Script **generated by TTT**

Title: FDS (05.05.2017)

Date: Fri May 05 08:30:40 CEST 2017

Duration: 95:57 min

Pages: 95

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

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

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

datatype — the general case

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

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

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

Distinctness and injectivity are applied automatically
Induction must be applied explicitly

Case expressions

Like in functional languages:

$$(\mathit{case}\ t\ \mathit{of}\ \ \mathit{pat}_1 \Rightarrow t_1 \mid \dots \mid \mathit{pat}_n \Rightarrow t_n)$$

Case expressions

Like in functional languages:

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

Complicated patterns mean complicated proofs!

Case expressions

Like in functional languages:

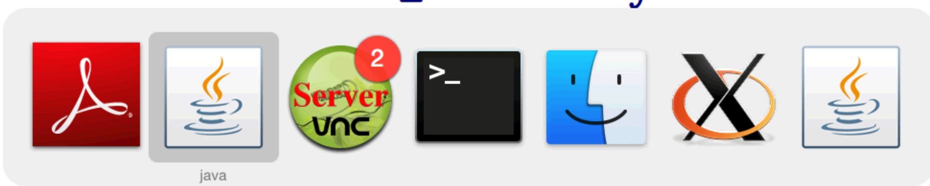
$$(case\ t\ of\ \ pat_1 \Rightarrow t_1 \ \mid \ \ldots \ \mid \ pat_n \Rightarrow t_n)$$

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

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 $f\ x_1\ \ldots\ x_n \ = \ \ldots$

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.
$\Longrightarrow 0 = 1$

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**
$sep\ a\ (x\#y\#zs) = x\ \#\ a\ \#\ sep\ a\ (y\#zs)\ \ |$
$sep\ a\ xs = xs$

primrec

- A restrictive version of **fun**
- Means *primitive recursive*
- Most functions are primitive recursive

primrec

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

primrec

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

The essence of primitive recursion:

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

primrec

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

The essence of primitive recursion:

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

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

primrec

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

The essence of primitive recursion:

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

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

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

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

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:

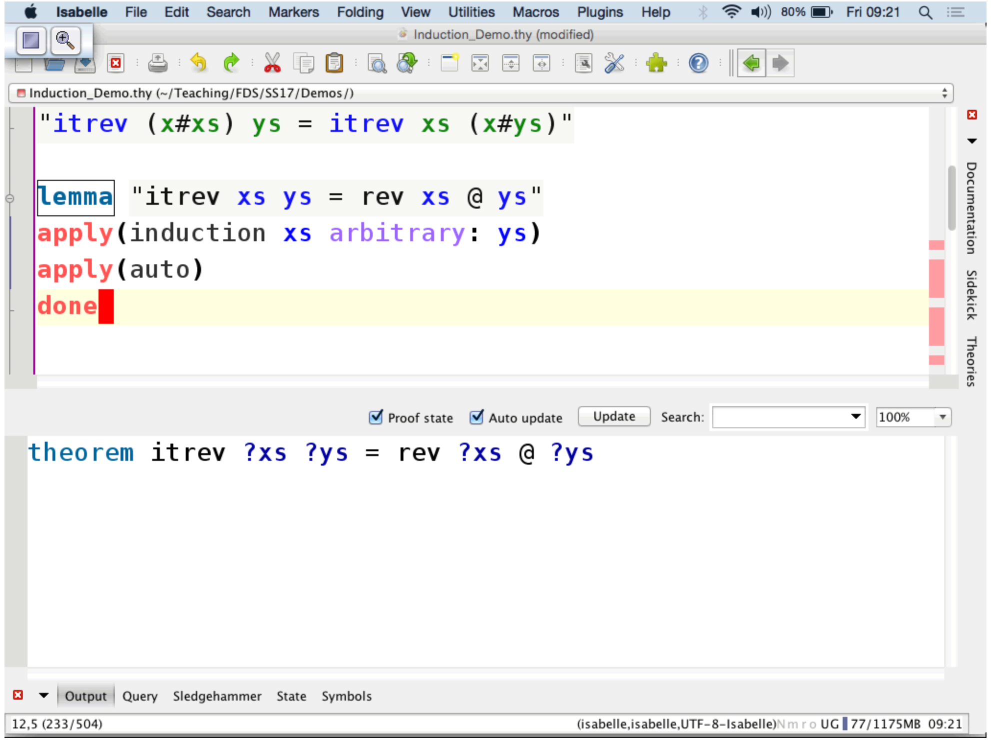
fun $rev :: {}'a\ list \Rightarrow {}'a\ list$ **where**
$rev\ [] = []\ \mid$
$rev\ (x\#xs) = rev\ xs\ @\ [x]$

