COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Simon Winwood, Toby Murray, June Andronick, Gerwin Klein

{P} ... {Q}

Slide 1

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
  - More recursion, Calculational reasoning
  - **Hoare logic, proofs about programs**
  - Locales, Presentation

Slide 2

Last Time

- ➜ Code generation
- ➜ Syntax of a simple imperative language
- ➜ Operational semantics
- ➜ Program proof on operational semantics

Slide 3

Proofs about Programs

Now we know:

- ➜ What programs are: Syntax
- ➜ On what they work: State
- ➜ How they work: Semantics

So we can prove properties about programs

Example:
Show that example program from last lecture implements the factorial.

$$\textbf{lemma}\ \langle \text{factorial}, \sigma \rangle \longrightarrow \sigma' \Longrightarrow \sigma' B = \text{fac}\ (\sigma A)$$

$$(\text{where}\quad \text{fac}\ 0 = 0, \quad \text{fac}\ (\text{Suc}\ n) = (\text{Suc}\ n) * \text{fac}\ n)$$

Slide 4

Too tedious

Induction needed for each loop

Is there something easier?

Floyd/Hoare

Idea: describe meaning of program by pre/post conditions

Examples:

$\{\mathsf{True}\} \quad x := 2 \quad \{x = 2\}$

$\{y = 2\} \quad x := 21 * y \quad \{x = 42\}$

$\{x = n\} \quad \mathsf{IF}\ y < 0\ \mathsf{THEN}\ x := x + y\ \mathsf{ELSE}\ x := x - y \quad \{x = n - |y|\}$

$\{A = n\} \quad \mathsf{factorial} \quad \{B = \mathsf{fac}\ n\}$

Proofs: have rules that directly work on such triples

Meaning of a Hoare-Triple

$$\{P\} \quad c \quad \{Q\}$$

What are the assertions P and Q?

- ➜ Here: again functions from state to bool (shallow embedding of assertions)
- ➜ Other choice: syntax and semantics for assertions (deep embedding)

What does $\{P\}\ c\ \{Q\}$ mean?

Partial Correctness:

$\models \{P\}\ c\ \{Q\} \quad \equiv \quad (\forall \sigma\ \sigma'.\ P\ \sigma \wedge \langle c, \sigma\rangle \longrightarrow \sigma' \Longrightarrow Q\ \sigma')$

Total Correctness:

$\models \{P\}\ c\ \{Q\} \quad \equiv \quad (\forall \sigma.\ P\ \sigma \Longrightarrow \exists \sigma'.\ \langle c, \sigma\rangle \longrightarrow \sigma' \wedge Q\ \sigma')$

This lecture: partial correctness only (easier)

Hoare Rules

$$\frac{}{\{P\} \quad \mathsf{SKIP} \quad \{P\}} \qquad \frac{}{\{P[x \mapsto e]\} \quad x := e \quad \{P\}}$$

$$\frac{\{P\}\ c_1\ \{R\} \quad \{R\}\ c_2\ \{Q\}}{\{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\} \quad \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2 \quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\} \quad P \wedge \neg b \Longrightarrow Q}{\{P\} \quad \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \mathsf{OD} \quad \{Q\}}$$

$$\frac{P \Longrightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Longrightarrow Q}{\{P\} \quad c \quad \{Q\}}$$

Hoare Rules

$$\frac{}{\vdash \{P\} \quad \text{SKIP} \quad \{P\}} \qquad \frac{}{\vdash \{\lambda\sigma.\ P\ (\sigma(x := e\ \sigma))\} \quad x := e \quad \{P\}}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c_1\ \{R\} \quad \vdash \{\lambda\sigma.\ P\ \sigma \wedge \neg b\ \sigma\}\ c_2\ \{Q\}}{\vdash \{P\} \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c\ \{P\} \quad \bigwedge\sigma.\ P\ \sigma \wedge \neg b\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\}}$$

$$\frac{\bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma \quad \vdash \{P'\}\ c\ \{Q'\} \quad \bigwedge\sigma.\ Q'\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad c \quad \{Q\}}$$

Slide 9

Nicer, but still kind of tedious

Hoare rule application seems boring & mechanical.

Automation?

Problem: While – need creativity to find right (invariant) P

Solution:

➜ annotate program with invariants

➜ then, Hoare rules can be applied automatically

Example:

$\{M = 0 \wedge N = 0\}$

WHILE $M \neq a$ INV $\{N = M * b\}$ DO $N := N + b; M := M + 1$ OD

$\{N = a * b\}$

Slide 11

Are the Rules Correct?

