Hoare Logic

KC Sivaramakrishnan

Spring 2020

IIT MADRAS

Context

- Previously: Lambda Calculus
  - This lecture: Prove deeper properties about programs correctness.
  - Not just absence of crashes (type soundness)
- Go back to an imperative language with valuation and heap (aliasing, pointers)
- Describe the semantics of the program using operational semantics
  - But directly proving properties on operational semantics becomes tedious
- Hoare Logic: Machinery for proving program correctness *automatically*
  - Invented by C.A.R.Hoare (the same person who invented quick sort).

Syntax

$$
\begin{array}{rccl}
\text{Numbers} & n & \in & \mathbb{N} \\
\text{Variables} & x & \in & \textsf{Strings} \\
\text{Expressions} & e & ::= & n \mid x \mid e + e \mid e - e \mid e \times e \mid *[e] \\
\text{Boolean expressions} & b & ::= & e = e \mid e < e \\
\text{Commands} & c & ::= & \textsf{skip} \mid x \leftarrow e \mid *[e] \leftarrow e \mid c; c \\
 & & & \mid \textsf{if } b \textsf{ then } c \textsf{ else } c \mid \{a\}\textsf{while } b \textsf{ do } c \mid \textsf{assert}(a)
\end{array}
$$

$$
\begin{array}{rccl}
\text{Heap} & h & \coloneqq & nat \to nat \\
\text{Valuation} & v & \coloneqq & var \to nat \\
\text{Assertion} & a & \coloneqq & heap \to valuation \to Prop
\end{array}
$$

Semantics of Expressions

$$
\begin{array}{rcl}
[\![n]\!](h,v) & = & n \\
[\![x]\!](h,v) & = & v(x) \\
[\![e_1 + e_2]\!](h,v) & = & [\![e_1]\!](h,v) + [\![e_2]\!](h,v) \\
[\![e_1 - e_2]\!](h,v) & = & [\![e_1]\!](h,v) - [\![e_2]\!](h,v) \\
[\![e_1 \times e_2]\!](h,v) & = & [\![e_1]\!](h,v) \times [\![e_2]\!](h,v) \\
[\![*[e]]\!](h,v) & = & h([\![e]\!](h,v)) \\
[\![e_1 = e_2]\!](h,v) & = & [\![e_1]\!](h,v) = [\![e_2]\!](h,v) \\
[\![e_1 < e_2]\!](h,v) & = & [\![e_1]\!](h,v) < [\![e_2]\!](h,v)
\end{array}
$$

Semantics of Commands

$$\frac{}{(h, v, \mathsf{skip}) \Downarrow (h, v)} \quad \frac{}{(h, v, x \leftarrow e) \Downarrow (h, v[x \mapsto [\![e]\!](h, v)])}$$

$$\frac{}{(h, v, *[e_1] \leftarrow e_2) \Downarrow (h[[\![e_1]\!](h, v) \mapsto [\![e_2]\!](h, v)], v)}$$

$$\frac{(h, v, c_1) \Downarrow (h_1, v_1) \quad (h_1, v_1, c_2) \Downarrow (h_2, v_2)}{(h, v, c_1; c_2) \Downarrow (h_2, v_2)}$$

$$\frac{[\![b]\!](h, v) \quad (h, v, c_1) \Downarrow (h', v')}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow (h', v')} \quad \frac{\neg[\![b]\!](h, v) \quad (h, v, c_2) \Downarrow (h', v')}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \Downarrow (h', v')}$$

$$\frac{[\![b]\!](h, v) \quad (h, v, c; \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h', v')}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h', v')} \quad \frac{\neg[\![b]\!](h, v)}{(h, v, \mathsf{while}\ b\ \mathsf{do}\ c) \Downarrow (h, v)}$$

$$\frac{a(h, v)}{(h, v, \mathsf{assert}(a)) \Downarrow (h, v)}$$

Hoare Triple

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

Pre-condition Post-condition

$$P, Q := heap \to valuation \to Prop$$

- Capture the *effect* of each command on the valuation and the heap.
  - ✦ Pre- and post-conditions are an abstraction of the program behaviour
- Use this to build up to the *effect* of the entire program

