COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein

Formal Methods

$$\longrightarrow \qquad \{\}$$

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

- ➜ Isar, structured proofs
- ➜ Term rewriting, rule applications
- ➜ Conditional term rewriting
- ➜ Congruence rules

NICTA

ADVANCED TERM 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

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

(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 $a \xleftrightarrow{*} b$), so we add $a \longrightarrow b$ as a rule

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

NICTA

Demo: Waldmeister

Definitions:

A **rule** $l \longrightarrow r$ is **left-linear** if no variable occurs twice in l.

A **rewrite system** is **left-linear** if all rules are.

A system is **orthogonal** if it is left-linear and has no critical pairs.

Orthogonal rewrite systems are confluent

Application: functional programming languages

That was Term Rewriting

More Isar

Last Time on Isar

➜ basic syntax

➜ proof and qed

➜ assume and show

➜ from and have

➜ the three modes of Isar

Backward and Forward

Backward reasoning: ... **have** "$A \wedge B$" **proof**

- ➜ **proof** picks an **intro** rule automatically
- ➜ conclusion of rule must unify with $A \wedge B$

Forward reasoning: ...

assume AB: "$A \wedge B$"

from AB **have** "..." **proof**

- ➜ now **proof** picks an **elim** rule automatically
- ➜ triggered by **from**
- ➜ first assumption of rule must unify with AB

General case: **from** $A_1 \ldots A_n$ **have** R **proof**

- ➜ first n assumptions of rule must unify with $A_1 \ldots A_n$
- ➜ conclusion of rule must unify with R

Fix and Obtain

fix $v_1 \dots v_n$

Introduces new arbitrary but fixed variables

($\sim$ parameters, $\bigwedge$)

obtain $v_1 \dots v_n$ **where** <prop> <proof>

Introduces new variables together with property

Demo

Fancy Abbreviations

this	=	the previous fact proved or assumed
then	=	**from** this
thus	=	**then show**
hence	=	**then have**
with $A_1 \ldots A_n$	=	**from** $A_1 \ldots A_n$ this
?thesis	=	the last enclosing goal statement

Moreover and Ultimately

have X_1: P_1 ...	**have** P_1 ...
have X_2: P_2 ...	**moreover have** P_2 ...
$\vdots$	$\vdots$
have X_n: P_n ...	**moreover have** P_n ...
from $X_1 \dots X_n$ **show** ...	**ultimately show** ...

wastes lots of brain power

on names $X_1 \dots X_n$

General Case Distinctions

show $formula$

proof -

have $P_1 \vee P_2 \vee P_3$ <proof>

moreover { **assume** P_1 ... **have** ?thesis <proof> }

moreover { **assume** P_2 ... **have** ?thesis <proof> }

moreover { **assume** P_3 ... **have** ?thesis <proof> }

ultimately show ?thesis **by** blast

qed

{ ... } is a proof block similar to **proof** ... **qed**

{ **assume** P_1 ... **have** P <proof> }
stands for $P_1 \Longrightarrow P$

MIXING PROOF STYLES

from . . .

have . . .

  apply -   make incoming facts assumptions

  apply (. . .)

  $\vdots$

  apply (. . .)

  done