NICTA

COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

$\longrightarrow$

Slide 1

NICTA

Content

- Intro & motivation, getting started [1]
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic [3]
  - Term rewriting [4[a]]
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6[b], 7]
  - Code generation, type classes [7]
  - Hoare logic, proofs about programs, refinement [8,9[c],10[d]]
  - Isar, locales [11,12]

[a] a1 due; [b] a2 due; [c] session break; [d] a3 due

Slide 2

NICTA

Last Time on HOL

- Defining HOL
- Higher Order Abstract Syntax
- Deriving proof rules
- More automation

Slide 3

NICTA

The Three Basic Ways of Introducing Theorems

- **Axioms**:

  Expample: **axioms** refl: "$t = t$"

  Do not use. Evil. Can make your logic inconsistent.

- **Definitions:**

  Example: **definition** inj **where** "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"
  Introduces a new lemma called inj_def.

- **Proofs:**

  Example: **lemma** "inj $(\lambda x.\ x + 1)$"

  The harder, but safe choice.

Slide 4

The Three Basic Ways of Introducing Types

NICTA

➜ **typedecl**: by name only

Example: **typedecl** names
Introduces new type *names* without any further assumptions

➜ **type_synonym**: by abbreviation

Example: **type_synonym** α rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"
Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow bool$
Type abbreviations are immediately expanded internally

➜ **typedef**: by definiton as a set

Example: **typedef** new_type = "{some set}" <proof>
Introduces a new type as a subset of an existing type.
The proof shows that the set on the rhs in non-empty.

More on **typedef** in later lectures.

Slide 5

NICTA

TERM REWRITING

Slide 6

The Problem

NICTA

Given a set of equations

$$l_1 = r_1$$
$$l_2 = r_2$$
$$\vdots$$
$$l_n = r_n$$

does equation $l = r$ hold?

Applications in:

➜ **Mathematics** (algebra, group theory, etc)
➜ **Functional Programming** (model of execution)
➜ **Theorem Proving** (dealing with equations, simplifying statements)

Slide 7

Term Rewriting: The Idea

NICTA

use equations as reduction rules

$$l_1 \longrightarrow r_1$$
$$l_2 \longrightarrow r_2$$
$$\vdots$$
$$l_n \longrightarrow r_n$$

decide $l = r$ by deciding $l \stackrel{*}{\longleftrightarrow} r$

Slide 8

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y)\|x=y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x)\|x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse
$\longleftrightarrow$	$=$	$\longleftarrow \cup \longrightarrow$	symmetric closure
$\xleftrightarrow{+}$	$=$	$\bigcup_{i>0} \xleftrightarrow{i}$	transitive symmetric closure
$\xleftrightarrow{*}$	$=$	$\xleftrightarrow{+} \cup \xleftrightarrow{0}$	reflexive transitive symmetric closure

Slide 9

How to Decide $l \xleftrightarrow{*} r$

Same idea as for β: look for n such that $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$

Does this always work?

If $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$ then $l \xleftrightarrow{*} r$. Ok.

If $l \xleftrightarrow{*} r$, will there always be a suitable n? **No!**

Example:

Rules: $f\ x \longrightarrow a, \quad g\ x \longrightarrow b, \quad f\ (g\ x) \longrightarrow b$

$f\ x \xleftrightarrow{*} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

But: $f\ x \longrightarrow a$ and $g\ x \longrightarrow b$ and a, b in normal form

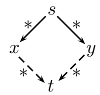
Works only for systems with **Church-Rosser** property:

$$l \xleftrightarrow{*} r \Longrightarrow \exists n.\ l \xrightarrow{*} n \wedge r \xrightarrow{*} n$$

Fact: $\longrightarrow$ is Church-Rosser iff it is confluent.

Slide 10

Confluence

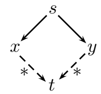


Problem:

is a given set of reduction rules confluent?

undecidable

Local Confluence



Fact: local confluence and termination $\Longrightarrow$ confluence

Slide 11

Termination

$\longrightarrow$ is **terminating** if there are no infinite reduction chains

$\longrightarrow$ is **normalizing** if each element has a normal form

$\longrightarrow$ is **convergent** if it is terminating and confluent

Example:

$\longrightarrow_\beta$ in λ is not terminating, but confluent

$\longrightarrow_\beta$ in $\lambda^{\rightarrow}$ is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?

undecidable

Slide 12

When is $\longrightarrow$ Terminating?

Basic idea: when each rule application makes terms simpler in some way.

