Simplification means ...

Using equations $l = r$ from left to right

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

An example

Equations:

\[
\begin{align*}
0 + n &= n & \text{(1)} \\
(Suc m) + n &= Suc (m + n) & \text{(2)} \\
(Suc m \leq Suc n) &= (m \leq n) & \text{(3)} \\
(0 \leq m) &= True & \text{(4)}
\end{align*}
\]

Rewriting:

\[
0 + Suc 0 \leq Suc 0 + x
\]
An example

Equations:

1. \[0 + n = n\]
2. \[(\text{Succ } m) + n = \text{Succ } (m + n)\]
3. \[(\text{Succ } m \leq \text{Succ } n) = (m \leq n)\]
4. \[(0 \leq m) = \text{True}\]

Rewriting:

An example

Equations:

1. \[0 + n = n\]
2. \[(\text{Succ } m) + n = \text{Succ } (m + n)\]
3. \[(\text{Succ } m \leq \text{Succ } n) = (m \leq n)\]
4. \[(0 \leq m) = \text{True}\]

Rewriting:
Conditional rewriting

Simplification rules can be conditional:

\[ [ P_1; \ldots; P_k ] \Rightarrow l = r \]

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

Example

\[ p(0) = True \]
\[ p(x) \Rightarrow f(x) = g(x) \]
Conditional rewriting

Simplification rules can be conditional:

\[
[ \ P_1; \ldots; \ P_k \ ] \implies l = r
\]

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

Example

\[
p(0) = True
\]
\[
p(x) \implies f(x) = g(x)
\]

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 \textit{simp}-rules (almost) blindly from left to right.

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

Termination

Simplification may not terminate.
Isabelle uses \textit{simp}-rules (almost) blindly from left to right.

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

Principle:

\[
[ \ P_1; \ldots; \ P_k \ ] \implies l = r
\]

is suitable as a \textit{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: \( f(x) = g(x), \quad g(x) = f(x) \)

Principle:

\[
\left[ P_1; \ldots; P_k \right] \implies l = r
\]

is suitable as a simp-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[
n < m \implies (n < \text{Suc } m) = \text{True}
\]

\[
\text{Suc } n < m \implies (n < m) = \text{True}
\]

Proof method \textit{simp}

Goal: 1. \[ P_1; \ldots; P_m \] \implies C

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

Proof method \textit{simp}

Goal: 1. \[ P_1; \ldots; P_m \] \implies C

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

Simplify \( P_1 \ldots P_m \) and \( C \) using

- lemmas with attribute \textit{simp}
Proof method \textit{simp}

Goal: \hspace{0.2cm} 1. \hspace{0.2cm} [ \hspace{0.1cm} P_1 ; \ldots ; P_m \hspace{0.1cm} ] \Rightarrow C

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

Simplify $P_1 \ldots P_m$ and $C$ using
- lemmas with attribute $simp$
- rules from \texttt{fun} and \texttt{datatype}

Variations:
- $(simp \ldots del: \ldots)$ removes $simp$-lemmas
- \texttt{add} and \texttt{del} are optional
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 can also be modified:
  (auto simp add: ... simp del: ...)
Rewriting with definitions

Definitions (definition) must be used explicitly:

\[(\text{simp add: f_def } \ldots)\]

Case splitting with \textit{simp/auto}

Automatic:

\[P \ (\text{if } A \ \text{then } s \ \text{else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))\]

By hand:

\[P \ (\text{case } e \ \text{of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. \ e = \text{Suc } n \rightarrow P(b))\]

Proof method: \textit{(simp split: nat.split)}
Case splitting with \textit{simp/auto}

Automatic:

\[
P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))
\]

By hand:

\[
P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. e = \text{Suc } n \rightarrow P(b))
\]

Proof method: \((\text{simp split: nat.split})\)
Or \textit{auto}. Similar for any datatype \(t\): \textit{t.split}

Splitting pairs with \textit{simp/auto}

How to replace

\[
P(\text{let } (x, y) = t \text{ in } u \ x \ y)
\]

or

\[
P(\text{case } t \text{ of } (x, y) \Rightarrow u \ x \ y)
\]

