SEPARATION LOGICN

KV Software Verification WS 18/19

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

The Classics: Hoare Calculus

Hoare, Charles Antony Richard. "An axiomatic basis for computer programming." Communications of the ACM 12.10 (1969): 576-580, citations: 7923

Hoare Triple

$$\{F\}\ \mathsf{P}\ \{G\}$$

precondition program postcondiction

Hoare Triple

$$\{F\} \quad \mathsf{P} \quad \{G\}$$

precondition program postcondiction

Informal meaning of a Hoare Triple:

If program is run in a state that satisfies F, then the state that results from P’s execution will satisfy G.

Hoare Triple

$$\{F\}\ \mathsf{P}\ \{G\}$$

precondition program postcondiction

Informal meaning of a Hoare Triple:

If program is run in a state that satisfies F, then the state that results from P's execution will satisfy G.

Questions

- How to specify program P?
- How to specify conditions F, G?
- How to do the reasoning?

The Simple Programming Language WHILE

P	::=	**skip**	no operation
	$\mid$	$P_1; P_2$	sequential composition
	$\mid$	$V := E$	assignment
	$\mid$	**if** B **then** P_1 **else** P_2	branching
	$\mid$	**while** B **do** P	loop

where

- E is an arithmetical expression
- B is a Boolean expression

Example: Swap

Consider the following simple program

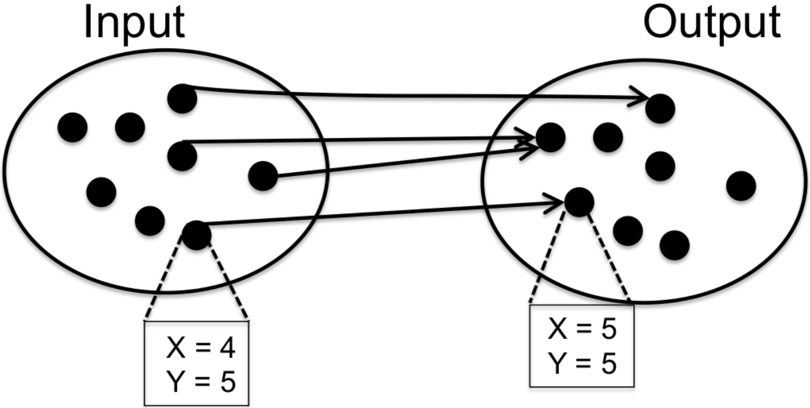
$$T := X; X := Y; Y := T$$

that is supposed to exchange the values of variables X and Y.

To show:

$$\{X = v_X \land Y = v_Y\}\ T := X; X := Y; Y := T\ \{X = v_Y \land Y = v_X\}$$

Programs as State Transformations



Program States: “Memory Snapshots”

Program state

mapping $\sigma\colon \mathsf{Var} \to \mathbb{Z}$ of variables from set of variables Var to values (integer)

Set of all states

$S = \{\sigma \mid \sigma\colon \mathsf{Var} \to \mathbb{Z}\}$

Example

A possible state of variables $\mathsf{Var} = \{x, y, z\}$ is $\sigma(x) = 1, \sigma(y) = 2, \sigma(z) = 3$

Configuration

- pair (P, σ) with $P \in prog$ and $\sigma \in S$
- state $\sigma \in S$ (final configuration)

Set of all configurations

$\mathsf{Configs} = (P \times S) \cup S$

Semantics of Expressions

for assignments, expressions E have to be evaluated. Therefore, we define the function $[\![-]\!]\colon \text{Exp} \times S \to \mathbb{Z}$ as follows:

$$\begin{array}{lcl}
[\![V]\!]\sigma & = & \sigma(V) \\
[\![n]\!]\sigma & = & n \in \mathbb{Z} \\
[\![E_1 + E_2]\!]\sigma & = & [\![E_1]\!]\sigma + [\![E_2]\!]\sigma \\
\ldots & &
\end{array}$$