Hoare Triple

$$\frac{}{\{P\}\mathsf{skip}\{P\}} \qquad \frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}} \qquad \frac{\forall s.\ P(s) \Rightarrow I(s)}{\{P\}\mathsf{assert}(I)\{P\}}$$

$$\frac{}{\{P\}x \leftarrow e\{\lambda(h,v).\ \exists v'.\ P(h,v') \wedge v = v'[x \mapsto [\![e]\!](h,v')]\}}$$

$$\frac{}{\{P\}{*}[e_1] \leftarrow e_2\{\lambda(h,v).\ \exists h'.\ P(h',v) \wedge h = h'[[\![e_1]\!](h',v) \mapsto [\![e_2]\!](h',v)]\}}$$

$$\frac{\{\lambda s.\ P(s) \wedge [\![b]\!](s)\}c_1\{Q_1\} \quad \{\lambda s.\ P(s) \wedge \neg[\![b]\!](s)\}c_2\{Q_2\}}{\{P\}\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\{\lambda s.\ Q_1(s) \vee Q_2(s)\}}$$

[Consequence] $$\frac{\{P\}c\{Q\} \quad (\forall s.\ P'(s) \Rightarrow P(s)) \quad (\forall s.\ Q(s) \Rightarrow Q'(s))}{\{P'\}c\{Q'\}}$$

Hoare Triple

$$\frac{(\forall s.\ P(s) \Rightarrow I(s)) \quad \{\lambda s.\ I(s) \wedge [\![b]\!](s)\}c\{I\}}{\{P\}\{I\}\text{while } b \text{ do } c\{\lambda s.\ I(s) \wedge \neg[\![b]\!](s)\}}$$

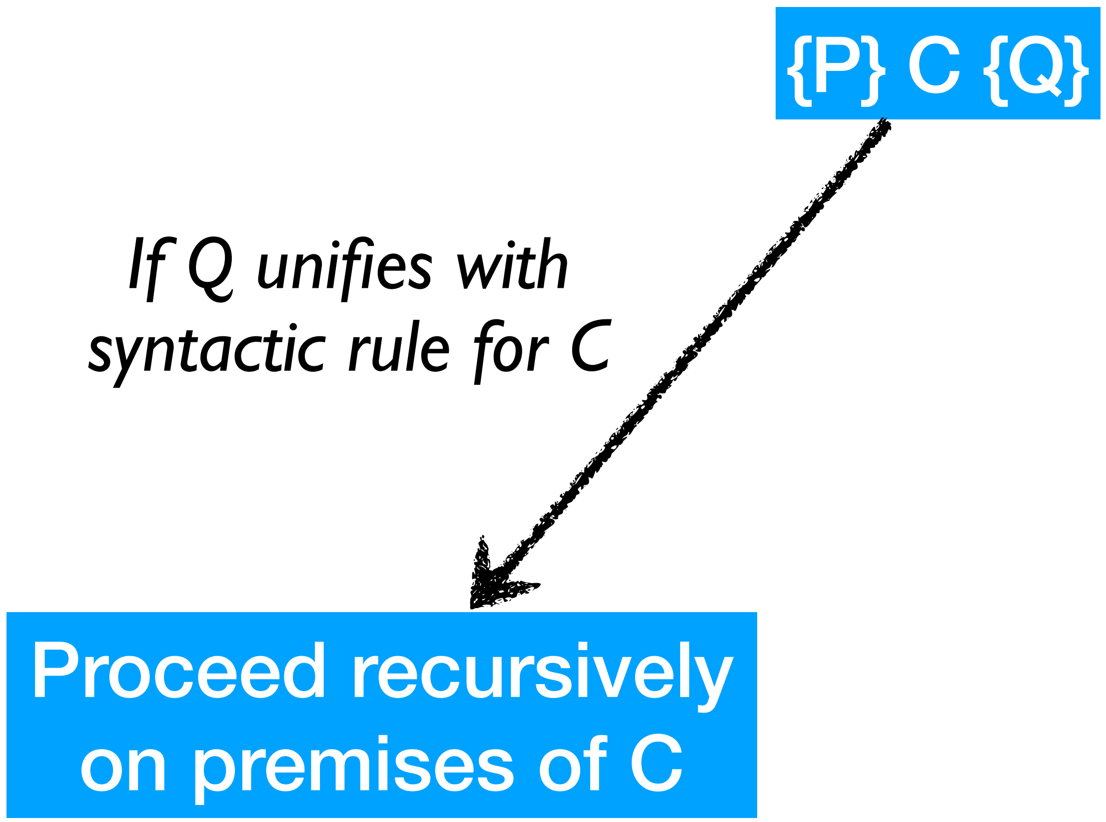
loop invariant

- Loop invariant holds true at the beginning, during and at the end of the loop
  - ✦ Closely connected to invariants in transition systems
- Loop invariants give the induction hypothesis that makes the correctness proof go through
  - ✦ Is not syntax directed
- Inferring good loop (inductive) invariants is active research

