COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

a = b = c = . . .

Last time ...

- ➜ fun, function
- ➜ Well founded recursion

Content

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

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

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

Chains of equations

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

NICTA

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

Designing [trans] Rules

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

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 code

Translate HOL functional programming concepts, i.e.

- ➜ datatypes
- ➜ function definitions
- ➜ inductive predicates

into a stand-alone code in:

- ➜ SML
- ➜ Ocaml
- ➜ Haskell
- ➜ Scala

Syntax

export_code <definition_names> **in** SML

module_name <module_name> **file** “<file path>”

export_code <definition_names> **in** Haskell

module_name <module_name> **file** “<directory path>”

Takes a space-separated list of constants for which code shall be generated.

Anything else needed for those is added implicitly Generates ML stucture.

DEMO

Program Refinement

Aim: choosing appropriate code equations explicitly

Syntax:

lemma [code]:

<list of equations on function_name>

Example: more efficient definition of fibonnacci function

DEMO

Inductive Predicates

Inductive specifications turned into equational ones

Example:

```
append [] ys ys
append xs ys zs ⟹ append (x # xs ) ys (x # zs )
```

Syntax:

code_pred append .

DEMO

We have seen today ...

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