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

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.

The proof state

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

The proof state

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

$x_1 \ \ldots \ x_p$ fixed local variables

The proof state

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

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

Multiple assumptions

$$\llbracket \ A_1; \ \ldots \ ; \ A_n \ \rrbracket \Longrightarrow B$$

abbreviates

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

Numeric types: *nat*, *int*, *real*

Numeric types: *nat*, *int*, *real*

Need conversion functions (inclusions):

$$\begin{array}{rcl} int & :: & nat \Rightarrow int \\ real & :: & nat \Rightarrow real \\ real\_of\_int & :: & int \Rightarrow real \end{array}$$

Numeric types: *nat*, *int*, *real*

Need conversion functions (inclusions):

$$\begin{array}{rcl} int & :: & nat \Rightarrow int \\ real & :: & nat \Rightarrow real \\ real\_of\_int & :: & int \Rightarrow real \end{array}$$

If you need type *real*,
import theory *Complex_Main* instead of *Main*

Numeric types: *nat*, *int*, *real*

Isabelle inserts conversion functions automatically

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

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

$(i::int) + (n::nat)$

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

$(i::int) + (n::nat) \quad \rightsquigarrow \quad i + int\ n$

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

$(i{::}int) + (n{::}nat) \;\rightsquigarrow\; i + int\ n$

$((n{::}nat) + n) :: real$

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

$(i{::}int) + (n{::}nat) \;\rightsquigarrow\; i + int\ n$

$((n{::}nat) + n) :: real \;\rightsquigarrow\; real(n{+}n),\ real\ n + real\ n$

Numeric types: nat, int, $real$

Coercion in the other direction:

$$nat \;::\; int \Rightarrow nat$$

Numeric types: nat, int, $real$

Coercion in the other direction:

$$nat \;::\; int \Rightarrow nat$$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$

Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  $+,\ -,\ * :: 'a \Rightarrow 'a \Rightarrow 'a$
  $- :: 'a \Rightarrow 'a$
- Division on *nat* and *int*:
  $div,\ mod :: 'a \Rightarrow 'a \Rightarrow 'a$

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*: $\hat{} :: 'a \Rightarrow nat \Rightarrow 'a$

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 $real$: $powr :: 'a \Rightarrow 'a \Rightarrow 'a$

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

datatype — the general case

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

datatype — the general case

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

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

datatype — the general case

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

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

datatype — the general case

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

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

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

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

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

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

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Like in functional languages:

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

Case expressions

Like in functional languages:

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

Complicated patterns mean complicated proofs!

Case expressions

Like in functional languages:

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

Complicated patterns mean complicated proofs!

Need ( ) in context

Tree_Demo.thy

Tree_Demo.thy

The *option* type

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

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

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

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**
$lookup\ []\ x = None \mid$
$lookup\ ((a,\ b)\ \#\ ps)\ x =$
$\quad (\mathit{if}\ a = x\ \mathit{then}\ Some\ b\ \mathit{else}\ lookup\ ps\ x)$

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
- 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: separation

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

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**

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**
$itrev\ []\quad ys = ys\ \mid$

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**
$itrev\ []\quad ys = ys\ \mid$
$itrev\ (x\#xs)\quad ys =$

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**
$itrev\ []\quad ys = ys\ \mid$
$itrev\ (x\#xs)\quad ys = itrev\ xs\ (x\#ys)$

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**
$itrev\ []\quad ys = ys\ \mid$
$itrev\ (x\#xs)\quad ys = itrev\ xs\ (x\#ys)$

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

Induction_Demo.thy

Generalisation

Generalisation

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

So far, all proofs were by structural induction

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

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**
$div2\ 0 = 0\ \ |$
$div2\ (Suc\ 0) = 0\ \ |$
$div2\ (Suc(Suc\ n)) = Suc(div2\ n)$

$\rightsquigarrow$ 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

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 $x :: \tau$:

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

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

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

`Induction_Demo.thy`

Computation Induction

Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Simplification means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *simplification rule*

An example

Equations:

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

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:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

$$
\begin{aligned}
0 + Suc\ 0 &\le Suc\ 0 + x & \stackrel{(1)}{=}\\
Suc\ 0 &\le Suc\ 0 + x & \stackrel{(2)}{=}\\
Suc\ 0 &\le Suc\ (0 + x) &
\end{aligned}
$$

An example

Equations:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

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

Conditional rewriting

Simplification rules can be conditional:

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

Conditional rewriting

Simplification rules can be conditional:

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

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

Conditional rewriting

Simplification rules can be conditional:

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

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:

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

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:

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

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:

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

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

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

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

$n < m \Longrightarrow (n < Suc\ m) = True$ YES
$Suc\ n < m \Longrightarrow (n < m) = True$ NO

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

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

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

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

77