
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 \rightsquigarrow *rewrite rule*

An example

Equations:

$$0 + n = n \quad (1)$$
$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$
$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$
$$(0 \leq m) = True \quad (4)$$

An example

Equations:

$$0 + n = n \quad (1)$$
$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$
$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$
$$(0 \leq m) = True \quad (4)$$

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

Rewriting:

An example

Equations:

$$0 + n = n \quad (1)$$

$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$

$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$

$$(0 \leq m) = True \quad (4)$$

$$0 + Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(1)}}$$

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

Rewriting:

An example

Equations:

$$0 + n = n \quad (1)$$

$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$

$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$

$$(0 \leq m) = True \quad (4)$$

Rewriting:

$$0 + Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(1)}}$$

$$Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(2)}}$$

$$Suc\ 0 \leq Suc\ (0 + x)$$

An example

Equations:

$$0 + n = n \quad (1)$$

$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$

$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$

$$(0 \leq m) = True \quad (4)$$

Rewriting:

$$0 + Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(1)}}$$

$$Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(2)}}$$

$$Suc\ 0 \leq Suc\ (0 + x) \quad \underline{\underline{(3)}}$$

$$0 \leq 0 + x$$

An example

Equations:

$$0 + n = n \quad (1)$$

$$(Suc\ m) + n = Suc\ (m + n) \quad (2)$$

$$(Suc\ m \leq Suc\ n) = (m \leq n) \quad (3)$$

$$(0 \leq m) = True \quad (4)$$

Rewriting:

$$0 + Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(1)}}$$

$$Suc\ 0 \leq Suc\ 0 + x \quad \underline{\underline{(2)}}$$

$$Suc\ 0 \leq Suc\ (0 + x) \quad \underline{\underline{(3)}}$$

$$0 \leq 0 + x \quad \underline{\underline{(4)}}$$

True

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:

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

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”)

Can be mixed: $\forall b. f ?a y = b$

Schematic variables

Three kinds of variables:

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

Can be mixed: $\forall b. f ?a y = b$

- Logically: free = schematic

Schematic variables

Three kinds of variables:

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

Can be mixed: $\forall b. f ?a y = b$

- 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

$$\text{rev}(a @ []) = \text{rev } a$$

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

Term rewriting in Isabelle

Basic simplification

Goal: 1. $\llbracket P_1; \dots ; P_m \rrbracket \implies C$

apply(simp add: eq₁ ... eq_n)

Basic simplification

Goal: 1. $\llbracket P_1; \dots ; P_m \rrbracket \implies C$

apply(simp add: eq₁ ... eq_n)

Simplify $P_1 \dots P_m$ and C using

- lemmas with attribute *simp*

Basic simplification

Goal: 1. $\llbracket P_1; \dots ; P_m \rrbracket \implies C$

apply(simp add: eq₁ ... eq_n)

Simplify $P_1 \dots P_m$ and C using

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

Basic simplification

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

apply(simp add: eq₁ ... eq_n)

Simplify $P_1 \dots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec** and **datatype**
- additional lemmas $eq_1 \dots eq_n$

Basic simplification

Goal: 1. $\llbracket P_1; \dots ; P_m \rrbracket \implies C$

apply(simp add: eq₁ ... eq_n)

Simplify $P_1 \dots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec** and **datatype**
- additional lemmas $eq_1 \dots eq_n$
- assumptions $P_1 \dots P_m$

Basic simplification

Goal: 1. $\llbracket P_1; \dots ; P_m \rrbracket \implies C$

apply(simp add: eq₁ ... eq_n)

Simplify $P_1 \dots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec** and **datatype**
- additional lemmas $eq_1 \dots eq_n$
- assumptions $P_1 \dots P_m$

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

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

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

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

is suitable as a *simp*-rule only

if l is “bigger” than r and each P_i

$$n < m \Longrightarrow (n < \text{Suc } m) = \text{True}$$

$$\text{Suc } n < m \Longrightarrow (n < m) = \text{True}$$

Termination

Simplification may not terminate.

Isabelle uses *simp*-rules (almost) blindly from left to right.