More formally: $\longrightarrow$ is terminating when
there is a well founded order $<$ on terms for which $s < t$ whenever $t \longrightarrow s$
(well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

Example: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $size(s) < size(t)$ with
$size(s) =$ number of function symbols in s

① Both rules always decrease $size$ by 1 when applied to any term t
② $<_r$ is well founded, because $<$ is well founded on $\mathbb{N}$

Slide 13

Termination in Practice

In practice: often easier to consider just the rewrite rules by themselves,
rather than their application to an arbitrary term t.

Show for each rule $l_i = r_i$, that $r_i < l_i$.

Example:
$g\ x < f\ (g\ x)$ and $f\ x < g\ (f\ x)$

Requires t to become smaller whenever any subterm of t is made smaller.

Formally:
Requires $<$ to be **monotonic** with respect to the structure of terms:
$s < t \longrightarrow u[s] < u[t]$.
True for most orders that don't treat certain parts of terms as special cases.

Slide 14

Example Termination Proof

Problem: Rewrite formulae containing $\neg$, $\wedge$, $\vee$ and $\longrightarrow$, so that they don't contain any implications and $\neg$ is applied only to variables and constants.

Rewrite Rules:

➜ Remove implications:
imp: $(A \longrightarrow B) = (\neg A \vee B)$

➜ Push $\neg$s down past other operators:
notnot: $(\neg\neg P) = P$
notand: $(\neg(A \wedge B)) = (\neg A \vee \neg B)$
notor: $(\neg(A \vee B)) = (\neg A \wedge \neg B)$

We show that the rewrite system defined by these rules is terminating.

Slide 15

Order on Terms

Each time one of our rules is applied, either:

➜ an implication is removed, or
➜ something that is not a $\neg$ is hoisted upwards in the term.

This suggests a 2-part order, $<_r$: $s <_r t$ iff:

➜ num_imps $s <$ num_imps t, or
➜ num_imps $s =$ num_imps $t \wedge$ osize $s <$ osize t.

Let:

➜ $s <_i t \equiv$ num_imps $s <$ num_imps t and
➜ $s <_n t \equiv$ osize $s <$ osize t

Then $<_i$ and $<_n$ are both well-founded orders (since both functions return nats).

$<_r$ is the lexicographic order over $<_i$ and $<_n$. $<_r$ is well-founded since $<_i$ and $<_n$ are both well-founded.

Slide 16

Order Decreasing

imp clearly decreases num_imps.

osize adds up all non-¬ operators and variables/constants, weights each one according to its depth within the term.

$$\begin{array}{ll}
\text{osize}'\ c & \text{acm} = 2^{\text{acm}} \\
\text{osize}'\ (\neg P) & \text{acm} = \text{osize}'\ P\ (\text{acm}+1) \\
\text{osize}'\ (P \wedge Q) & \text{acm} = 2^{\text{acm}} + (\text{osize}'\ P\ (\text{acm}+1)) + (\text{osize}'\ Q\ (\text{acm}+1)) \\
\text{osize}'\ (P \vee Q) & \text{acm} = 2^{\text{acm}} + (\text{osize}'\ P\ (\text{acm}+1)) + (\text{osize}'\ Q\ (\text{acm}+1)) \\
\text{osize}'\ (P \longrightarrow Q) & \text{acm} = 2^{\text{acm}} + (\text{osize}'\ P\ (\text{acm}+1)) + (\text{osize}'\ Q\ (\text{acm}+1)) \\
\text{osize}\ \ P & \quad\ = \text{osize}'\ P\ 0
\end{array}$$

The other rules decrease the depth of the things osize counts, so decrease osize.

Slide 17

Term Rewriting in Isabelle

Term rewriting engine in Isabelle is called **Simplifier**

apply simp

- ➜ uses simplification rules
- ➜ (almost) blindly from left to right
- ➜ until no rule is applicable.

termination: not guaranteed
(may loop)

confluence: not guaranteed
(result may depend on which rule is used first)

Slide 18

Control

- ➜ Equations turned into simplification rules with **[simp]** attribute
- ➜ Adding/deleting equations locally:
  apply (simp add: <rules>) and **apply** (simp del: <rules>)
- ➜ Using only the specified set of equations:
  apply (simp only: <rules>)

Slide 19

DEMO

Slide 20

We have seen today...

➜ Equations and Term Rewriting
➜ Confluence and Termination of reduction systems
➜ Term Rewriting in Isabelle

Slide 21

Exercises

➜ Show, via a pen-and-paper proof, that the osize function is monotonic with respect to the structure of terms from that example.

Slide 22