COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods

$\longrightarrow$

Content

- ➜ Intro & motivation, getting started with Isabelle
- ➜ **Foundations & Principles**
  - Lambda Calculus
  - Higher Order Logic, natural deduction
  - **Term rewriting**
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

Last Time

- ➜ Introducing new Types
- ➜ Equations and Term Rewriting
- ➜ Confluence and Termination of reduction systems
- ➜ Term Rewriting in Isabelle

EXERCISES

- use **typedef** to define a new type v with exactly one element.
- define a constant u of type v
- show that every element of v is equal to u
- design a set of rules that turns formulae with $\wedge, \vee, \longrightarrow, \neg$
  into disjunctive normal form
  (= disjunction of conjunctions with negation only directly on variables)
- prove those rules in Isabelle
- use **simp only** with these rules on $(\neg B \longrightarrow C) \longrightarrow A \longrightarrow B$

ISAR

A LANGUAGE FOR STRUCTURED PROOFS

NIC

apply scripts

- ➜ unreadable
- ➜ hard to maintain
- ➜ do not scale

No structure.

What about..

- ➜ Elegance?
- ➜ Explaining deeper insights?
- ➜ Large developments?

Isar!

A typical Isar proof

proof

  assume $formula_0$

  have $formula_1$ **by** simp

  $\vdots$

  have $formula_n$ **by** blast

  show $formula_{n+1}$ **by** $\ldots$

qed

proves $formula_0 \Longrightarrow formula_{n+1}$

(analogous to **assumes/shows** in lemma statements) proof = **proof** [method] statement* **qed**

| **by** method

method = (simp . . .) | (blast . . .) | (rule . . .) | . . .

statement = **fix** variables ($\bigwedge$)

| **assume** proposition ($\Longrightarrow$)

| [**from** name$^+$] (**have** | **show**) proposition proof

| **next** (separates subgoals)

proposition = [name:] formula

proof [method] statement* **qed**

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)
  assume A: "A"
  from A **show** "A" **by** assumption
next
  assume B: "B"
  from B **show** "B" **by** assumption
qed

- ➜ **proof** (<method>) applies method to the stated goal
- ➜ **proof** applies a single rule that fits
- ➜ **proof -** does nothing to the goal

How do I know what to Assume and Show?

Look at the proof state!

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)

- ➜ **proof** (rule conjI) changes proof state to
  1. $[\![A; B]\!] \Longrightarrow A$
  2. $[\![A; B]\!] \Longrightarrow B$
- ➜ so we need 2 shows: **show** "A" and **show** "B"
- ➜ We are allowed to **assume** A,
  because A is in the assumptions of the proof state.

The Three Modes of Isar

- **[prove]**:
  goal has been stated, proof needs to follow.
- **[state]**:
  proof block has openend or subgoal has been proved,
  new *from* statement, goal statement or assumptions can follow.
- **[chain]**:
  from statement has been made, goal statement needs to follow.

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[state]**
  assume A: "A" **[state]**
  from A **[chain]** **show** "A" **[prove]** **by** assumption **[state]**
next **[state]** **. . .**

HAVE

Can be used to make intermediate steps.

Example:

lemma "$(x :: \mathsf{nat}) + 1 = 1 + x$"

proof -

  have A: "$x + 1 = \mathsf{Suc}\ x$" **by** simp

  have B: "$1 + x = \mathsf{Suc}\ x$" **by** simp

  show "$x + 1 = 1 + x$" **by** (simp only: A B)

qed

DEMO: ISAR PROOFS

Back to Term Rewriting …

APPLYING A REWRITE RULE

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

Example:

Rule: $0 + n \longrightarrow n$

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

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

Result: $a + (b + c)$

Rewrite rules can be conditional:

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

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

- ➜ $\sigma\ l = s$ and
- ➜ $\sigma\ P_1, \ldots, \sigma\ P_n$ are provable by rewriting.

Rewriting with Assumptions

Last time: Isabelle uses assumptions in rewriting.

Can lead to non-termination.