for loop- and if-statements, Boolean expressions have to be evaluated. Therefore, we define the function $[\![-]\!]\colon \text{BExp} \times S \to \mathbb{B}$ as follows:

$$\begin{array}{lcl}
[\![\top]\!]\sigma & = & \text{true} \\
[\![\bot]\!]\sigma & = & \text{false} \\
[\![E_1 == E_2]\!]\sigma & = & [\![E_1]\!]\sigma == [\![E_2]\!]\sigma \\
[\![B_1 \wedge B_2]\!]\sigma & = & [\![B_1]\!]\sigma \wedge [\![B_2]\!]\sigma \\
\ldots & &
\end{array}$$

Program Semantics

The small-step semantics of programs is defined by the relation $\rightsquigarrow$: Configs × Configs which is defined as follows:

$$
\begin{array}{lcl}
(\textbf{skip}, \sigma) & \rightsquigarrow & \sigma \\
(P_1; P_2, \sigma) & \rightsquigarrow & \begin{cases} (P_1'; P2, \sigma') & \text{if } (P_1, \sigma) \rightsquigarrow (P_1', \sigma') \\ (P_2, \sigma') & \text{if } (P_1, \sigma) \rightsquigarrow \sigma' \end{cases} \\
V := E & \rightsquigarrow & \sigma' \text{ with } \begin{array}{rl} \sigma'(V) = & [\![E]\!]\sigma \\ \sigma'(X) = & \sigma(X) \text{ for } X \neq V \end{array} \\
\textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2 & \rightsquigarrow & \begin{cases} (P_1, \sigma) & \text{if } [\![B]\!]\sigma \text{ is true} \\ (P_2, \sigma) & \text{otherwise} \end{cases} \\
\textbf{while } B \textbf{ do } P & \rightsquigarrow & \begin{cases} (P; \textbf{while } B \textbf{ do } P, \sigma) & \text{if } [\![B]\!]\sigma \text{ is true} \\ \sigma & \text{otherwise} \end{cases}
\end{array}
$$

Function $[\![-]\!]$: Prog $\times\, S \to S$ describes the computation performed by a program P starting in state σ as follows:

$$[\![P]\!]\sigma = \sigma' \text{ iff } (P, \sigma) \rightsquigarrow^* \sigma'$$

Specification of the Conditions

- formulas in first-order logic (FO) with usual syntax and semantics
- $\sigma \vdash F$ means: F holds in state σ
- $\{F\} = \{\sigma \mid \sigma \vdash F\}$

Correctness Assertions:

$$\{F\}\ P\ \{G\}$$

holds iff for all states $\sigma \in S$, if

1. $\sigma \vdash F$
2. $(P, \sigma) \rightsquigarrow^* \sigma'$

then $\sigma' \vdash G$

Hoare Calculus: Rule Schemas

Inference rule:

$$\frac{S_1 \qquad \ldots \qquad S_n}{S}$$

S can be derived from assumptions S_1, …, S_n

Axiom rule:

$$\frac{}{S}$$

Inference rule without any assumption.

Proof tree of S:

Derivation with S in the root and only axioms in the leaves.

Rules of the Hoare Calculus

$$\frac{}{\{F\}\ \textbf{skip}\ \{F\}} \qquad \frac{}{\{F[E/V]\}\ V := E\ \{F\}}$$

$$\frac{\{F\}\ P_1\ \{H\} \qquad \{H\}\ P_2\{G\}}{\{F\}\ P_1; P_2\{G\}}$$

$$\frac{\{F \land B\}\ P_1\ \{G\} \qquad \{F \land \neg B\}\ P_2\{G\}}{\{F\}\ \textbf{if}\ B\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \{G\}}$$

$$\frac{\{F \land B\}\ P\{F\}}{\{F\}\ \textbf{while}\ B\ \textbf{do}\ P\ \{F \land \neg B\}}$$