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



So far, all proofs were by structural induction

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`:

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

Computation Induction

Example

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

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

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

Computation Induction

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

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

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

Heuristic:

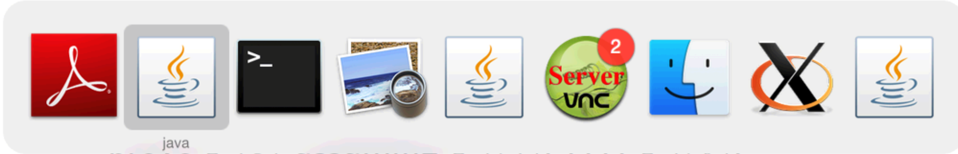
- there should be a call $f\ a_1\ \ldots\ a_n$ in your goal
- ideally the a_i should be variables.

Induction_Demo.thy

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



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

How to apply $f.induct$

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

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 $\leadsto$ *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:

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

$$0 + Suc\ 0 \;\leq\; Suc\ 0 + x$$

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:

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

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:

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

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:

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

Rewriting:

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

Conditional rewriting

Simplification rules can be conditional:

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

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

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

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

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

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

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

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

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

Termination

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

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

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

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

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

Proof method *simp*

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

apply(*simp add: eq_1 ... eq_n*)

Proof method *simp*

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

apply(*simp add: eq_1 ... eq_n*)

Simplify P_1 ... P_m and C using

- lemmas with attribute *simp*

Proof method *simp*

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

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

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

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

Proof method *simp*

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

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 $P_1\ \ldots\ P_m$

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* can also be modified:
  $(\mathit{auto\ simp\ add}{:}\ \ldots\ \mathit{simp\ del}{:}\ \ldots)$

Rewriting with definitions

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

$$(\mathit{simp\ add}{:}\ \mathit{f\_def}\ \ldots)$$

Case splitting with *simp*/*auto*

Automatic:

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

Case splitting with *simp*/*auto*

Automatic:

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

By hand:

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

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:

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

Proof method: *(simp split: 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:

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

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

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

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

Splitting pairs with *simp/auto*

How to replace

$$P\ (\textit{let } (x,\ y) = t \textit{ in } u\ x\ y)$$

Splitting pairs with *simp/auto*

How to replace

$$P\ (let\ (x,\ y) = t\ in\ u\ x\ y)$$
or
$$P\ (case\ t\ of\ (x,\ y) \Rightarrow u\ x\ y)$$

Splitting pairs with *simp/auto*

How to replace

$$P\ (let\ (x,\ y) = t\ in\ u\ x\ y)$$
or
$$P\ (case\ t\ of\ (x,\ y) \Rightarrow u\ x\ y)$$
by
$$\forall x\ y.\ t = (x,\ y) \longrightarrow P\ (u\ x\ y)$$

Splitting pairs with *simp/auto*

How to replace

$$P\ (let\ (x,\ y) = t\ in\ u\ x\ y)$$
or
$$P\ (case\ t\ of\ (x,\ y) \Rightarrow u\ x\ y)$$
by
$$\forall x\ y.\ t = (x,\ y) \longrightarrow P\ (u\ x\ y)$$

Proof method: *(simp split: prod.split)*

`Simp_Demo.thy`