COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods
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
→ 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 `∧`, `∨`, `→`, `¬` 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 `(¬B → C) → A → B`
Isar

A Language for Structured Proofs
apply scripts

- unreadable
- hard to maintain
- do not scale

What about..

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

No structure. Isar!
A typical ISAR proof

proof

assume \( \text{formula}_0 \)

have \( \text{formula}_1 \) by simp

\vdots

have \( \text{formula}_n \) by blast

show \( \text{formula}_{n+1} \) by \ldots

qed

proves \( \text{formula}_0 \implies \text{formula}_{n+1} \)

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

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

statement = fix variables (∧)
    | assume proposition (⇒)
    | [from name+] (have | show) proposition proof
    | next (separates subgoals)

proposition = [name:] formula
proof [method] statement* qed

lemma 

\[ [A; B] \implies A \land B \]

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** 

\[ [A; B] \implies A \land B \]

**proof** (rule conjI)

\[ \begin{align*}
\text{proof } \text{(rule conjI)} \text{ changes proof state to} \\
1. [A; B] \implies A \\
2. [A; B] \implies B \\
\end{align*} \]

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 opened 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] → A ∧ B*" **[prove]**
**proof** (rule conjI) **[state]**
  assume A: "*A*" **[state]**
  from A **[chain]** show "*A*" **[prove]** by assumption **[state]**
next **[state]** ...
Can be used to make intermediate steps.

Example:

lemma 

\[(x :: \text{nat}) + 1 = 1 + x\]

proof -

have A: 

\[x + 1 = \text{Suc } x\]

by simp

have B: 

\[1 + x = \text{Suc } x\]

by simp

show 

\[x + 1 = 1 + x\]

by (simp only: A B)

qed
Demo: ISAR Proofs
BACK TO TERM Rewriting ...
Applying a Rewrite Rule

\[ l \rightarrow r \text{ applicable} \] to term \( t[s] \)
if there is substitution \( \sigma \) such that \( \sigma l = s \)

\[ \Rightarrow \text{ Result: } t[\sigma r] \]
\[ \Rightarrow \text{ Equationally: } t[s] = t[\sigma r] \]

Example:

Rule: \( 0 + n \rightarrow n \)

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

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

Result: \( a + (b + c) \)
Rewrite rules can be conditional:

\[
[P_1 \ldots P_n] \implies l = r
\]

is **applicable** to term \(t[s]\) with \(\sigma\) 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 ∧ g x = f x ⇒ 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:

\[ \neg A \iff A = False \]
\[ A \rightarrow B \iff A \rightarrow\rightarrow B \]
\[ A \land B \iff A, B \]
\[ \forall x. A \; x \iff A \; ?x \]
\[ A \iff A = True \]

Example:

\[ (p \rightarrow q \land \neg r) \land s \]
\[ \iff \]
\[ p \rightarrow q = True \quad r = False \quad s = True \]
DEMO
CASE SPLITTING WITH SIMP

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

Automatic

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

Manually: apply \ (\text{simp split: nat.split})

Similar for any data type \( t \): \texttt{t.split}
CONGRUENCE RULES

congruence rules are about using context

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

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

For other operators expressed with conditional rewriting.

Example: $[P = P'; P' \Rightarrow Q = Q'] \Rightarrow (P \rightarrow Q) = (P' \rightarrow Q')$

Read: to simplify $P \rightarrow Q$

$\Rightarrow$ first simplify $P$ to $P'$
$\Rightarrow$ then simplify $Q$ to $Q'$ using $P'$ as assumption
$\Rightarrow$ the result is $P' \rightarrow Q'$
Sometimes useful, but not used automatically (slowdown):

**conj_cong:** \[ P = P'; P' \implies Q = Q' \] \implies (P \land Q) = (P' \land Q')

Context for if-then-else:

**if_cong:** \[ b = c; c \implies x = u; \neg c \implies y = v \] \implies

(if \( b \) then \( x \) else \( y \)) = (if \( c \) then \( u \) else \( v \))

Prevent rewriting inside then-else (default):

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

→ declare own congruence rules with \[\text{cong}\] attribute
→ delete with \[\text{cong del}\]
Problem: \( x + y \longrightarrow y + x \) does not terminate

Solution: use permutative rules only if term becomes lexicographically smaller.

Example: \( b + a \sim a + b \) but not \( a + b \sim b + a \).

For types nat, int etc:

- lemmas \texttt{add_ac} sort any sum (+)
- lemmas \texttt{times_ac} sort any product (*)

Example: apply (simp add: add_ac) yields
\[
(b + c) + a \sim \cdots \sim a + (b + c)
\]
Example for associative-commutative rules:

**Associative:** \((x \circ y) \circ z = x \circ (y \circ z)\)

**Commutative:** \(x \circ y = y \circ x\)

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

**Example:** \((z \circ x) \circ (y \circ v)\)

**We want:** \((z \circ x) \circ (y \circ v) = v \circ (x \circ (y \circ z))\)

**We get:** \((z \circ x) \circ (y \circ v) = v \circ (y \circ (x \circ z))\)

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

If these 3 rules are present for an AC operator, Isabelle will order terms correctly.
**Last time:** confluence in general is undecidable.
**But:** confluence for terminating systems is decidable!
**Problem:** overlapping lhs of rules.

**Definition:**
Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ 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) $f \ x \rightarrow a$  (2) $g \ y \rightarrow b$  (3) $f \ (g \ z) \rightarrow b$
Critical pairs:

(1)+(3) \[ \{ x \mapsto g \ z \} \quad a \overset{(1)}{\leftarrow} f \ g \ t \overset{(3)}{\rightarrow} b \]
(3)+(2) \[ \{ z \mapsto y \} \quad b \overset{(3)}{\leftarrow} f \ g \ t \overset{(2)}{\rightarrow} b \]
(1) \( f \ x \rightarrow a \)  \( (2) \ g \ y \rightarrow b \)  \( (3) \ f \ (g \ z) \rightarrow b \)

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 \( a \xrightarrow{*} b \)), so we add \( a \rightarrow b \) 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