Rules of the Hoare Calculus

$$\frac{\vdash F \rightarrow F' \qquad \{F'\}\ P\ \{G'\} \qquad \vdash G' \rightarrow G}{\{F\}\ P\ \{G\}}$$

$$\frac{\{F_1\}\ P\ \{G\} \qquad \{F_2\}\ P\{G\}}{\{F_1 \vee F_2\}\ P\ \{G\}}$$

$$\frac{\{F\}\ P\ \{G_1\} \qquad \{F\}\ P\{G_2\}}{\{F\}\ P\ \{G_1 \wedge G_2\}}$$

Separation Logic

Reynolds, John C. "Separation logic: A logic for shared mutable data structures." Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. IEEE, 2002, citations: 2317

O'Hearn, Peter, John Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures." International Workshop on Computer Science Logic. Springer, Berlin, Heidelberg, 2001, citations: 758

Extension of WHILE Language

We consider an extension of WHILE with pointers, memory allocation, and memory deallocation

P	::=	**skip**	no operation
	$\mid$	$P_1 ; P_2$	sequential composition
	$\mid$	$V := E$	assignment
	$\mid$	**if** B **then** P_1 **else** P_2	branching
	$\mid$	**while** B **do** P	loop
	$\mid$	$V :=$ **cons**$(E_1, \ldots, E_n)$	allocation
	$\mid$	**free**(E)	deallocation
	$\mid$	$V := [E]$	dereferencing
	$\mid$	$[E] := E$	heap assignment

Remarks:

- reading, writing, and disposing pointers can fail if not allocated properly
- allocation never fails

Heap Memory Model

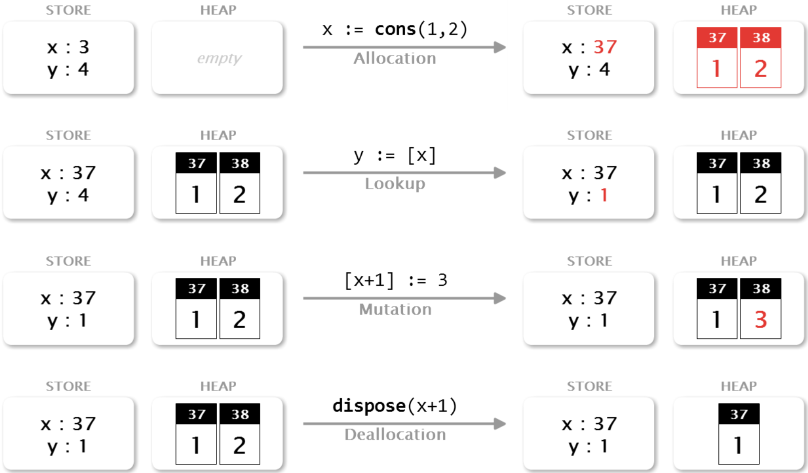
program state

- *stack*:
  mapping $s\colon \mathrm{Var} \to \mathbb{Z}$ of variables from set of variables Var to values (integers)
- *heap*:
  mapping $h\colon \mathrm{Addr} \to \mathbb{Z}$ of addresses (finite subset of $\mathbb{N}$) to values
  $\Rightarrow$ arithmetic with addresses is possible

set of all states

$S = \{(s, h) \mid s \text{ is stack}, h \text{ is heap}\}$

Program Semantics by Example



STORE	HEAP		STORE	HEAP
x : 3 y : 4	*empty*	x := **cons**(1,2) Allocation	x : 37 y : 4	37 38 1 2
x : 37 y : 4	37 38 1 2	y := [x] Lookup	x : 37 y : 1	37 38 1 2
x : 37 y : 1	37 38 1 2	[x+1] := 3 Mutation	x : 37 y : 1	37 38 1 3
x : 37 y : 1	37 38 1 2	**dispose**(x+1) Deallocation	x : 37 y : 1	37 1

Examples by Cristina Serban, `http://www-verimag.imag.fr/~serban/`

Problem with the Hoare Calculus

In the classical Hoare Calculus the following rules holds

$$\frac{\{E_1\}P\{E_2\}}{\{E_1 \land E\}P\{E_2 \land E\}}$$

if no free variable occurring in E is modified by P.

