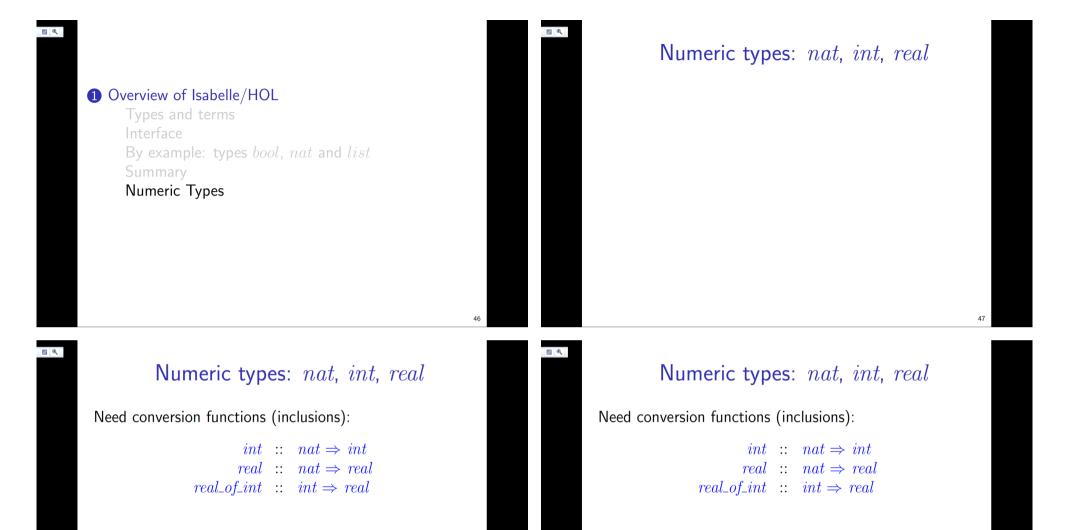
Top down proofs

Command

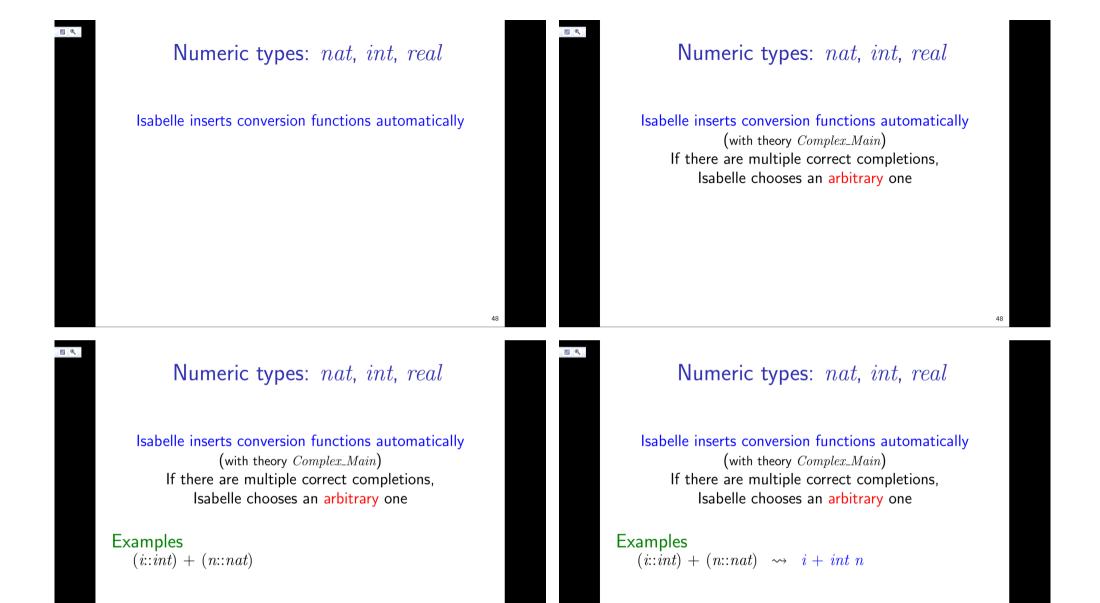
sorry

"completes" any proof.