Splitting pairs with \textit{simp/auto}

How to replace

\[
P(\text{let } (x, y) = t \text{ in } u \ x \ y)
\]

or

\[
P(\text{case } t \text{ of } (x, y) \Rightarrow u \ x \ y)
\]

by

\[
\forall x \ y. t = (x, y) \rightarrow P(u \ x \ y)
\]
Splitting pairs with simp/auto

How to replace

\[ P \left( \text{let} \ (x, \ y) = t \ \text{in} \ u \ x \ y \right) \]
\[
\text{or}
\]

\[ P \left( \text{case} \ t \ \text{of} \ (x, \ y) \ \Rightarrow \ u \ x \ y \right) \]
\[
\text{by}
\]

\[ \forall \ x, y. \ t = (x, \ y) \rightarrow P \ (u \ x \ y) \]

Proof method: \((\text{simp split}: \text{prod.split})\)
Type: 'a set
Operations: $a \in A$, $A \cup B$, ...

Bounded quantification: $\forall a \in A. \; P$

Proof method \textit{auto} knows (a little) about sets.

The (binary) tree library

\texttt{imports "HOL-Library.Tree"}
The (binary) tree library

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

datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)

Abbreviations:

\[
\emptyset \equiv \text{Leaf}
\]

\[
\langle l, a, r \rangle \equiv \text{Node} \ l \ a \ r
\]
The (binary) tree library

Size = number of nodes:
\[ \text{size} :: \text{a tree} \Rightarrow \text{nat} \]
\[ \text{size} \emptyset = 0 \]
\[ \text{size} (l, _, r) = \text{size} l + \text{size} r + 1 \]

Height:
\[ \text{height} :: \text{a tree} \Rightarrow \text{nat} \]
\[ \text{height} \emptyset = 0 \]
\[ \text{height} (l, _, r) = \max (\text{height} l) (\text{height} r) + 1 \]
The (binary) tree library

The set of elements in a tree:

\[
\text{set\_tree} :: \text{a tree} \Rightarrow \text{'a set}
\]

\[
\text{set\_tree}() = \{}
\]

\[
\text{set\_tree}(l, a, r) = \text{set\_tree} l \cup \{a\} \cup \text{set\_tree} r
\]

Inorder listing:

\[
\text{inorder} :: \text{a tree} \Rightarrow \text{'a list}
\]

\[
\text{inorder}() = []
\]

\[
\text{inorder}(l, x, r) = \text{inorder} l @ [x] @ \text{inorder} r
\]
The (binary) tree library

Binary search tree invariant:
\( \text{bst} :: \forall 'a \Rightarrow \text{bool} \)

\[
\begin{align*}
\text{bst} \emptyset &= \text{True} \\
\text{bst} \langle l, a, r \rangle &= \\
&(l \wedge \text{bst} l) \\
&\quad \wedge (r \wedge \text{bst} r) \\
&\quad \wedge \left( \forall x \in \text{set}_{\text{tree}} l . x < a \right) \\
&\quad \wedge \left( \forall x \in \text{set}_{\text{tree}} r . a < x \right)
\end{align*}
\]

For any type 'a?

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 ≤, <

---

**The (binary) tree library**

Binary search tree invariant:

\[ bst :: 'a tree \Rightarrow \text{bool} \]

\[
bst \emptyset = \text{True} \\
bst (l, a, r) = \\
(bst l \land \\
bst r \land \\
(\forall x : \text{set tree} l. x < a) \land (\forall x : \text{set tree} r. a < x))
\]

For any type 'a?

---

**Isabelle’s type classes**

A *type class* is defined by

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

**Example:** class *linorder*: linear orders with ≤, <

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 $\tau$ belongs to class $C$

Example: $bst :: (\forall a :: \text{linorder}. \, \text{tree} \Rightarrow \text{bool}$

---

Case study

**BST_Demo.thy**
Chapter 4

Logic and Proof
Beyond Equality