This rule does not with pointers:

$$\frac{\{X \mapsto 0\}[X] := 2\{X \mapsto 2\}}{\{X \mapsto 0 \land Y \mapsto 0\}[X] := 2\{X \mapsto 2 \land Y \mapsto 0\}}$$

Semantics of the Extended WHILE Language

Now $\rightsquigarrow$: Configs $\times$ Configs is defined as follows:

$$(V := \textbf{cons}(E_1, \ldots, E_n), (s, h)) \quad \rightsquigarrow \quad (s', h') \text{ with } \begin{array}{rcr} s'(V) = & \alpha_1 & \alpha_i \in \text{Addr}[1] \\ s'(X) = & s(X) & X \neq V \\ h'(\alpha_i) = & [\![E_i]\!]s & \\ h'(\beta) = & h(\beta) & \beta \neq \alpha_i \end{array}$$

$$(\textbf{free}(E), (s, h)) \quad \rightsquigarrow \quad (s, h|dom(h)\backslash\{[\![E]\!]s\}) \text{ if}$$

$$(V := [E], (s, h)) \quad \rightsquigarrow \quad (s', h) \text{ with } \begin{array}{rr} s'(V) = & h([\![E]\!]s) \\ s'(X) = & s(X) \text{ for } X \neq V \end{array}$$

$$([E] := E', (s, h)) \quad \rightsquigarrow \quad (s, h') \text{ with } \begin{array}{rcr} h'([\![E]\!]s) = & [\![E']\!]s & \\ h'(\alpha) = & h(\alpha) & \beta \neq [\![E]\!]s \end{array}$$

The last three rules are only applicable if $[\![E]\!]s \in dom(h)$.

---

[1] α_i are n new addresses

Heap Assertions

- **emp**
  The heap is empty.
- $E \mapsto E'$
  The cell in the heap with address E contains content E'.
- $P_1 * P_2$ (separating conjunction)
  The heap consists of two disjoint parts such that in one part P_1 holds and in the other part P_2 holds.

Examples

- $X \mapsto 1 * X \mapsto 1$ is unsatisfiable.
- $X \mapsto 1 * Y \mapsto 1$ is unsatisfiable in a state in which X and Y refer to the same location.
- $X \mapsto E_1 \land Y \mapsto E_2$ asserts that $E_1 = E_2$.

Example of Sharing Patterns

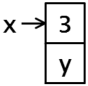
$x \mapsto 3, y$

STORE

x : a
y : b

HEAP

a	a+1
3	b



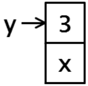
$y \mapsto 3, x$

STORE

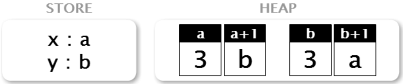
x : a
y : b

HEAP

b	b+1
3	a



$x \mapsto 3, y \ast y \mapsto 3, x$

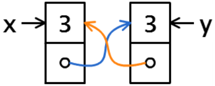


STORE

x : a
y : b

HEAP

a	a+1	b	b+1
3	b	3	a



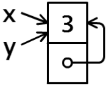
$x \mapsto 3, y \wedge y \mapsto 3, x$

STORE

x : a
y : a

HEAP

a	a+1
3	a



Examples by Cristina Serban, `http://www-verimag.imag.fr/~serban/`

Semantics of Heap Assertions

Let s be a stack, h, be a heap and P an assertion. We define that P is true in (s,h) (written as $s,h \vdash P$) if the following holds:

$$
\begin{array}{lll}
s,h \vdash \textbf{emp} & \text{iff} & dom(h) = \{\} \\
s,h \vdash E \mapsto E' & \text{iff} & dom(h) = \{[\![E]\!]s\} \\
 & & h([\![E]\!]s) = [\![E']\!]s \\
s,h \vdash P_1 * P_2 & \text{iff} & \exists h_1, h_2 : dom(h_1) \cap dom(h_2) = \emptyset, h_1 \cup h_2 = h, \\
 & & s,h_1 \vdash P_1, \\
 & & s,h_2 \vdash P_2 \\
s,h \vdash B & \text{iff} & [\![B]\!]s \text{ where } B \text{ is a pure formula}
\end{array}
$$

