COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray

{P} ... {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

➜ Foundations & Principles
  • Lambda Calculus, natural deduction [2,3,4]
  • Higher Order Logic [5,6,7]
  • Term rewriting [8,9,10]

➜ Proof & Specification Techniques
  • Isar [11,12]
  • Inductively defined sets, rule induction [13,15]
  • Datatypes, recursion, induction [16,17,18,19]
  • Calculational reasoning, mathematics style proofs [20]
  • Hoare logic, proofs about programs [21,22,23]

---

\( a \) a1 out; \( b \) a1 due; \( c \) a2 out; \( d \) a2 due; \( e \) session break; \( f \) a3 out; \( g \) a3 due
Last time: Hoare rule application is nicer than using operational semantic.

BUT:

→ it’s still kind of tedious
→ it seems boring & mechanical
Invariant

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

**Solution:**

- annotate program with invariants
- then, Hoare rules can be applied automatically

**Example:**

$$\{M = 0 \land N = 0\} \quad \text{WHILE} \ M \neq a \ \text{INV} \ \{N = M \times b\} \ \text{DO} \ N := N + b; \ M := M + 1 \ \text{OD} \ \{N = a \times b\}$$
Weakest Preconditions

\[ \text{pre } c \ Q = \text{weakest } P \text{ such that } \{P\} \ c \ \{Q\} \]

With annotated invariants, easy to get:

\[
\begin{align*}
\text{pre } \text{SKIP } Q & = Q \\
\text{pre } (x := a) \ Q & = \lambda \sigma. \ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1 ; c_2) \ Q & = \text{pre } c_1 \ (\text{pre } c_2 \ Q) \\
\text{pre } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q & = \lambda \sigma. \ (b \rightarrow \text{pre } c_1 \ Q \ \sigma) \land \\
& \quad (\neg b \rightarrow \text{pre } c_2 \ Q \ \sigma) \\
\text{pre } (\text{WHILE } b \ \text{INV } I \ \text{DO } c \ \text{OD}) \ Q & = I
\end{align*}
\]
Verification Conditions

{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
- $vc$ $(c_1; c_2)$ $Q$ = $vc$ $c_2$ $Q$ $\land$ ($vc$ $c_1$ (pre $c_2$ $Q$))
- $vc$ $(IF$ $b$ $THEN$ $c_1$ ELSE $c_2)$ $Q$ = $vc$ $c_1$ $Q$ $\land$ $vc$ $c_2$ $Q$
- $vc$ $(WHILE$ $b$ $INV$ $I$ $DO$ $c$ $OD)$ $Q$ = ($\forall \sigma. I\sigma \land b\sigma \rightarrow$ pre $c$ $I\sigma$)$\land$
  ($\forall \sigma. I\sigma \land \neg b\sigma \rightarrow$ $Q\sigma$)$\land$
  $vc$ $c$ $I$

$$vc$ $c$ $Q$ $\land$ ($P \rightarrow$ pre $c$ $Q$) $\rightarrow$ {$P$} $c$ {$Q$}
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:

```plaintext
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:

```plaintext
record B = A +
  c :: nat list

( | a = Suc 0, b = −1, c = [0,0] |)
```
Arrays

Depending on language, model arrays as functions:

- Array access = function application:
  \[a[i] = a \ s i\]

- 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:
  \[a.length = \text{length } a\]
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
Choice 2 (Burstall ’72, Bornat ’00)

struct with next pointer and element

```plaintext
datatype  ref    = Ref int | Null
types     next_hp = int ⇒ ref
          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 ...

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