Towards automated verification

{P} C {Q}

Towards automated verification



Towards automated verification

{P} C {Q}

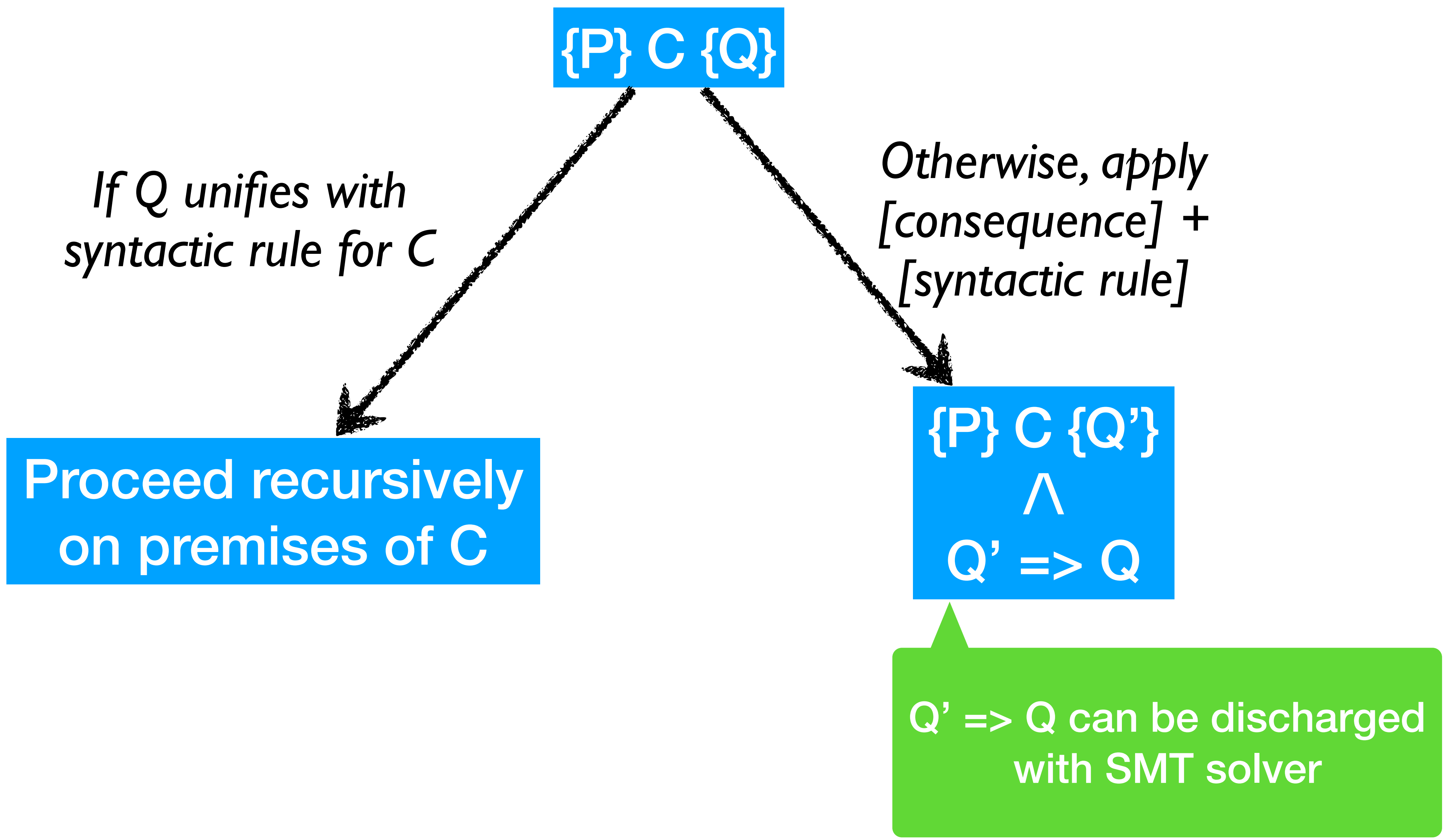
If Q unifies with syntactic rule for C

Proceed recursively on premises of C

Otherwise, apply [consequence] + [syntactic rule]

{P} C {Q'}
∧
Q' => Q

Towards automated verification



Soundness

- Connect Hoare Triple with Operational Semantics
  - ✦ Similar to types and operational semantics
  - ✦ What is the analogy of "well-typed programs do not crash" here?