Hoare Triples for Separation Logic

A hoare triple $\{F\}P\{G\}$ holds iff for all configurations (s, h) with $s, h \vdash F$

1. $(P, (s, h)) \not\rightsquigarrow^* $ **error**
2. $\forall(s', h')$ with $(P, (s, h)) \rightsquigarrow^* (s', h') : (s', h') \vdash G$

Example

triple	*holds*
$\{V \mapsto -\}[V] := 0\{V \mapsto 0\}$	✓
$\{V \mapsto -\}[V] := 1\{V \mapsto 0\}$	✗
$\{\}[V] := 0\{V \mapsto 0\}$	✗

where $E \mapsto -$ means $\exists E' : E \mapsto E'$

Inference Rules: Axioms

$$
\begin{array}{cccc}
\{E \mapsto -\} & \textbf{free}(E) & \{\textbf{emp}\} & \\
\{(E \mapsto -) * R\} & \textbf{free}(E) & \{R\} & \\
\{\textbf{emp}\} & V := \textbf{cons}(E) & \{V \mapsto E\} & \\
\{R\} & V := \textbf{cons}(E) & \{V \mapsto E * R\} & \\
\{E \mapsto -\} & [E] := E' & \{E \mapsto E'\} & \\
\{(E \mapsto -) * R\} & [E] := E' & \{E \mapsto E' * R\} & \\
\{(E \mapsto E') * R\} & X := [E] & \{X = E' \wedge E \mapsto E' * R\} & X \notin E, E', R \\
\{(E \mapsto E') \wedge X = x\} & X := [E] & \{X = E' \wedge E[x/X] \mapsto E'\} &
\end{array}
$$

where $E \mapsto -$ means $\exists E' : E \mapsto E'$

Frame Rule

$$\frac{\{E_1\}P\{E_2\}}{\{E_1 * E\}P\{E_2 * E\}}$$

where for all free variables X of E: $X \notin mod(P)$ and $mod(P)$ is defined as follows

$$\begin{array}{lcl}
mod(\textbf{skip}) & = & \emptyset \\
mod(V := E) & = & \{V\} \\
mod(P_1; P_2) & = & mod(P_1) \cup mod(P_2) \\
mod(\textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2) & = & mod(P_1) \cup mod(P_2) \\
mod(\textbf{while } B \textbf{ do } P) & = & mod(P)
\end{array}$$

Outlook: Inductive Data Structures & Recursion

definition of $tree(x)$ with root pointer x:

$x = nil$: **emp** $\Rightarrow$ tree(x)

$x \neq nil$: $x \mapsto (y, z) * tree(y) * tree(z)$ $\Rightarrow$ tree(x)

```
deltree($*x$) {
  if $x == nil$ then return;
  else {
    $l, r := x.left, x.right$;
    deltree($l$);
    deltree($r$);
    free($x$);
  }
}
```

Example by James Brotherston, `http://www0.cs.ucl.ac.uk/staff/J.Brotherston/`

Outlook: Deletion of a Tree

```
$\{tree(x)\}$
deltree($*x$) {
    if $x == nil$ then return;
$\{eheap\}$
    else {
$\{x \mapsto (y, z) * tree(y) * tree(z)\}\}$
        $l, r := x.$left, $x.$right;
$\{x \mapsto (y, z) * tree(l) * tree(r)\}$
        deltree($l$);
$\{x \mapsto (y, z) * \textbf{emp} * tree(r)\}$
        deltree($r$);
$\{x \mapsto (y, z) * \textbf{emp} * \textbf{emp}\}$
        free($x$);
$\{\textbf{emp} * \textbf{emp} * \textbf{emp}\}$
    }
$\{\textbf{emp}\}$
}
$\{\textbf{emp}\}$
```