COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Toby Murray, June Andronick, Gerwin Klein

more Isar

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
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6, 7]
  - Hoare logic, proofs about programs, C verification [8[b],9]
  - (mid-semester break)
  - Writing Automated Proof Methods [10]
  - Isar, codegen, typeclasses, locales [11[c],12]

[a] a1 due; [b] a2 due; [c] a3 due

Last time ... Isar!

➜ syntax: proof, qed, assume, from, show, have, next

➜ modes: prove, state, chain

➜ backward/forward reasoning

➜ fix, obtain

➜ abbreviations: this, then, thus, hence, with, ?thesis

➜ moreover, ultimately

➜ case distinction

Today

➜ Datatypes in Isar

➜ Calculational reasoning

NICTA

Datatypes in Isar

Datatype case distinction

proof (cases $term$)

  case Constructor$_1$

  $\vdots$

next

$\vdots$

next

  case (Constructor$_k$ $\vec{x}$)

  $\cdots\ \vec{x}\ \cdots$

qed

case (Constructor$_i$ $\vec{x}$) $\equiv$

fix $\vec{x}$ **assume** Constructor$_i$: "$term =$ Constructor$_i$ $\vec{x}$"

Structural induction for type nat

```
show $P\ n$
proof (induct $n$)
    case $0$              $\equiv$  let $?case$ = $P\ 0$
    . . .
    show $?case$
next
    case (Suc $n$)        $\equiv$  fix $n$ assume Suc: $P\ n$
    . . .                           let $?case$ = $P$ (Suc $n$)
    $\cdots\ n\ \cdots$
    show $?case$
qed
```

Structural induction with $\Longrightarrow$ and $\bigwedge$

```
show "⋀x. A n ⟹ P n"
proof (induct n)
  case 0                ≡  fix x assume 0: "A 0"
                           let ?case = "P 0"
  . . .
  show ?case
next
  case (Suc n)          ≡  fix n and x
                           assume Suc: "⋀x. A n ⟹ P n"
                                       "A (Suc n)"
  . . .                    let ?case = "P (Suc n)"
  . . . n . . .
  . . .
  show ?case
qed
```

NICTA

DEMO: DATATYPES IN ISAR

CALCULATIONAL REASONING

The Goal

Prove:

$x \cdot x^{-1} = 1$

using:	assoc:	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
	left_inv:	$x^{-1} \cdot x = 1$
	left_one:	$1 \cdot x = x$

The Goal

Prove:

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

using:

assoc:	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
left_inv:	$x^{-1} \cdot x = 1$
left_one:	$1 \cdot x = x$

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

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

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

	calculation register
have ”$t_0 = t_1$” [proof]	
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: `print_trans_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

CODE GENERATION

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 ...

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