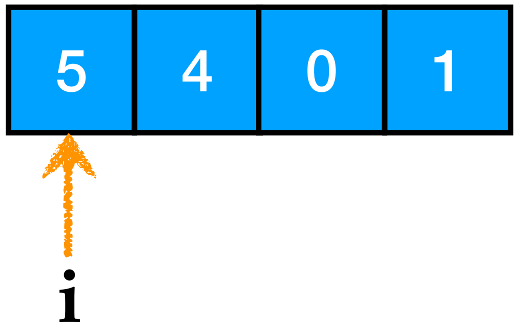
THEOREM 12.2 (Soundness of Hoare logic). *If* $\{P\}c\{Q\}$, $(h, v, c) \Downarrow (h', v')$, *and* $P(h, v)$, *then* $Q(h', v')$.

Selection Sort

```
selection_sort (a,n) =
  i <- 0;
  while i < n loop
    j <- i + 1;
    best <- i;
    while j < n loop
      when *(a + j) < *(a + best) then
        best <- j
      else skip;
      j <- j + 1
    done
    tmp <- *(a + best);
    *(a + best) <- *(a + i);
    *(a + i) <- tmp;
    i <- i + 1
  done
```

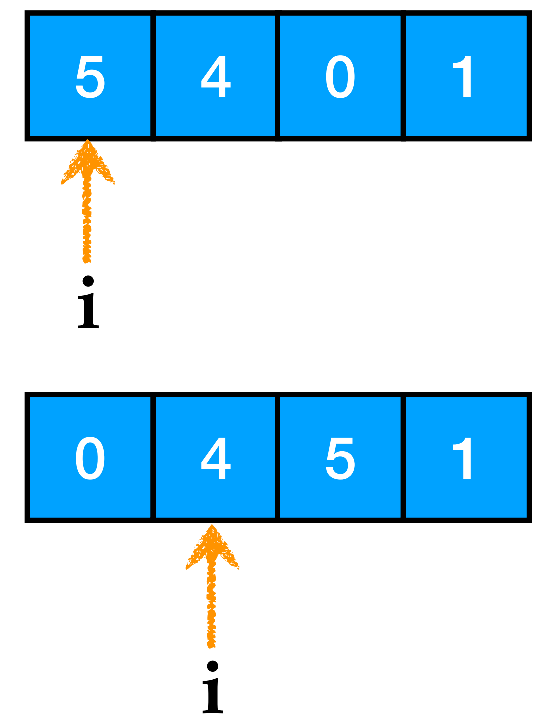
Selection Sort

```
selection_sort (a,n) =
  i <- 0;
  while i < n loop
    j <- i + 1;
    best <- i;
    while j < n loop
      when *(a + j) < *(a + best) then
        best <- j
      else skip;
      j <- j + 1
    done
    tmp <- *(a + best);
    *(a + best) <- *(a + i);
    *(a + i) <- tmp;
    i <- i + 1
  done
```