 $\label{eq:local_local} \text{If you need type } \textit{real}, \\ \text{import theory } \textit{Complex_Main} \text{ instead of } \textit{\underline{Main}} \\$





Numeric types: nat, int, real

Isabelle inserts conversion functions automatically

(with theory Complex_Main)

If there are multiple correct completions, Isabelle chooses an arbitrary one

Examples

```
(i::int) + (n::nat) \rightarrow i + int n
((n::nat) + n) :: real
```

E Q

Numeric types: nat, int, real

Isabelle inserts conversion functions automatically

(with theory *Complex_Main*)

If there are multiple correct completions, Isabelle chooses an arbitrary one

Examples

```
(i::nt) + (n::nat) \implies i + int \ n
((n::nat) + n) :: real \implies real(n+n), real \ n + real \ n
```

48

Q

Numeric types: nat, int, real

Coercion in the other direction:

$$nat :: int \Rightarrow nat$$

(1)

Numeric types: nat, int, real

Coercion in the other direction:

$$nat :: int \Rightarrow nat$$

E

Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

E Q

Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

 $- :: 'a \Rightarrow 'a$

EΟ

1 0

Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

 $- :: 'a \Rightarrow 'a$

• Division on *nat* and *int*:

$$div, mod :: 'a \Rightarrow 'a \Rightarrow 'a$$

8

Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

 $- :: 'a \Rightarrow 'a$

• Division on *nat* and *int*:

$$div, mod :: 'a \Rightarrow 'a \Rightarrow 'a$$

- Division on real: $/:: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with nat: $a \Rightarrow nat \Rightarrow a$



Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

 $- :: 'a \Rightarrow 'a$

• Division on nat and int: $div. mod :: 'a \Rightarrow 'a \Rightarrow 'a$

• Division on real:
$$/::'a \Rightarrow 'a \Rightarrow 'a$$

- Exponentiation with nat: $a \Rightarrow nat \Rightarrow a$
- Exponentiation with real: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$

E

Overloaded arithmetic operations

• Basic arithmetic functions are overloaded:

$$+, -, * :: 'a \Rightarrow 'a \Rightarrow 'a$$

 $- :: 'a \Rightarrow 'a$

• Division on *nat* and *int*:

$$div, mod :: 'a \Rightarrow 'a \Rightarrow 'a$$

- Division on real: $/:: 'a \Rightarrow 'a \Rightarrow 'a$
- Exponentiation with nat: $\hat{}$:: $'a \Rightarrow nat \Rightarrow 'a$
- Exponentiation with real: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$
- Absolute value: $abs :: 'a \Rightarrow 'a$

Above all binary operators are infix

50

•

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

(

datatype — the general case

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

• •

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype} \ (\alpha_1,\dots,\alpha_n)t &=& C_1 \ \tau_{1,1}\dots\tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1}\dots\tau_{k,n_k} \end{array}$$

• 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
$$(\alpha_1,\ldots,\alpha_n)t=C_1\ au_{1,1}\ldots au_{1,n_1}$$
 $|\ \ldots\ |\ C_k\ au_{k,1}\ldots au_{k,n_k}$

- 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 $i \neq j$

53

9

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype} \ (\alpha_1,\dots,\alpha_n)t &=& C_1 \ \tau_{1,1}\dots\tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1}\dots\tau_{k,n_k} \end{array}$$

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

(

datatype — the general case

$$\begin{array}{lcl} \textbf{datatype} \ (\alpha_1,\dots,\alpha_n)t &=& C_1 \ \tau_{1,1}\dots\tau_{1,n_1} \\ & | & \dots \\ & | & C_k \ \tau_{k,1}\dots\tau_{k,n_k} \end{array}$$

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

5

(

datatype — the general case

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

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

Distinctness and injectivity are applied automatically Induction must be applied explicitly

E Q

Case expressions

Like in functional languages:

(case t of
$$pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n$$
)

54

(

Case expressions

Like in functional languages:

(case t of
$$pat_1 \Rightarrow t_1 \mid \dots \mid pat_n \Rightarrow t_n$$
)

Complicated patterns mean complicated proofs!

