The Rule of Constancy (Derived Frame Rule)

- The following derived rule is used on the next slide

\[
\begin{align*}
\Gamma &\vdash \{ P \} \ C \ \{ Q \} \\
\Gamma &\vdash \{ P \land R \} \ C \ \{ Q \land R \}
\end{align*}
\]

where no variable assigned to in \( C \) occurs in \( R \)

- Outline of derivation
  - prove \( \{ R \} \ C \ \{ R \} \) by induction on \( C \)
  - then use Specification Conjunction

- Assume \textbf{C doesn’t modify } \( V \) and \( \Gamma \vdash \{ P \} \ C \ \{ P[V+1/V] \} \) then:
  \[
  \begin{align*}
  \Gamma &\vdash \{ P \land V=v \} \ C \ \{ P[V+1/V] \land V=v \} \quad \text{(assumption + constancy rule)} \\
  \Gamma &\vdash \{ P[V+1/V] \land V=v \} \ V:=V+1 \ \{ P \land V=v+1 \} \quad \text{(assign. ax + pre. streng.)} \\
  \Gamma &\vdash \{ P \land V=v \} \ C; \ V:=V+1 \ \{ P \land V=v+1 \} \quad \text{(sequencing)}
  \end{align*}
  \]

- So \( C; \ V:=V+1 \) has \( P \) as an invariant and increments \( V \)
Towards the FOR-Rule

- If \( e_1 \leq e_2 \) the FOR-command is equivalent to:

\[
\text{BEGIN VAR } V; \ V := e_1; \ldots \ C; \ V := V + 1; \ldots \ V := e_2; \ C \ \text{END}
\]

- Assume \( C \) doesn’t modify \( V \) and \( \vdash \{ P \} \ C \{ P[V+1/V] \} \)

- Hence:

\[
\vdash \{ P[e_1/V] \} \ V := e_1 \ \{ P \land V = e_1 \} \quad \text{(assign. ax + pre. streng.)}
\]

\[
\vdash \{ P \land V = v \} \ C; \ V := V + 1 \ \{ P \land V = v + 1 \} \quad \text{(last slide; } V = e_1, e_1 + 1, \ldots, e_2 - 1 \text{)}
\]

\[
\vdash \{ P \land V = v \} \ C; \ V := V + 1 \ \{ P \land V = e_2 + 1 \}
\]

\[
\vdash \{ P \land V = e_2 \} \ C \ \{ P[V+1/V] \land V = e_2 \} \quad \text{(assign. ax + assumption + constancy)}
\]

\[
\vdash \{ P \land V = e_2 \} \ C \ \{ P[e_2 + 1/V] \} \quad \text{(post. weak.)}
\]

- Hence by the sequencing and block rules

\[
\vdash \{ P \} \ C \{ P[V+1/V] \}
\]

\[
\vdash \{ P[e_1/V] \} \text{BEGIN VAR } V; V := e_1; \ldots \ C; \ V := V + 1; \ldots \ V := e_2; \ C \ \text{END} \{ P[e_2 + 1/V] \}
\]
• To rule out the problems that arise when the controlled variable or variables in the bounds expressions, are changed by the body, we simply impose a side condition on the rule that stipulates that it cannot be used in these situations.

The FOR-rule

\[
\begin{align*}
\vdash &\{P \land (E_1 \leq V) \land (V \leq E_2)\} C \{P[V+1/V]\} \\
\vdash &\{P[E_1/V] \land (E_1 \leq E_2)\} \text{ FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C \{P[E_2+1/V]\}
\end{align*}
\]

where neither $V$, nor any variable occurring in $E_1$ or $E_2$, is assigned to in the command $C$.

• Note $(E_1 \leq V) \land (V \leq E_2)$ in precondition of rule hypothesis
  - added to strengthen rule to allow proofs to use facts about $V$’s range of values

• Can be tricky to think up $P$
The FOR-rule does not enable anything to be deduced about FOR-
commands whose body assigns to variables in the bounds expres-
sions

This precludes such assignments being used if commands are to be reasoned about

Only defining rules of inference for non-tricky uses of constructs motivates writing programs in a perspicuous manner

It is possible to devise a rule that does cope with assignments to variables in bounds expressions

Consider the rule below (\(e_1, e_2\) are fresh auxiliary variables):

\[
\vdash \{ P \wedge (e_1 \leq V) \wedge (V \leq e_2) \} \quad C \quad \{ P[V+1/V] \} \\
\vdash \{ P[E_1/V] \wedge (E_1 \leq E_2) \wedge (E_1 = e_1) \wedge (E_2 = e_2) \} \quad \text{FOR} \quad V := E_1 \quad \text{UNTIL} \quad E_2 \quad \text{DO} \quad C \quad \{ P[e_2+1/V] \}
\]
To cover the case when $E_2 < E_1$, we need the **FOR-axiom** below:

\[
\vdash \{P \land (E_2 < E_1)\} \text{ FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C \{P\}
\]

This says that when $E_2$ is less than $E_1$ the **FOR**-command has no effect.
Ensuring Soundness

- It is clear from the discussion of the FOR-rule that it is not always straightforward to devise correct rules of inference.

