COMP4011/8011
Advanced Topics in
Formal Methods and Programming Languages

– Software Verification with Isabelle/HOL –

Peter Höfner

September 22, 2024

Section 16

Floyd-Hoare Logic

Semantics (A Crash Course)

Further Details

- see Concrete Semantics
- COMP3610/6361 Principles of Programming Languages
  `https://comp.anu.edu.au/courses/comp3610/`

IMP - a small Imperative Language

Commands:

datatype com	=	SKIP	
	|	Assign vname aexp	(_ := _)
	|	Semi com com	(_; _)
	|	Cond bexp com com	(IF _ THEN _ ELSE _)
	|	While bexp com	(WHILE _ DO _ OD)

type_synonym vname = string
type_synonym state = vname $\Rightarrow$ nat

type_synonym aexp = state $\Rightarrow$ nat
type_synonym bexp = state $\Rightarrow$ bool

Example Program

Usual syntax:

$$
\begin{array}{l}
B := 1; \\
\text{WHILE } A \neq 0 \text{ DO} \\
\quad B := B * A; \\
\quad A := A - 1 \\
\text{OD}
\end{array}
$$

Expressions are functions from state to bool or nat:

$$
\begin{array}{l}
B := (\lambda\sigma.\ 1); \\
\text{WHILE } (\lambda\sigma.\ \sigma\ A \neq 0) \text{ DO} \\
\quad B := (\lambda\sigma.\ \sigma\ B * \sigma\ A); \\
\quad A := (\lambda\sigma.\ \sigma\ A - 1) \\
\text{OD}
\end{array}
$$

What does it do?

So far we have defined:

- Syntax of commands and expressions
- State of programs (function from variables to values)

Now we need: the meaning (semantics) of programs

How to define execution of a program?

- A wide field of its own
- Some choices:
  - Operational (inductive relations, big step, small step)
  - Denotational (programs as functions on states, state transformers)
  - Axiomatic (pre-/post conditions, Hoare logic)

Structural Operational Semantics

$$\frac{}{\langle \mathsf{SKIP}, \sigma \rangle \to \sigma}$$

$$\frac{e\ \sigma = v}{\langle \mathsf{x} := \mathsf{e}, \sigma \rangle \to \sigma[x \mapsto v]}$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''}$$

$$\frac{b\ \sigma = \mathsf{True} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

$$\frac{b\ \sigma = \mathsf{False} \quad \langle c_2, \sigma \rangle \to \sigma'}{\langle \mathsf{IF}\ b\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2, \sigma \rangle \to \sigma'}$$

Structural Operational Semantics

$$\frac{b\ \sigma = \mathsf{False}}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \mathsf{OD}, \sigma\rangle \rightarrow \sigma}$$

$$\frac{b\ \sigma = \mathsf{True} \quad \langle c, \sigma\rangle \rightarrow \sigma' \quad \langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \mathsf{OD}, \sigma'\rangle \rightarrow \sigma''}{\langle \mathsf{WHILE}\ b\ \mathsf{DO}\ c\ \mathsf{OD}, \sigma\rangle \rightarrow \sigma''}$$

Demo: The Definitions in Isabelle

Proofs about Programs

Now we know:

- What programs are: Syntax
- On what they work: State
- How they work: Semantics

So we can prove properties about programs

Example:
Show that example program from earlier implements the factorial.

$$\textbf{lemma}\ \langle \mathsf{factorial}, \sigma \rangle \rightarrow \sigma' \Longrightarrow \sigma' B = \mathsf{fac}\ (\sigma A)$$
$$\text{(where}\quad \mathsf{fac}\ 0 = 1, \quad \mathsf{fac}\ (\mathsf{Suc}\ n) = (\mathsf{Suc}\ n) * \mathsf{fac}\ n\text{)}$$

Demo: Example Proof

Too tedious

Induction needed for each loop

Is there something easier?

