Script **generated by TTT**

Title: FDS (27.04.2018)

Date: Fri Apr 27 08:33:08 CEST 2018

Duration: 85:01 min

Pages: 72

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:

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

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:

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

Rewriting:

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

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:

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

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:

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

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:

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

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

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

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

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

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:

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

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:

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

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

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

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

Proof method *simp*

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

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

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*

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**

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:* $eq_1 \ \ldots \ eq_n$)

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

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 $eq_1 \ \ldots \ eq_n$
- assumptions $P_1 \ \ldots \ P_m$

Variations:

- (*simp* ... *del:* ...) 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:
  $(auto\ simp\ add\colon \ldots\ simp\ del\colon \ldots)$

auto versus *simp*

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

Rewriting with definitions

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

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

Case splitting with *simp/auto*

Automatic:

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

Case splitting with *simp/auto*

Automatic:

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

By hand:

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

Case splitting with *simp/auto*

Automatic:

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

By hand:

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

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

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: $t.split$

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

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

or

$$P\ (\textit{case}\ t\ \textit{of}\ (x,\ y) \Rightarrow u\ x\ y)$$

Splitting pairs with $simp/auto$

How to replace

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

or

$$P\ (\textit{case}\ t\ \textit{of}\ (x,\ y) \Rightarrow u\ x\ y)$$

by

$$\forall x\ y.\ t = (x,\ y) \longrightarrow P\ (u\ x\ y)$$

Splitting pairs with *simp/auto*

How to replace

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

or

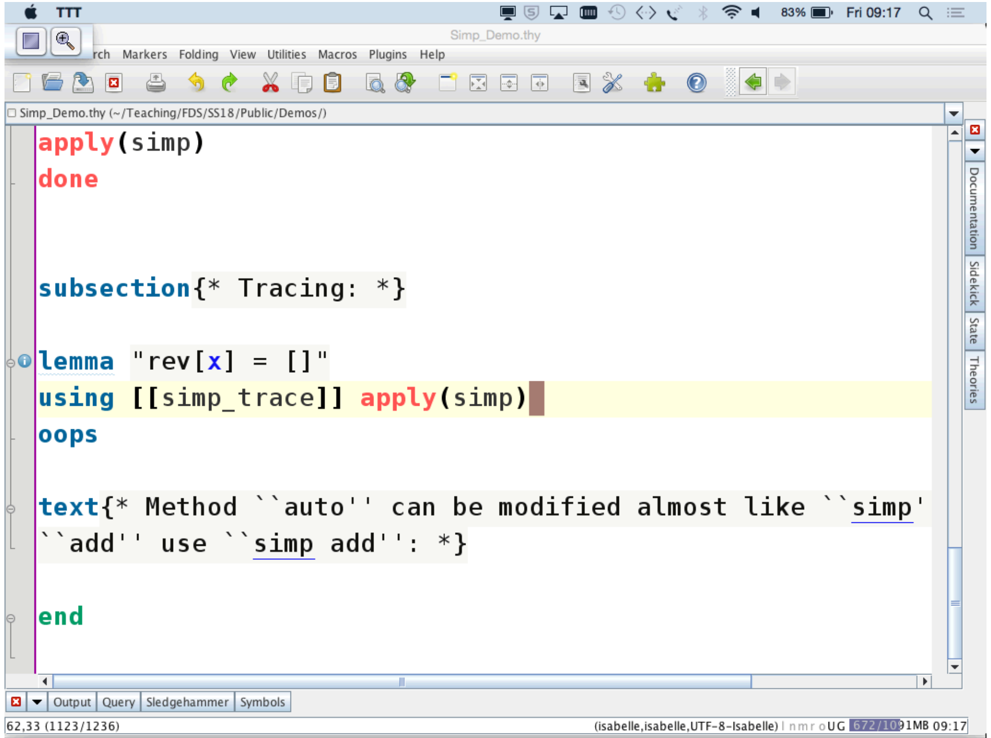
$$P\ (\mathit{case}\ t\ \mathit{of}\ (x,\ y) \Rightarrow u\ x\ y)$$

by

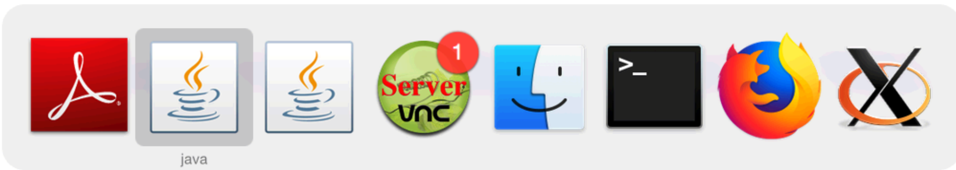
$$\forall\, x\ y.\ t = (x,\ y) \longrightarrow P\ (u\ x\ y)$$

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

Simp_Demo.thy



Chapter 3



Preview: sets

Type: $'a\ set$

Operations: $a \in A$, $A \cup B$, ...

Preview: sets

Type: $'a\ set$

Operations: $a \in A$, $A \cup B$, ...

Bounded quantification: $\forall\, a{\in}A.\ P$

Preview: sets

Type: $'a\ set$

Operations: $a \in A$, $A \cup B$, ...