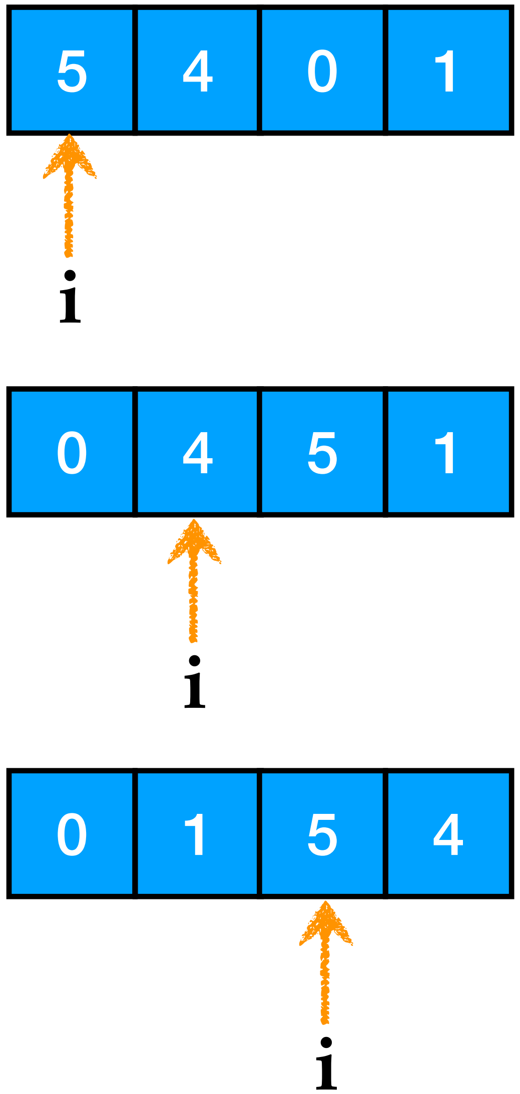
Selection Sort

```
selection_sort (a,n) =
  i <- 0;
  while i < n loop
    j <- i + 1;
    best <- i;
    while j < n loop
      when *(a + j) < *(a + best) then
        best <- j
      else skip;
      j <- j + 1
    done
    tmp <- *(a + best);
    *(a + best) <- *(a + i);
    *(a + i) <- tmp;
    i <- i + 1
  done
```



Selection Sort

```
selection_sort (a,n) =
  i <- 0;
  while i < n loop
    j <- i + 1;
    best <- i;
    while j < n loop
      when *(a + j) < *(a + best) then
        best <- j
      else skip;
      j <- j + 1
    done
    tmp <- *(a + best);
    *(a + best) <- *(a + i);
    *(a + i) <- tmp;
    i <- i + 1
  done
```



Hoare Logic + Small-step

- As we know, big step operational semantics can only deal with terminating programs
- Hoare Logic naturally applies to small step semantics as well

Small-step Operational Semantics

$$\frac{}{(h, v, x \leftarrow e) \rightarrow (h, v[x \mapsto [\![e]\!](h, v)], \mathsf{skip})}$$

$$\frac{}{(h, v, *[e_1] \leftarrow e_2) \rightarrow (h[[\![e_1]\!](h, v) \mapsto [\![e_2]\!](h, v)], v, \mathsf{skip})}$$

$$\frac{}{(h, v, \mathsf{skip}; c_2) \rightarrow (h, v, c_2)} \qquad \frac{(h, v, c_1) \rightarrow (h', v', c_1')}{(h, v, c_1; c_2) \rightarrow (h', v', c_1'; c_2)}$$

$$\frac{[\![b]\!](h, v)}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow (h, v, c_1)} \qquad \frac{\neg [\![b]\!](h, v)}{(h, v, \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \rightarrow (h, v, c_2)}$$

$$\frac{[\![b]\!](h, v)}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \rightarrow (h, v, c; \{I\}\mathsf{while}\ b\ \mathsf{do}\ c)} \qquad \frac{\neg [\![b]\!](h, v)}{(h, v, \{I\}\mathsf{while}\ b\ \mathsf{do}\ c) \rightarrow (h, v, \mathsf{skip})}$$

$$\frac{a(h, v)}{(h, v, \mathsf{assert}(a)) \rightarrow (h, v, \mathsf{skip})}$$

Invariant Safety

- Small-step semantics is said to be *stuck* when the command is not Skip, but no way to take a step.
  - In lambda calculus, 0 + (\x.x). What is an example of stuck expression in our language?

$$\frac{a(h, v)}{(h, v, \mathsf{assert}(a)) \to (h, v, \mathsf{skip})}$$

THEOREM 12.6 (Invariant Safety). *If $\{P\}c\{Q\}$ and $P(h, v)$, then unstuckness is an invariant for the small-step transition system starting at (h, v, c).*

LEMMA 12.3 (Progress). *If $\{P\}c\{Q\}$ and $P(h, v)$, then (h, v, c) is unstuck.*

LEMMA 12.5 (Preservation). *If $\{P\}c\{Q\}$, $(h, v, c) \to (h', v', c')$, and $P(h, v)$, then $\{\lambda s.\ s = (h', v')\}c'\{Q\}$.*