Floyd-Hoare Logic

Floyd-Hoare Logic

Idea: describe meaning of program by pre/post conditions

Examples:

$\{\mathrm{True}\} \quad x := 2 \quad \{x = 2\}$

$\{y = 2\} \quad x := 21 * y \quad \{x = 42\}$

$\{x = n\} \quad \text{IF } y < 0 \text{ THEN } x := x + y \text{ ELSE } x := x - y \quad \{x = n - |y|\}$

$\{A = n\} \quad \text{factorial} \quad \{B = \text{fac } n\}$

Proofs: have rules that directly work on such triples

Meaning of a Hoare-Triple

$$\{P\} \quad c \quad \{Q\}$$

What are the assertions P and Q?

- Here: again functions from state to bool (shallow embedding of assertions)
- Other choice: syntax and semantics for assertions (deep embedding)

What does $\{P\}\ c\ \{Q\}$ mean?

Partial Correctness:

$$\models \{P\}\ c\ \{Q\} \quad \equiv \quad \forall \sigma\ \sigma'.\ P\ \sigma \wedge \langle c, \sigma\rangle \rightarrow \sigma' \longrightarrow Q\ \sigma'$$

Total Correctness:

$$\models \{P\}\ c\ \{Q\} \quad \equiv \quad (\forall \sigma\ \sigma'.\ P\ \sigma \wedge \langle c, \sigma\rangle \rightarrow \sigma' \longrightarrow Q\ \sigma') \wedge$$
$$(\forall \sigma.\ P\ \sigma \longrightarrow \exists \sigma'.\ \langle c, \sigma\rangle \rightarrow \sigma')$$

This lecture: partial correctness only (easier)

Hoare Rules

$$\frac{}{\{P\} \quad \text{SKIP} \quad \{P\}} \qquad \frac{}{\{P[x \mapsto e]\} \quad x := e \quad \{P\}}$$

$$\frac{\{P\}\ c_1\ \{R\} \quad \{R\}\ c_2\ \{Q\}}{\{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\} \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\} \quad P \wedge \neg b \Longrightarrow Q}{\{P\} \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\}}$$

$$\frac{P \Longrightarrow P' \quad \{P'\}\ c\ \{Q'\} \quad Q' \Longrightarrow Q}{\{P\} \quad c \quad \{Q\}}$$

Hoare Rules

$$\frac{}{\vdash \{P\} \quad \text{SKIP} \quad \{P\}} \qquad \frac{}{\vdash \{\lambda\sigma.\ P\ (\sigma(x := e\ \sigma))\} \quad x := e \quad \{P\}}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \quad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\} \quad c_1; c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c_1\ \{Q\} \quad \vdash \{\lambda\sigma.\ P\ \sigma \wedge \neg b\ \sigma\}\ c_2\ \{Q\}}{\vdash \{P\} \quad \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \quad \{Q\}}$$

$$\frac{\vdash \{\lambda\sigma.\ P\ \sigma \wedge b\ \sigma\}\ c\ \{P\} \quad \bigwedge\sigma.\ P\ \sigma \wedge \neg b\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad \text{WHILE } b \text{ DO } c \text{ OD} \quad \{Q\}}$$

$$\frac{\bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma \quad \vdash \{P'\}\ c\ \{Q'\} \quad \bigwedge\sigma.\ Q'\ \sigma \Longrightarrow Q\ \sigma}{\vdash \{P\} \quad c \quad \{Q\}}$$

Are the Rules Correct?

Soundness: $\vdash \{P\}\ c\ \{Q\} \Longrightarrow \models \{P\}\ c\ \{Q\}$

Proof: by rule induction on $\vdash \{P\}\ c\ \{Q\}$

Demo: Hoare Logic in Isabelle

We have seen ...

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

Automation?

Hoare rule application is nicer than using operational semantics.

BUT:

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

Automation?

Invariant

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

Solution:

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

Example:

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

Weakest Preconditions

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

With annotated invariants, easy to get:

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

Verification Conditions

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

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

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 (ext. 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 $=$ length a

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)

- a bit clunky
- 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	= 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 ...

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