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

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

is suitable as a *simp*-rule only

if l is “bigger” than r and each P_i

$$n < m \Longrightarrow (n < \text{Suc } m) = \text{True} \quad \text{YES}$$

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

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

```
apply(simp (no_asm_simp) ...)
```

Simplify only conclusion

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

`apply(simp (no_asm_simp) ...)`

Simplify only conclusion

`apply(simp (no_asm_use) ...)`

Simplify but do not use assumptions

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

`apply(simp (no_asm_simp) ...)`

Simplify only conclusion

`apply(simp (no_asm_use) ...)`

Simplify but do not use assumptions

`apply(simp (no_asm) ...)`

Ignore assumptions completely

Rewriting with definitions (constdefs)

Definitions do not have the *simp* attribute.

Rewriting with definitions (constdefs)

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

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

Case splitting with simp

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

Automatic

Case splitting with simp

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

Automatic

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

Case splitting with simp

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

Automatic

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

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

Case splitting with simp

$$\begin{aligned} & P(\text{if } A \text{ then } s \text{ else } t) \\ & \quad = \\ & (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t)) \end{aligned}$$

Automatic

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

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: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

Ordered rewriting

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

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

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

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: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types *nat*, *int* etc:

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

Example: (*simp add: add_ac*) yields

$$(b + c) + a \rightsquigarrow \dots \rightsquigarrow a + (b + c)$$

Preprocessing

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

$$\neg A \mapsto A = \textit{False}$$

$$A \longrightarrow B \mapsto A \implies B$$

$$A \wedge B \mapsto A, B$$

$$\forall x.A(x) \mapsto A(?x)$$

$$A \mapsto A = \textit{True}$$

Preprocessing

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

$$\neg A \mapsto A = \textit{False}$$

$$A \longrightarrow B \mapsto A \implies B$$

$$A \wedge B \mapsto A, B$$

$$\forall x.A(x) \mapsto A(?x)$$

$$A \mapsto A = \textit{True}$$

Example:

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

Preprocessing

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

$$\neg A \mapsto A = \textit{False}$$

$$A \longrightarrow B \mapsto A \implies B$$

$$A \wedge B \mapsto A, B$$

$$\forall x.A(x) \mapsto A(?x)$$

$$A \mapsto A = \textit{True}$$

Example:

$$(p \longrightarrow q \wedge \neg r) \wedge s \mapsto p \implies q = \textit{True}, r = \textit{False}, s = \textit{True}$$

When everything else fails: Tracing

Set trace mode on/off in Proof General:

Isabelle/Isar → Settings → Trace simplifier

Output in separate buffer:

Proof-General → Buffers → Trace

Demo: simp

Induction heuristics

Basic heuristics

Theorems about recursive functions are proved by
induction

Basic 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

`consts itrev :: 'a list \Rightarrow 'a list \Rightarrow 'a list`

A tail recursive reverse

consts *itrev* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

primrec

itrev [] *ys* = *ys*

itrev (x#xs) *ys* =

A tail recursive reverse

consts *itrev* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

primrec

itrev [] *ys* = *ys*

itrev (x#xs) *ys* = *itrev* xs (x#*ys*)

A tail recursive reverse

consts *itrev* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

primrec

itrev [] $ys = ys$

itrev (x#xs) $ys = itrev\ xs\ (x\#\ys)$

lemma *itrev* xs [] = *rev* xs

A tail recursive reverse

consts *itrev* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

primrec

itrev [] $ys = ys$

itrev (x#xs) $ys = itrev xs (x#ys)$

lemma *itrev xs [] = rev xs*

Why in this direction?

A tail recursive reverse

consts *itrev* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

primrec

itrev [] $ys = ys$

itrev (x#xs) $ys = itrev xs (x#ys)$

lemma *itrev xs [] = rev xs*

Why in this direction?

Because the lhs is “more complex” than the rhs.

Demo: first proof attempt

Generalisation (1)

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Demo: second proof attempt

Generalisation (2)

Quantify free variables by \forall
(except the induction variable)

lemma $\forall ys. \text{itrev } xs \ ys = \text{rev } xs \ @ \ ys$