Proof by Simplification

Overview

- Term rewriting foundations
- Term rewriting in Isabelle/HOL
  - Basic simplification
  - Extensions

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *rewrite rule*

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:

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

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

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:

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

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:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

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

More formally

substitution = mapping from variables to terms

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$ if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

Result: $a + (b + c)$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is *applicable* to term $t[s]$ with σ if

- $\sigma(l) = s$ and
- $\sigma(P_1), \dots, \sigma(P_n)$ are provable (again by rewriting).

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ ("unknown")

Schematic variables:

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ ("unknown")

Schematic variables:

- Logically: free = schematic

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: xs @ [] = xs*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Example: rewriting

rev(a @ []) = rev a

using *app_Nil2* with $\sigma = \{?xs \mapsto a\}$

Term rewriting in Isabelle

Basic simplification

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

apply(*simp add: eq*$_1$ $\ldots$ *eq*$_n$)

Basic simplification

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*

Basic simplification

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 **primrec**, **fun** and **datatype**

Basic simplification

Goal: *1.* $\llbracket$ *P*$_1$; ... ; *P*$_m$ $\rrbracket \Longrightarrow$ *C*

apply(*simp add: eq*$_1$ ... *eq*$_n$)

Simplify *P*$_1$ ... *P*$_m$ and *C* using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas *eq*$_1$ ... *eq*$_n$

Basic simplification

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 **primrec**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

Basic simplification

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 **primrec**, **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* applies *simp* and more

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

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

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

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

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

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

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

Rewriting with definitions

Definitions do not have the *simp* attribute.

Rewriting with definitions

Definitions do not have the *simp* attribute.

They must be used explicitly: *(simp add: f_def ...)*

Extensions of rewriting

Local assumptions

Simplification of $A \longrightarrow B$:

1. Simplify A to A'
2. Simplify B using A'

Case splitting with simp

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

Case splitting with simp

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

Automatic

Case splitting with simp

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

Automatic

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

Case splitting with simp

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

Automatic

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

By hand: *(simp split: nat.split)*

Case splitting with simp

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

Automatic

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

By hand: *(simp split: nat.split)*

Similar for any datatype *t*: *t.split*

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \leadsto a + b$ but not $a + b \leadsto b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Example: *(simp add: add_ac)* yields

$$(b + c) + a \leadsto \cdots \leadsto a + (b + c)$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$
\begin{array}{rcl}
\neg A & \mapsto & A = False \\
A \longrightarrow B & \mapsto & A \Longrightarrow B \\
A \wedge B & \mapsto & A, B \\
\forall x. A(x) & \mapsto & A(?x) \\
A & \mapsto & A = True
\end{array}
$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x. A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Example:

$(p \longrightarrow q \wedge \neg r) \wedge s \quad \mapsto$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{array}{rcl} \neg A & \mapsto & A = False \\ A \longrightarrow B & \mapsto & A \Longrightarrow B \\ A \wedge B & \mapsto & A,\ B \\ \forall x. A(x) & \mapsto & A(?x) \\ A & \mapsto & A = True \end{array}$$

Example:

$$(p \longrightarrow q \wedge \neg\, r) \wedge s \quad \mapsto \quad \left\{ \begin{array}{r} p \Longrightarrow q = True \\ p \Longrightarrow r = False \\ s = True \end{array} \right\}$$

When everything else fails: Tracing

Set trace mode on/off in Proof General:

Isabelle $\rightarrow$ Settings $\rightarrow$ Trace simplifier

Output in separate `trace` buffer

Demo: simp