Soundness: $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \models \{P\}\ c\ \{Q\}$

Proof: by rule induction on $\vdash \{P\}\ c\ \{Q\}$

Demo: Hoare Logic in Isabelle

Slide 10

Weakest Preconditions

pre c Q = weakest P such that $\{P\}\ c\ \{Q\}$

With annotated invariants, easy to get:

pre SKIP Q	$=$	Q
pre $(x := a)$ Q	$=$	$\lambda\sigma.\ Q(\sigma(x := a\sigma))$
pre $(c_1; c_2)$ Q	$=$	pre c_1 (pre c_2 Q)
pre (IF b THEN c_1 ELSE c_2) Q	$=$	$\lambda\sigma.\ (b \longrightarrow$ pre c_1 Q $\sigma) \wedge$
		$(\neg b \longrightarrow$ pre c_2 Q $\sigma)$
pre (WHILE b INV I DO c OD) Q	$=$	I

Slide 12

Verification Conditions

$\{\text{pre } c\ Q\}\ c\ \{Q\}$ **only true under certain conditions**

These are called **verification conditions** vc $c\ Q$:

$$
\begin{array}{lcl}
\text{vc SKIP } Q & = & \text{True} \\
\text{vc } (x := a)\ Q & = & \text{True} \\
\text{vc } (c_1; c_2)\ Q & = & \text{vc } c_2\ Q \wedge (\text{vc } c_1\ (\text{pre } c_2\ Q)) \\
\text{vc } (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2)\ Q & = & \text{vc } c_1\ Q \wedge \text{vc } c_2\ Q \\
\text{vc } (\text{WHILE } b \text{ INV } I \text{ DO } c \text{ OD})\ Q & = & (\forall \sigma.\ I\sigma \wedge b\sigma \longrightarrow \text{pre } c\ I\ \sigma) \wedge \\
 & & (\forall \sigma.\ I\sigma \wedge \neg b\sigma \longrightarrow Q\ \sigma) \wedge \\
 & & \text{vc } c\ I
\end{array}
$$

$$\text{vc } c\ Q \wedge (\text{pre } c\ Q \Longrightarrow P) \Longrightarrow \{P\}\ c\ \{Q\}$$

Slide 13

Syntax Tricks

- ➜ $x := \lambda\sigma.\ 1$ instead of $x := 1$ sucks
- ➜ $\{\lambda\sigma.\ \sigma\ x = n\}$ instead of $\{x = n\}$ sucks as well

Problem: program variables are functions, not values

Solution: distinguish program variables syntactically

Choices:

- ➜ declare program variables with each Hoare triple
  - nice, usual syntax
  - works well if you state full program and only use vcg
- ➜ separate program variables from Hoare triple (use extensible records), indicate usage as function syntactically
  - more syntactic overhead
  - program pieces compose nicely

Slide 14

Records in Isabelle

Records are a tuples with named components

Example:

record A = a :: nat
b :: int

- ➜ Selectors: a :: A ⇒ nat, b :: A ⇒ int, a r = Suc 0
- ➜ Constructors: ⦇ a = Suc 0, b = −1 ⦈
- ➜ Update: r⦇ a := Suc 0 ⦈

Records are extensible:

record B = A +
c :: nat list

⦇ a = Suc 0, b = −1, c = [0, 0] ⦈

Slide 15

Arrays

Depending on language, model arrays as functions:

- ➜ Array access = function application:
  a[i] = a i
- ➜ Array update = function update:
  a[i] :== v = a :== a(i:= v)

Use lists to express length:

- ➜ Array access = nth:
  a[i] = a ! i
- ➜ Array update = list update:
  a[i] :== v = a :== a[i:= v]
- ➜ Array length = list length:
  a.length = length a

Slide 16

Pointers

Choice 1

datatype ref = Ref int | Null
types heap = int ⇒ val
datatype val = Int int | Bool bool | Struct_x int int bool | . . .

- ➜ hp :: heap, p :: ref
- ➜ Pointer access: *p = the_Int (hp (the_addr p))
- ➜ Pointer update: *p :== v = hp :== hp ((the_addr p) := v)

- ➜ a bit klunky
- ➜ gets even worse with structs
- ➜ lots of value extraction (the_Int) in spec and program

Pointers

Choice 2 (Burstall ’72, Bornat ’00)

struct with next pointer and element

datatype ref = Ref int | Null
types next_hp = int ⇒ ref
types elem_hp = int ⇒ int

- ➜ next :: next_hp, elem :: elem_hp, p :: ref
- ➜ Pointer access: p→next = next (the_addr p)
- ➜ Pointer update: p→next :== v = next :== next ((the_addr p) := v)

- ➜ a separate heap for each struct field
- ➜ buys you p→next ≠ p→elem automatically (aliasing)
- ➜ still assumes type safe language

DEMO

We have seen today ...

- ➜ Hoare logic rules
- ➜ Soundness of Hoare logic
- ➜ Verification conditions
- ➜ Example program proofs
- ➜ Arrays, pointers