- It is important that the axioms and rules be sound. There are two approaches to ensure this:

  (i) define the language by the axioms and rules of the logic
  (ii) prove that the logic is sound for the language

- **Approach (i) is called axiomatic semantics**
  - the idea is to define the semantics of the language by requiring that it make the axioms and rules of inference true
  - it is then up to implementers to ensure that the logic matches the language

- **Approach (ii) is proving soundness of the logic**
• One snag with axiomatic semantics is that most existing languages have already been defined in some other way
  • usually by informal and ambiguous natural language statements

• The other snag with axiomatic semantics is that by Clarke’s Theorem it is known to be impossible to devise relatively complete Floyd-Hoare logics for languages with certain constructs
  • it could be argued that this is not a snag at all but an advantage, because it forces programming languages to be made logically tractable

• An example of a language defined axiomatically is Euclid
7.1. (module rule)

(1) \( Q \Rightarrow Q_0(A/t), \)

(2) \( P_1\{\text{const} \ K; \ \text{var} \ V; \ S_4\} \ Q_4(A/t) \land Q. \)

(3) \( P_2(A/t) \land Q \{S_2\} \ Q_2(A/t) \land Q. \)

(4) \( \exists \ g_1(P_3(A/t) \land Q \{S_3\} \ Q_3(A/t) \land g = g_1(A, c, d)). \)

(5) \( \exists \ g(P_3(A/t) \land Q \Rightarrow Q_3(A/t)). \)

(6) \( P_6(A/t) \land Q \{S_6\} \ Q_1. \)

(7) \( P \Rightarrow P_1(a/c). \)

(8.1) \( [Q_0(a/c, x/t, x'/t') \Rightarrow (P_2(x/t, x'/t', a2/x2, e2/c2, a/c) \land \)

\( (Q_2(x2#/t, x'/t', a2#/x2, e2/c2, a/c, y2#/y2, a2/x2', y2/y2') \Rightarrow \)

\( R_1(x2#/x, a2#/a2, y2#/y2)) \{x. p(a2, e2) \} R_1 \land Q_0(a/c, x/t, x'/t'). \)

(8.2) \( (Q_0(a/c, x/t) \Rightarrow P_3(x/t, a3/c3, a/c)) \Rightarrow \)

\( Q_3(x/t, a3/c3, a/c, f(a3, d3)/g) \land Q_0(a/c, x/t). \)

(8.3) \( P_1(a/c) \land (Q_4(x4#/t, x'/t', a/c, y4#/y4, y4/y4') \Rightarrow R_4(x4#/x, y4#/y4)) \)

\( \{x. \text{Initially}\} R_4 \land Q_0(a/c, x/t, x'/t'). \)

(8.4) \( (Q_0(a/c, x/t, x'/t') \Rightarrow P_6(x/t, x'/t', a/c)) \land (Q_1(a/c, y6#/y6, y6/y6') \Rightarrow \)

\( R(y6#/y6)) \{x. \text{Finally}\} R]. \)

(8.5) \( P(x#/x) \{x. \text{Initially}; \ S; x. \text{Finally}\} R(x#/x) \)

\( \overline{P\{\text{var} \ x: T(a); \ S\} \ R \land Q_1} \)
• Syntax: $V(E_1):=E_2$

• Semantics: the state is changed by assigning the value of the term $E_2$ to the $E_1$-th component of the array variable $V$

• Example: $A(X+1) := A(X)+2$
  
  • if the the value of $X$ is $x$
  
  • and the value of the $x$-th component of $A$ is $n$
  
  • then the value stored in the $(x+1)$-th component of $A$ becomes $n+2$
The axiom

\( \vdash \{P[E_2/A(E_1)]\} \ A(E_1):=E_2 \ \{P\} \)

doesn’t work

• Take \( P \equiv 'X=Y \land A(Y)=0', \ E_1 \equiv 'X', \ E_2 \equiv '1'\)

  • since \( A(X) \) does not occur in \( P \)
  • it follows that \( P[1/A(X)] = P \)
  • hence the axiom yields: \( \vdash \{X=Y \land A(Y)=0\} \ A(X):=1 \ \{X=Y \land A(Y)=0\} \)

• Must take into account possibility that changes to \( A(X) \) may change \( A(Y), A(Z) \) etc

  • since \( X \) might equal \( Y, Z \) etc (i.e. \textbf{aliasing})

• Related to the \textit{Frame Problem} in AI
The naive array assignment axiom

\[ \vdash \{ P[E_2/A(E_1)] \} \ A(E_1):=E_2 \ {P} \]

does not work: changes to \( A(X) \) may also change \( A(Y), A(Z), \ldots \).

