MP4161

Advanced Topics in Software Verification

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

Gerwin Klein, June Andronick, Miki Tanaka, Johannes Åman Pohjola

T3/2022

Last Time

- ➜ Syntax of a simple [illegible] language
- ➜ Operational semant[illegible]
- ➜ Program proof on d[illegible] semantics
- ➜ Hoare logic rules
- ➜ Soundness of Hoare logic

Content

- ➜ Foundations & Principles
  - Intro, Lambda [illegible]atural deduction [1,2]
  - Higher Order [illegible] part 1) [2,3[a]]
  - Term rewriting [illegible] [3,4]

- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction [4,5]
  - Datatype induction, primitive recursion [5,7]
  - General recursive functions, termination proofs [7[b]]
  - Proof automation, Isar (part 2) [8]
  - Hoare logic, proofs about programs, invariants [8,9]
  - C verification [9,10]
  - Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

Automation?

Last time: Hoare rule [illegible] is nicer than using operational semantics.

BUT:

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

Automation?

Invariant

Invariant

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

Invariant

Problem: While – nee[illegible] to find right (invariant) P

Solution:

➜ annotate program w[illegible]ts

Invariant

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

Solution:

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

Invariant

Problem: While – nee[illegible]y to find right (invariant) P

Solution:

- ➜ annotate program w[illegible]ts
- ➜ 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:

pre SKIP Q $=$

Weakest Preconditions

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

With annotated invaria[illegible]o get:

pre SKIP Q = Q

pre $(x := a)$ Q =

Weakest Preconditions

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

With annotated invariants, easy to get:

$$
\begin{array}{lcl}
\text{pre SKIP } Q & = & Q \\
\text{pre } (x := a)\ Q & = & \lambda\sigma.\ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1; c_2)\ Q & = &
\end{array}
$$

Weakest Preconditions

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

With annotated invaria[illegible]o get:

$$\begin{array}{lcl}
\text{pre SKIP } Q & = & Q \\
\text{pre } (x := a)\ Q & = & \lambda\sigma.\ Q(\sigma(x := a\sigma)) \\
\text{pre } (c_1; c_2)\ Q & = & \text{pre } c_1\ (\text{pre } c_2\ Q) \\
\text{pre } (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2)\ Q & = &
\end{array}$$

Weakest Preconditions

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

With annotated invaria[illegible]o get:

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

Weakest Preconditions

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

With annotated invariants, easy to get:

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

Verification Conditions

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

Verification Conditions

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

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

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

Verification Conditions

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

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

$$
\begin{array}{lcl}
\text{vc SKIP } Q & = & \text{True} \\
\text{vc } (x := a)\ Q & = & \text{True}
\end{array}
$$

Verification Conditions

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

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

$$
\begin{array}{lcl}
\text{vc SKIP } Q & = & \text{True} \\
\text{vc } (x := a)\ Q & = & \text{True} \\
\text{vc } (c_1; c_2)\ Q & = & \text{vc } c_2\ Q \wedge (\text{vc } c_1\ (\text{pre } c_2\ Q))
\end{array}
$$

Verification Conditions

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

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

$$
\begin{array}{lcl}
\text{vc SKIP } Q & = & \text{True} \\
\text{vc } (x := a) \; Q & = & \text{True} \\
\text{vc } (c_1; c_2) \; Q & = & \text{vc } c_2 \; Q \wedge (\text{vc } c_1 \; (\text{pre } c_2 \; Q)) \\
\text{vc } (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) \; Q & = & \text{vc } c_1 \; Q \wedge \text{vc } c_2 \; Q
\end{array}
$$

Verification Conditions

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

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

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

Verification Conditions

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

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

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

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

Syntax Tricks

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

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

Syntax Tricks

- $x := \lambda\sigma.\ 1$ instead of $x := 1$ sucks
- $\{\lambda\sigma.\ \sigma\ x = n\}$ in[illegible] $\{x = n\}$ sucks as well

Problem: program var[illegible] functions, not values

Solution: distinguish program variables syntactically

Syntax Tricks

- $x := \lambda\sigma.\ 1$ instead of $x := 1$ sucks
- $\{\lambda\sigma.\ \sigma\ x = n\}$ in[illegible] $\{x = n\}$ sucks as well

Problem: program var[illegible]unctions, not values

Solution: distinguish program variables syntactically

Choices:

- declare program variables with each Hoare triple

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

Syntax Tricks

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

Problem: program var 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

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)

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 = … Null
types heap = …
datatype val = … Bool bool | Struct_x int int bool | …

Pointers

Choice 1

datatype	ref	= ... Null
types	heap	= ...
datatype	val	= ... 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)

Pointers

Choice 1

datatype ref = ... Null
types heap = ...
datatype val = ... 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

datatype	ref	= Ref int | Null
types	next_hp	$= \text{int} \Rightarrow \text{ref}$
types	elem_hp	$= \text{int} \Rightarrow \text{int}$

Pointers

Choice 2 (Burstall '72, Bornat '00)

Example: struct with [illegible]er and element

datatype	ref	[illegible]t $\mid$ 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 $=$ next (the_addr p)
- ➜ Pointer update: p$\rightarrow$next := v $=$ next := next ((the_addr p) := v)

Pointers

Choice 2 (Burstall '72, Bornat '00)

Example: struct with [illegible]er and element

datatype	ref	= [illegible]t | 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 $=$ next (the_addr p)
- ➜ Pointer update: p$\rightarrow$next :== v $=$ next :== next ((the_addr p) := v)

In general:

- ➜ 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 ...

- Weakest preconditi
- Verification conditi
- Example program p
- Arrays, pointers