COMP4161: Advanced Topics in Software Verification

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

Gerwin Klein, June Andronick, Ramana Kumar, Miki Tanaka
S2/2017

data61.csiro.au
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 [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]

^a1 due; ^b2 due; ^c3 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:

$\rightarrow$ annotate program with invariants

$\rightarrow$ then, Hoare rules can be applied automatically

Example:

\[
\begin{align*}
\{ M = 0 \land N = 0 \} \\
\text{WHILE } M \neq a \text{ INV } \{ N = M \ast b \} \text{ DO } N := N + b; M := M + 1 \text{ OD } \}
\{ N = a \ast b \}
\end{align*}
\]
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

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

These are called verification conditions \( vc \ c \ Q \):

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

\[
vc \ c \ Q \land (P \rightarrow \text{pre } c \ Q) \rightarrow \{P\} \ c \ \{Q\}
\]
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
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:
  \[ a.length = \text{length } a \]
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
Pointers

Choice 2 (Burstall ’72, Bornat ’00)

Example: struct with next pointer and element

```plaintext
datatype ref = Ref int | Null

next_hp = int ⇒ ref

next_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)
```

In general:

- 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