8

Case expressions

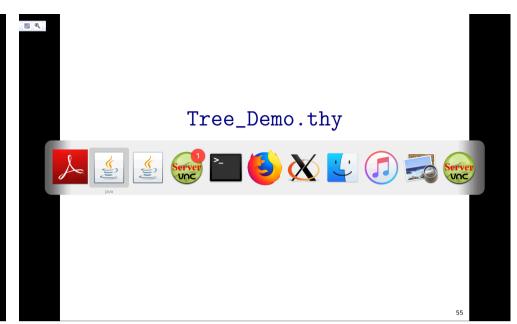
Like in functional languages:

(case t of
$$pat_1 \Rightarrow t_1 \mid \dots \mid pat_n \Rightarrow t_n$$
)

Complicated patterns mean complicated proofs!

Need () in context





The option type datatype 'a $option = None \mid Some \ 'a$ If 'a has values $a_1, \ a_2, \ldots$ then 'a option has values $None, \ Some \ a_1, \ Some \ a_2, \ldots$

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 $lookup [] \ x = None \ |$ $lookup \ ((a, b) \# ps) \ x =$

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 , ...

Typical application:

fun $lookup :: ('a \times 'b) \ list \Rightarrow 'a \Rightarrow 'b \ option$ where $lookup \ [] \ x = None \ |$ $lookup \ ((a, b) \# ps) \ x =$ $(if \ a = x \ then \ Some \ b \ else \ lookup \ ps \ x)$

56

9

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.

$$\implies 0 = 1$$

Key features of **fun**

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

• Pattern-matching over datatype constructors



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

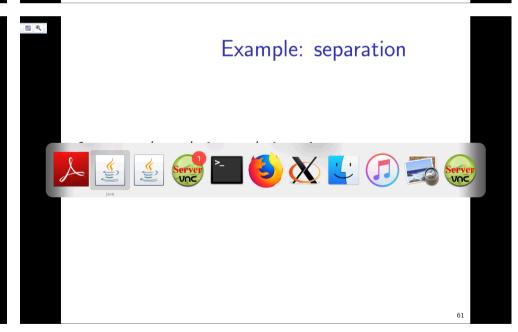
- Pattern-matching over datatype constructors
- Order of equations matters
- Termination must be provable automatically by size measures
- Proves customized induction schema

60

•

Example: separation

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





Basic induction heuristics

Theorems about recursive functions are proved by induction

15

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

64

9

A tail recursive reverse

Our initial reverse:

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

E

A tail recursive reverse

Our initial reverse:

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

A tail recursive version:

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

6



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** $itrev \mid$ $ys = ys \mid$

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 $itrev \ [] \qquad ys = ys \ |$ $itrev \ (x\#xs) \quad ys =$

65

9

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 $itrev \ [] \qquad ys = ys \ |$ $itrev \ (x\#xs) \quad ys = itrev \ xs \ (x\#ys)$

