COMP4161: Advanced Topics in Software Verification

{P} . . . {Q}

Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka
T3/2020
Last Time

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

→ Foundations & Principles
  • Intro, Lambda calculus, natural deduction [1,2]
  • Higher Order Logic, Isar (part 1) [2,3\textsuperscript{a}]
  • Term rewriting [3,4]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
  • General recursive functions, termination proofs [7\textsuperscript{b}]
  • Proof automation, Hoare logic, proofs about programs, invariants [8]
  • C verification [9,10]
  • Practice, questions, examp prep [10\textsuperscript{c}]

\textsuperscript{a}a1 due; \textsuperscript{b}a2 due; \textsuperscript{c}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) $P$
Invariant

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

**Solution:**
- annotate program with invariants
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 – need creativity to find right (invariant) \( P \)

Solution:

\( \Rightarrow \) annotate program with invariants
\( \Rightarrow \) then, Hoare rules can be applied automatically

Example:

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

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

With annotated invariants, easy to get:

pre SKIP Q

pre (x := a) Q = \lambda \sigma. Q(\sigma(x := a \sigma))

pre (c_1; c_2) Q = pre c_1 (pre c_2 Q)

pre (IF b THEN c_1 ELSE c_2) Q = \lambda \sigma. (b \sigma \rightarrow pre c_1 Q \sigma) \land (\neg b \sigma \rightarrow pre c_2 Q \sigma)

pre (WHILE b INV I DO c OD) Q = I
Weakest Preconditions

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

With annotated invariants, easy to get:

\[
\begin{align*}
\text{pre } \text{SKIP } Q & = Q \\
\text{pre } (x := a) \ Q & = \lambda \sigma. Q(\sigma(x := a \sigma))
\end{align*}
\]
Weakest Preconditions

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

With annotated invariants, easy to get:

pre SKIP $Q$ \hspace{1cm} = \hspace{1cm} Q
pre $(x := a) \ Q$ \hspace{1cm} = \hspace{1cm} $\lambda\sigma. \ Q(x := a\sigma)$
pre $(c_1; c_2) \ Q$ \hspace{1cm} = \hspace{1cm}
Weakest Preconditions

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

With annotated invariants, easy to get:

- \[ \text{pre skip } Q = Q \]
- \[ \text{pre } (x := a) \ Q = \lambda \sigma. Q(\sigma(x := a\sigma)) \]
- \[ \text{pre } (c_1; c_2) \ Q = \text{pre } c_1 \ (\text{pre } c_2 \ Q) \]
- \[ \text{pre } (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \ Q = \]

6 | COMP4161 | CC-BY-4.0 License
Weakest Preconditions

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

With annotated invariants, easy to get:

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

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

With annotated invariants, easy to get:

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

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

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

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

\[
vc \ \text{SKIP} \ Q \quad = \quad \text{True}
\]
Verification Conditions

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

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

\[
\begin{align*}
vc \ \text{SKIP} \ Q & = \text{True} \\
vc \ (x := a) \ Q & = \text{True}
\end{align*}
\]
Verification Conditions

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

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

\[
\begin{align*}
vc \ SKIP \ Q & = True \\
vc \ (x := a) \ Q & = True \\
vc \ (c_1; c_2) \ Q & = vc c_2 \ Q \land (vc c_1 (\text{pre } c_2 \ Q))
\end{align*}
\]
Verification Conditions

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

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

\begin{align*}
\text{vc } \text{SKIP} \ Q & = \text{True} \\
\text{vc } (x := a) \ Q & = \text{True} \\
\text{vc } (c_1; c_2) \ Q & = \text{vc } c_2 \ Q \land (\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 \land \text{vc } c_2 \ Q
\end{align*}
Verification Conditions

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

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

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

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

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

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

\[
vc \ c \ Q \land (P \rightarrow \ pre \ c \ Q) \rightarrow \ {P} \ c \ {Q}
\]
Syntax Tricks

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

**Problem:** program variables are functions, not values

**Solution:** distinguish program variables 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
Syntax Tricks

\[ x := \lambda \sigma. 1 \quad \text{instead of} \quad x := 1 \quad \text{sucks} \]
\[ \{ \lambda \sigma. \sigma x = n \} \quad \text{instead of} \quad \{ x = n \} \quad \text{sucks as well} \]

**Problem:** program variables are functions, not values

**Solution:** distinguish program variables syntactically

**Choices:**

\[ \rightarrow \text{declare program variables with each Hoare triple} \]
  - 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

→ $\{\lambda \sigma. \sigma \ x = n\}$ 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
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:
  \[ a.length = \text{length } a \]
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 | ...
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)
Pointers

Choice 1

```haskell
 datatype ref = Ref int | Null
 types heap = int \rightarrow val
 datatype val = Int int | Bool bool | Struct x int int bool | ...

\rightarrow hp :: heap, p :: ref
\rightarrow Pointer access: *p = the_int (hp (the_addr p))
\rightarrow Pointer update: *p := v = hp := hp ((the_addr p) := v)

\rightarrow a bit klunky
\rightarrow gets even worse with structs
\rightarrow lots of value extraction (the_int) in spec and program
```
Pointers

Choice 2 (Burstall ’72, Bornat ’00)

Example: struct with next pointer and element

\[
\begin{align*}
\text{datatype} & \quad \text{ref} \quad = \text{Ref int} \mid \text{Null} \\
\text{types} & \quad \text{next_hp} \quad = \text{int} \Rightarrow \text{ref} \\
\text{types} & \quad \text{elem_hp} \quad = \text{int} \Rightarrow \text{int}
\end{align*}
\]
Pointers

Choice 2 (Burstall ’72, Bornat ’00)

Example: struct with next pointer and element

```
<table>
<thead>
<tr>
<th>datatype</th>
<th>ref</th>
<th>= Ref int</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>next_hp</td>
<td>= int ⇒ ref</td>
<td></td>
</tr>
<tr>
<td>types</td>
<td>elem_hp</td>
<td>= int ⇒ int</td>
<td></td>
</tr>
</tbody>
</table>

⇒ 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)
```
Pointers

Choice 2 (Burstall ’72, Bornat ’00)

Example: struct with next pointer and element

```
datatype ref = Ref int | Null

types next_hp = int ⇒ ref

types 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)

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