Example:

lemma "$f\ x = g\ x \land g\ x = f\ x \Longrightarrow f\ x = 2$"

simp	**use and simplify** assumptions
(simp (no_asm))	**ignore** assumptions
(simp (no_asm_use))	**simplify**, but do **not use** assumptions
(simp (no_asm_simp))	**use**, but do **not simplify** assumptions

Preprocessing (recursive) 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:

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

$$\mapsto$$

$$p \Longrightarrow q = True \qquad r = False \qquad s = True$$

Demo

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

Automatic

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

Manually: apply (simp split: nat.split)

Similar for any data type t: **t.split**

Congruence Rules

congruence rules are about using context

Example: in $P \longrightarrow Q$ we could use P to simplify terms in Q

For $\Longrightarrow$ hardwired (assumptions used in rewriting)

For other operators expressed with conditional rewriting.

Example: $[\![P = P'; P' \Longrightarrow Q = Q']\!] \Longrightarrow (P \longrightarrow Q) = (P' \longrightarrow Q')$

Read: to simplify $P \longrightarrow Q$

- ➜ first simplify P to P'
- ➜ then simplify Q to Q' using P' as assumption
- ➜ the result is $P' \longrightarrow Q'$

More Congruence

Sometimes useful, but not used automatically (slowdown):

conj_cong: $\llbracket P = P'; P' \Longrightarrow Q = Q' \rrbracket \Longrightarrow (P \wedge Q) = (P' \wedge Q')$

Context for if-then-else:

if_cong: $\llbracket b = c; c \Longrightarrow x = u; \neg c \Longrightarrow y = v \rrbracket \Longrightarrow$
$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$

Prevent rewriting inside then-else (default):

if_weak_cong: $b = c \Longrightarrow (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$

- ➜ declare own congruence rules with **[cong]** attribute
- ➜ delete with **[cong del]**

ORDERED REWRITING

Problem: $x + y \longrightarrow y + x$ does not terminate

Solution: use permutative rules only if 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: **apply** (simp add: add_ac) yields

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

AC Rules

Example for associative-commutative rules:

Associative: $(x \odot y) \odot z = x \odot (y \odot z)$

Commutative: $x \odot y = y \odot x$

These 2 rules alone get stuck too early (not confluent).

Example: $(z \odot x) \odot (y \odot v)$

We want: $(z \odot x) \odot (y \odot v) = v \odot (x \odot (y \odot z))$

We get: $(z \odot x) \odot (y \odot v) = v \odot (y \odot (x \odot z))$

We need: **AC rule** $x \odot (y \odot z) = y \odot (x \odot z)$

If these 3 rules are present for an AC operator
Isabelle will order terms correctly

DEMO

Back to Confluence

Last time: confluence in general is undecidable.

But: confluence for terminating systems is decidable!

Problem: overlapping lhs of rules.

Definition:

Let $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ be two rules with disjoint variables.

They form a **critical pair** if a non-variable subterm of l_1 unifies with l_2.

Example:

Rules: (1) $f\ x \longrightarrow a$ (2) $g\ y \longrightarrow b$ (3) $f\ (g\ z) \longrightarrow b$

Critical pairs:

(1)+(3)	$\{x \mapsto g\ z\}$	$a \xleftarrow{(1)} f\ g\ t \xrightarrow{(3)} b$
(3)+(2)	$\{z \mapsto y\}$	$b \xleftarrow{(3)} f\ g\ t \xrightarrow{(2)} b$

COMPLETION

(1) $f\ x \longrightarrow a$ (2) $g\ y \longrightarrow b$ (3) $f\ (g\ z) \longrightarrow b$

is not confluent

But it can be made confluent by adding rules!

How: join all critical pairs

Example:

(1)+(3) $\{x \mapsto g\ z\}$ $a \xleftarrow{(1)} f\ g\ t \xrightarrow{(3)} b$

shows that $a = b$ (because $a \overset{*}{\longleftrightarrow} b$), so we add $a \longrightarrow b$ as a rule

This is the main idea of the Knuth-Bendix completion algorithm.

DEMO: WALDMEISTER

We have learned today ...

- → Isar
- → Conditional term rewriting
- → Congruence rules
- → AC rules
- → More on confluence