NICTA

COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein

Formal Methods

a = b = c = . . .

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**
  - Hoare logic, proofs about programs
  - Locales, Presentation

Last time ...

- fun, function
- Well founded recursion

DEMO

MORE FUN

Calculational Reasoning

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

Can we do this in Isabelle?

➜ Simplifier: too eager

➜ Manual: difficult in apply style

➜ Isar: with the methods we know, too verbose

The Problem

$$\begin{array}{lcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

➜ Keywords **also** and **finally** to delimit steps

➜ **...**: predefined schematic term variable,
refers to right hand side of last expression

➜ Automatic use of transitivity rules to connect steps

ALSO/FINALLY

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"
have "$\cdots = t_n$" [proof]	
finally	$t_0 = t_n$
show P	

— 'finally' pipes fact "$t_0 = t_n$" into the proof

MORE ABOUT ALSO

- Works for all combinations of $=$, $\leq$ and $<$.
- Uses all rules declared as `[trans]`.
- To view all combinations in Proof General:

  Isabelle/Isar $\rightarrow$ Show me $\rightarrow$ Transitivity rules

calculation = "$l_1 \odot r_1$"
have "$\ldots \odot r_2$" [proof]
also $\Longleftarrow$

Anatomy of a [trans] rule:

- ➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- ➜ More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- ➜ pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- ➜ mixed: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$
- ➜ substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$
- ➜ antisymmetry: $[\![a < b; b < a]\!] \Longrightarrow P$
- ➜ monotonicity: $[\![a = f\ b; b < c; \bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y]\!] \Longrightarrow a < f\ c$

Demo

HOL AS PROGRAMMING LANGUAGE

We have

- ➜ numbers, arithmetic
- ➜ recursive datatypes
- ➜ constant definitions, recursive functions
- ➜ = a functional programming language
- ➜ can be used to get fully verified programs

Executed using the simplifier. But:

- ➜ slow, heavy-weight
- ➜ does not run stand-alone (without Isabelle)

GENERATING ML CODE

Generate stand-alone ML code for

- ➜ datatypes
- ➜ function definitions
- ➜ inductive definitions (sets)

Syntax (simplified):

code_module <structure-name> [**file** <name>]

contains

<ML-name> = <term>

⋮

<ML-name> = <term>

Generates ML stucture, puts it in own file or includes in current context

VALUE AND QUICKCHECK

Evaluate big terms quickly:

value ”<term>”

- ➜ generates ML code
- ➜ runs ML
- ➜ converts back into Isabelle term

Try some values on current proof state:

quickcheck

- ➜ generates ML code
- ➜ runs ML on random values for numbers and datatypes
- ➜ increasing size of data set until limit reached

Customisation

- lemma instead of definition: **[code]** attribute

  lemma [code]: "(0 $<$ Suc n) = True" by simp

- provide own code for types: **types_code**

  types_code "×" ("(_ */ _)")

- provide own code for consts: **consts_code**

  consts_code "Pair" ("(_,/ _)")

- complex code template: patterns + **attach**

  consts_code "wfrec" ("\ <module>wfrec?")
  attach {* fun wfrec f x = f (wfrec f) x; *}

CODE FOR INDUCTIVE DEFINITIONS

Inductive definitions are Horn clauses:

$$(0, \mathrm{Suc}\ n) \in L$$

$$(n,m) \in L \Longrightarrow (\mathrm{Suc}\ n, \mathrm{Suc}\ m) \in L$$

Can be evaluated like Prolog

code_module T

contains x = "λx y. (x, y) $\in$ L"

y = "(_, 5) $\in$ L"

generates

- ➜ something of type bool for x
- ➜ a possibly infinite sequence for y, enumerating all suitable _ in (_, 5) $\in$ L

DEMO

WE HAVE SEEN TODAY ...

NICTA

- ➜ More fun
- ➜ Calculations: also/finally
- ➜ [trans]-rules
- ➜ Code generation