E

A tail recursive reverse

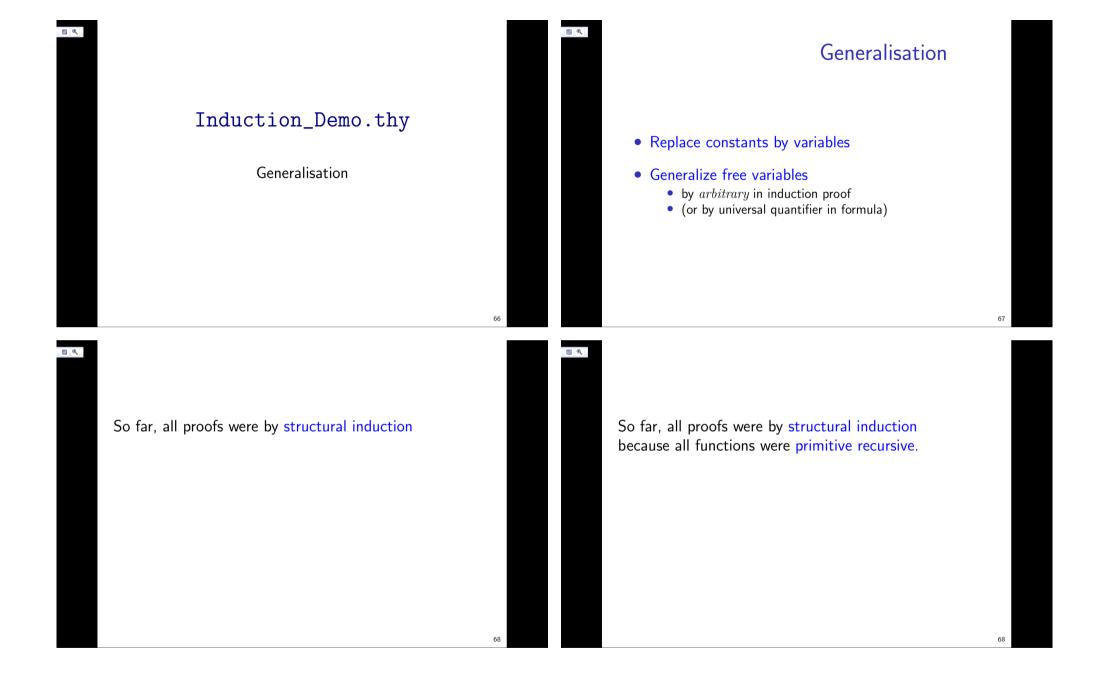
Our initial reverse:

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

A tail recursive version:

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

lemma itrev xs [] = rev xs



E

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 $div2 \ 0 = 0 \mid$ $div2 \ (Suc \ 0) = 0 \mid$ $div2 \ (Suc(Suc \ n)) = Suc(div2 \ n)$

→ induction rule div2.induct:

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

9

Computation Induction

Example

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

→ induction rule div2.induct:

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

E Q

Computation Induction

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

1 0

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

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

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

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

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

Induction follows course of (terminating!) computation

70

0

Computation Induction

Example

fun $div2 :: nat \Rightarrow nat$ **where** $div2 \ 0 = 0$

$$div2 (Suc 0) = 0 \mid div2 (Suc (Suc n)) = Suc (div2 n)$$

→ induction rule div2.induct:

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

1 Q

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

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

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

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

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

71

•

How to apply f.induct

If $f :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau'$: $(induction \ a_1 \ \dots \ a_n \ rule: f.induct)$

