COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods

$$\{P\} \ldots \{Q\}$$

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

LAST TIME

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

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.

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

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

Too tedious

Induction needed for each loop

Is there something easier?

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

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

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

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

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

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

Hoare Rules

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

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

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

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

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

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

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

$\{N = a * b\}$

WEAKEST PRECONDITIONS

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

With annotated invariants, easy to get:

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

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

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

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

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

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

RECORDS IN ISABELLE

Records are a tuples with named components

Example:

record A = a :: nat
b :: int

- ➜ Selectors: a :: A $\Rightarrow$ nat, b :: A $\Rightarrow$ int, a $r = \mathsf{Suc}\ 0$
- ➜ Constructors: $(\!|\ \mathsf{a} = \mathsf{Suc}\ 0,\ \mathsf{b} = -1\ |\!)$
- ➜ Update: $r(\!|\ \mathsf{a} := \mathsf{Suc}\ 0\ |\!)$

Records are extensible:

record B = A +
c :: nat list

$$(\!|\ \mathsf{a} = \mathsf{Suc}\ 0,\ \mathsf{b} = -1,\ \mathsf{c} = [0, 0]\ |\!)$$

Depending on language, model arrays as functions:

➜ Array access = function application:

$$a[i] \quad = \quad a\ i$$

➜ Array update = function update:

$$a[i] :== v \quad = \quad a :== a(i := v)$$

Use lists to express length:

➜ Array access = nth:

$$a[i] \quad = \quad a\ !\ i$$

➜ Array update = list update:

$$a[i] :== v \quad = \quad a :== a[i := v]$$

➜ Array length = list length:

$$a.\mathrm{length} \quad = \quad \mathrm{length}\ a$$

Choice 1

datatype	ref	= Ref int $\mid$ Null
types	heap	= int $\Rightarrow$ val
datatype	val	= Int int $\mid$ Bool bool $\mid$ Struct_x int int bool $\mid$ ...

➜ 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

Choice 2 (Burstall '72, Bornat '00)

struct with next pointer and element

datatype	ref	= Ref int $\mid$ Null
types	next_hp	= int $\Rightarrow$ ref
types	elem_hp	= int $\Rightarrow$ int

- ➜ next :: next_hp, elem :: elem_hp, p :: ref
- ➜ Pointer access: p$\rightarrow$next $\quad = \quad$ next (the_addr p)
- ➜ Pointer update: p$\rightarrow$next :== v $\quad = \quad$ next :== next ((the_addr p) := v)

- ➜ a separate heap for each struct field
- ➜ buys you p$\rightarrow$next $\neq$ p$\rightarrow$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