Bounded quantification: $\forall\, a{\in}A.\ P$

Proof method *auto* knows (a little) about sets.

The (binary) tree library

```
imports "HOL-Library.Tree"
```

The (binary) tree library

imports "HOL-Library.Tree"
(File: `isabelle/src/HOL/Library/Tree.thy`)

The (binary) tree library

imports "HOL-Library.Tree"
(File: `isabelle/src/HOL/Library/Tree.thy`)

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

The (binary) tree library

imports "HOL-Library.Tree"
(File: `isabelle/src/HOL/Library/Tree.thy`)

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations:

$$\langle\rangle \equiv Leaf$$

The (binary) tree library

imports "HOL-Library.Tree"
(File: `isabelle/src/HOL/Library/Tree.thy`)

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

Abbreviations:

$$\langle\rangle \equiv Leaf$$

$$\langle l,\ a,\ r\rangle \equiv Node\ l\ a\ r$$

The (binary) tree library

Size = number of nodes:

$size :: {'a}\ tree \Rightarrow nat$

The (binary) tree library

Size = number of nodes:

$size :: {'a}\ tree \Rightarrow nat$

$size\ \langle\rangle = 0$

$size\ \langle l, \_, r\rangle = size\ l + size\ r + 1$

The (binary) tree library

Size = number of nodes:

$size :: {'a}\ tree \Rightarrow nat$

$size\ \langle\rangle = 0$

$size\ \langle l, \_, r\rangle = size\ l + size\ r + 1$

Height:

$height :: {'a}\ tree \Rightarrow nat$

$height\ \langle\rangle = 0$

$height\ \langle l, \_, r\rangle = max\ (height\ l)\ (height\ r) + 1$

The (binary) tree library

The set of elements in a tree:

$set\_tree :: {'a}\ tree \Rightarrow {'a}\ set$

The (binary) tree library

The set of elements in a tree:

$set\_tree :: 'a\ tree \Rightarrow 'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ a,\ r\rangle = set\_tree\ l \cup \{a\} \cup set\_tree\ r$

The (binary) tree library

The set of elements in a tree:

$set\_tree :: 'a\ tree \Rightarrow 'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ a,\ r\rangle = set\_tree\ l \cup \{a\} \cup set\_tree\ r$

Inorder listing:

$inorder :: 'a\ tree \Rightarrow 'a\ list$

The (binary) tree library

The set of elements in a tree:

$set\_tree :: 'a\ tree \Rightarrow 'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ a,\ r\rangle = set\_tree\ l \cup \{a\} \cup set\_tree\ r$

Inorder listing:

$inorder :: 'a\ tree \Rightarrow 'a\ list$

$inorder\ \langle\rangle = []$
$inorder\ \langle l,\ x,\ r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

The (binary) tree library

Binary search tree invariant:

$bst :: 'a\ tree \Rightarrow bool$

The (binary) tree library

Binary search tree invariant:
$bst :: 'a\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x \in set\_tree\ l.\ x < a) \wedge (\forall x \in set\_tree\ r.\ a < x))$

The (binary) tree library

Binary search tree invariant:
$bst :: 'a\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x \in set\_tree\ l.\ x < a) \wedge (\forall x \in set\_tree\ r.\ a < x))$

For any type $'a$?

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Example: class *linorder*: linear orders with $\leq$, $<$

The (binary) tree library

Binary search tree invariant:
$bst :: {'a}\ tree \Rightarrow bool$

$bst\ \langle\rangle\ =\ True$
$bst\ \langle l,\ a,\ r\rangle\ =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x{\in}set\_tree\ l.\ x < a) \wedge (\forall x{\in}set\_tree\ r.\ a < x))$

For any type $'a$?

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Example: class *linorder*: linear orders with $\leq$, $<$

A type belongs to some class if

- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Example: class *linorder*: linear orders with $\leq$, $<$

A type belongs to some class if

- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

Notation: $\tau :: C$ means type τ belongs to class C

Isabelle's type classes

A *type class* is defined by

- a set of required functions (the interface)
- and a set of axioms about those functions

Example: class *linorder*: linear orders with $\leq$, $<$

A type belongs to some class if

- the interface functions are defined on that type
- and satisfy the axioms of the class (proof needed!)

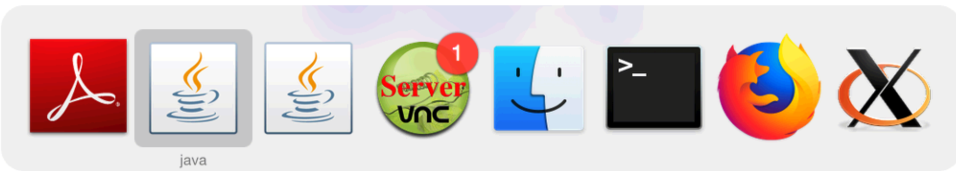
Notation: $\tau :: C$ means type τ belongs to class C

Example: $bst :: ('a :: linorder)\ tree \Rightarrow bool$

Case study

`BST_Demo.thy`

Case study



Chapter 4

Logic and Proof Beyond Equality