COMP4161: Advanced Topics in Software Verification

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

Gerwin Klein, June Andronick, Ramana Kumar

S2/2016

# Last Time

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Content

➜ Intro & motivation, getting started [1]

➜ 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]

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Automation?

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

**BUT:**

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

**Automation?**

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Invariant

# Invariant

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Invariant

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

**Solution:**

➜ annotate program with invariants

© Data61, CSIRO: provided under Creative Commons Attribution License

# Invariant

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

**Solution:**

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Invariant

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

**Solution:**

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

**Example:**

$$\{M = 0 \wedge N = 0\}$$
$$\text{WHILE } M \neq a \text{ INV } \{N = M * b\} \text{ DO } N := N + b; M := M + 1 \text{ OD}$$
$$\{N = a * b\}$$

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

pre SKIP $Q$ =

© Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Weakest Preconditions

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

With annotated invariants, easy to get:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Verification Conditions

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Verification Conditions

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

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Verification Conditions

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

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Verification Conditions

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

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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 $c$ $Q$:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Verification Conditions

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

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

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Syntax Tricks

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

© Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

 © Data61, CSIRO: provided under Creative Commons Attribution License

# Pointers

### Choice 1

| | | |
|---|---|---|
| **datatype** | ref | $=$ Ref int $\mid$ Null |
| **types** | heap | $=$ int $\Rightarrow$ val |
| **datatype** | val | $=$ Int int $\mid$ Bool bool $\mid$ Struct_x int int bool $\mid \ldots$ |

© Data61, CSIRO: provided under Creative Commons Attribution License

# 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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

© Data61, CSIRO: provided under Creative Commons Attribution License

# 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 |

© Data61, CSIRO: provided under Creative Commons Attribution License

# 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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

# 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

© Data61, CSIRO: provided under Creative Commons Attribution License

# Demo

# We have seen today ...

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

 © Data61, CSIRO: provided under Creative Commons Attribution License