(

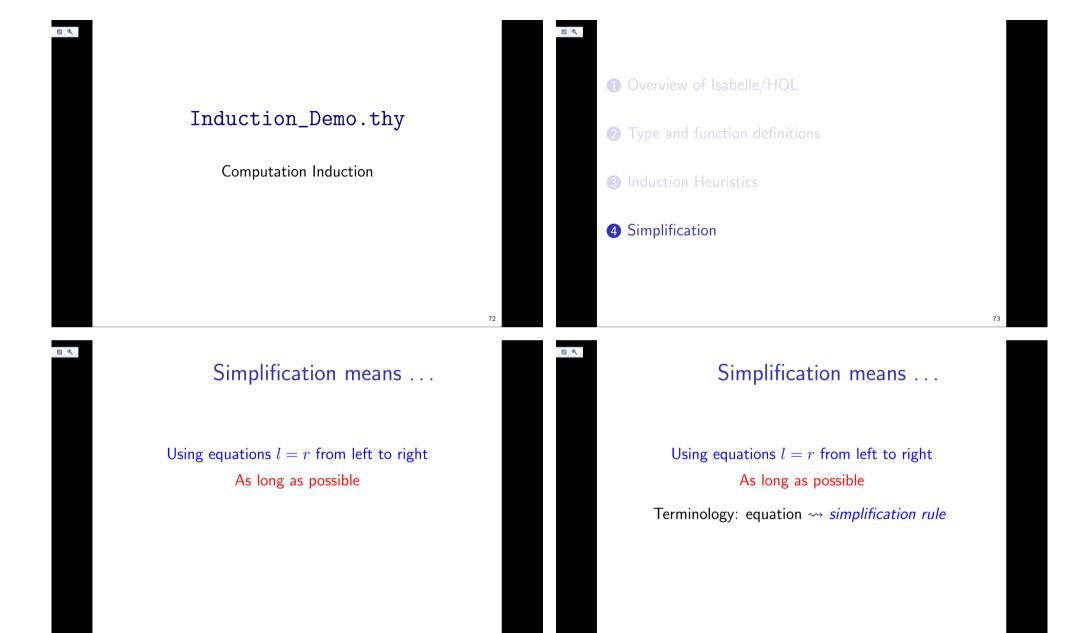
How to apply f.induct

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

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

Heuristic:

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



An example

$$0 + n = n \tag{1}$$

Equations:
$$(Suc m) + n = Suc (m+n)$$
 (2)

$$(Suc \ m \le Suc \ n) = (m \le n)$$

$$(0 \le m) = True$$

$$(4)$$

An example

$$0 + n = n \tag{1}$$

$$(0 \le m) = True \tag{4}$$

$$0 + Suc 0 < Suc 0 + x$$

Rewriting:

An example

$$0 + n = n \tag{1}$$

Equations:
$$(Suc \ m) + n = Suc \ (m+n) (2)$$
$$(Suc \ m \le Suc \ n) = (m \le n) (3)$$

$$\leq Suc\ n) = (m \leq n) \tag{3}$$

$$(0 \le m) = True \tag{4}$$

$$0 + Suc \ 0 \le Suc \ 0 + x \stackrel{(1)}{=}$$

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

Rewriting:

An example

$$0 + n = n \tag{1}$$

Equations:
$$(Suc \ m) + n = Suc \ (m+n) \ (2)$$

(Suc
$$m \le Suc \ n$$
) = $(m \le n)$ (3)

$$(0 \le m) = True \tag{4}$$

$$0 + Suc \ 0 \le Suc \ 0 + x \stackrel{(1)}{=}$$

$$Suc 0 \leq Suc 0 + x \stackrel{(2)}{=}$$

Rewriting:
$$Suc \ 0 \le Suc \ (0+x)$$

1 Q

An example

$$0 + n = n \tag{1}$$

Equations:
$$(Suc m) + n = Suc (m+n) (2)$$

$$(Suc \ m \le Suc \ n) = (m \le n) \tag{3}$$

$$(0 \le m) = True \tag{4}$$

$$0 + Suc \ 0 \le Suc \ 0 + x \stackrel{\text{(1)}}{=}$$

$$Suc \ 0 \le Suc \ 0 + x \stackrel{\text{(2)}}{=}$$

Rewriting:
$$Suc 0 \leq Suc (0+x) \stackrel{(3)}{=}$$

$$0 \leq 0 + x$$

An example

$$0 + n = n \tag{1}$$

$$(0 \le m) = True$$
 (4)

$$0 + Suc \ 0 \ \le \ Suc \ 0 + x \qquad \stackrel{(1)}{=}$$

Suc 0 < Suc 0 + x

Rewriting:
$$Suc \ 0 \le Suc \ (0+x) = 0$$

$$0 \le 0 + x \qquad \stackrel{(4)}{=}$$

$$True$$

75

(4)

Conditional rewriting

Simplification rules can be conditional:

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

E

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.

7

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

$$p(0) = True$$

 $p(x) \Longrightarrow f(x) = g(x)$

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

$$p(0) = True$$

$$p(x) \Longrightarrow f(x) = g(x)$$

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

.

1 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

$$p(0) = True$$

$$p(x) \Longrightarrow f(x) = g(x)$$

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

8

Termination

(

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:

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

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

77

(e

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

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

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

E

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

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

-

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