
COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

$\{P\} \dots \{Q\}$

Last Time

- Syntax of a simple imperative language
- Operational semantics
- Program proof on operational semantics
- Hoare logic rules
- Soundness of Hoare logic

Content

- Intro & motivation, getting started [1]

- Foundations & Principles
 - Lambda Calculus, natural deduction [1,2]
 - Higher Order Logic [3]
 - Term rewriting [4^a]

- Proof & Specification Techniques
 - Inductively defined sets, rule induction [5]
 - Datatypes, recursion, induction [6, 7]
 - Automated proof and disproof [7]
 - Hoare logic, proofs about programs, refinement [8^b,9^c,10]
 - Isar, locales [11^d,12]

^a a1 due; ^b a2 due; ^c session break; ^d a3 due

Automation?

Last time: Hoare rule application is nicer than using operational semantic.

BUT:

- it's still kind of tedious
- it seems boring & mechanical

Automation?

Invariant

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:

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

Verification Conditions

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

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

$$\text{vc SKIP } Q = \text{True}$$

$$\text{vc } (x := a) \ Q = \text{True}$$

$$\text{vc } (c_1; c_2) \ Q = \text{vc } c_2 \ Q \wedge (\text{vc } c_1 \ (\text{pre } c_2 \ Q))$$

$$\text{vc } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q = \text{vc } c_1 \ Q \wedge \text{vc } c_2 \ Q$$

$$\begin{aligned} \text{vc } (\text{WHILE } b \ \text{INV } I \ \text{DO } c \ \text{OD}) \ Q &= (\forall \sigma. I \sigma \wedge b \sigma \longrightarrow \text{pre } c \ I \ \sigma) \wedge \\ &(\forall \sigma. I \sigma \wedge \neg b \sigma \longrightarrow Q \ \sigma) \wedge \\ &\text{vc } c \ I \end{aligned}$$

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

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

DEMO

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:

$$a.length = length \ a$$

Pointers

Choice 1

datatype ref = Ref int | Null

types heap = int \Rightarrow 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 \Rightarrow ref

types elem_hp = int \Rightarrow int

→ next :: next_hp, elem :: elem_hp, p :: ref

→ Pointer access: $p \rightarrow \text{next} = \text{next } (\text{the_addr } p)$

→ Pointer update: $p \rightarrow \text{next} ::= v = \text{next} ::= \text{next } ((\text{the_addr } p) ::= v)$

→ a separate heap for each struct field

→ buys you $p \rightarrow \text{next} \neq p \rightarrow \text{elem}$ automatically (aliasing)

→ still assumes type safe language

DEMO

We have seen today ...

- Weakest precondition
- Verification conditions
- Example program proofs
- Arrays, pointers