The solution, due to Hoare, is to treat an array assignment

\[ A(E_1):=E_2 \]

as an ordinary assignment

\[ A := A\{E_1←E_2\} \]

where the term \( A\{E_1←E_2\} \) denotes an array identical to \( A \), except that the \( E_1 \)-th component is changed to have the value \( E_2 \).
Array Assignment axiom

- Array assignment is a special case of ordinary assignment

\[ A := A \{ E_1 \leftarrow E_2 \} \]

- Array assignment axiom just ordinary assignment axiom

\[ \vdash \{ P[A\{E_1 \leftarrow E_2\}/A] \} \ A := A\{E_1 \leftarrow E_2\} \ \{P\} \]

- Thus:

\[
\vdash \{ P[A\{E_1 \leftarrow E_2\}/A] \} \ A(E_1) := E_2 \ \{P\}
\]

The array assignment axiom

Where \( A \) is an array variable, \( E_1 \) is an integer valued expression, \( P \) is any statement and the notation \( A\{E_1 \leftarrow E_2\} \) denotes the array identical to \( A \), except that \( A(E_1) = E_2 \).
Array Axioms

- In order to reason about arrays, the following axioms, which define the meaning of the notation $A\{E_1 \leftarrow E_2\}$, are needed

\[
\begin{align*}
\vdash A\{E_1 \leftarrow E_2\}(E_1) &= E_2 \\
\vdash E_1 \neq E_3 &\Rightarrow A\{E_1 \leftarrow E_2\}(E_3) = A(E_3)
\end{align*}
\]

- Second of these is a *Frame Axiom*
  - don’t confuse with Frame Rule of Separation Logic (later)
  - “frame” is a rather overloaded word!
New Topic: Separation logic

- One of several competing methods for reasoning about pointers
- Details took 30 years to evolve
- Shape predicates due to Rod Burstall in the 1970s
- Separation logic: by O’Hearn, Reynolds and Yang around 2000
- Several partially successful attempts before separation logic
- Very active research area
  - QMUL, UCL, Cambridge, Harvard, Princeton, Yale
  - Microsoft
  - startup Monoidics bought by Facebook
Pointers and the state

- So far the state just determined the values of variables
  - values assumed to be numbers
  - preconditions and postconditions are first-order logic statements
  - state same as a valuation \( s : \text{Var} \rightarrow \text{Val} \)

- To model pointers – e.g. as in C – add heap to state
  - heap maps locations (pointers) to their contents
  - store maps variables to values (previously called state)
  - contents of locations can be locations or values

\[
\begin{align*}
X & \mapsto l_1 & l_1 & \mapsto l_2 & l_2 & \mapsto v \\
\text{store} & & \text{heap} & & \text{heap}
\end{align*}
\]

Heap semantics

\[
\begin{align*}
\text{Store} &= \text{Var} \rightarrow \text{Val} \\
\text{Heap} &= \text{Num} \Rightarrow_{\text{fin}} \text{Val} \\
\text{State} &= \text{Store} \times \text{Heap}
\end{align*}
\]

(assume \( \text{Num} \subseteq \text{Val}, \text{nil} \in \text{Val} \) and \( \text{nil} \notin \text{Num} \))

- Note: store also called stack or environment; heap also called store
Adding pointer operations to our language

Expressions:
\[ E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots \]

Boolean expressions:
\[ B ::= T \mid F \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots \]

Commands:
\[ C ::= V := E \quad \text{value assignments} \]
\[ V := [E] \quad \text{fetch assignments} \]
\[ [E_1] := E_2 \quad \text{heap assignments (heap mutation)} \]
\[ V := \text{cons}(E_1, \ldots, E_n) \quad \text{allocation assignments} \]
\[ \text{dispose}(E) \quad \text{pointer disposal} \]
\[ C_1 ; C_2 \quad \text{sequences} \]
\[ \text{IF} B \text{ THEN } C_1 \text{ ELSE } C_2 \quad \text{conditionals} \]
\[ \text{WHILE} B \text{ DO } C \quad \text{while commands} \]
Pointer manipulation constructs and faulting

- Commands executed in a state \((s, h)\)
- Reading, writing or disposing pointers might \textit{fault}
- Fetch assignments: \(V := [E]\)
  - evaluate \(E\) to get a location \(l\)
  - fault if \(l\) is not in the heap
  - otherwise assign contents of \(l\) in heap to the variable \(V\)
- Heap assignments: \([E_1] := E_2\)
  - evaluate \(E_1\) to get a location \(l\)
  - fault if the \(l\) is not in the heap
  - otherwise store the value of \(E_2\) as the new contents of \(l\) in the heap
- Pointer disposal: \(\text{dispose}(E)\)
  - evaluate \(E\) to get a pointer \(l\) (a number)
  - fault if \(l\) is not in the heap
  - otherwise remove \(l\) from the heap
• **Allocation assignments:**  \( V := \text{cons}(E_1, \ldots, E_n) \)
  
  • choose \( n \) consecutive locations that are not in the heap, say \( l, l+1, \ldots \)
  
  • extend the heap by adding \( l, l+1, \ldots \) to it
  
  • assign \( l \) to the variable \( V \) in the store
  
  • make the values of \( E_1, E_2, \ldots \) be the new contents of \( l, l+1, \ldots \) in the heap

• **Allocation assignments never fault**

• **Allocation assignments are** non-deterministic
  
  • any suitable \( l, l+1, \ldots \) not in the heap can be chosen
  
  • always exists because the heap is finite
Example (different from the background reading)

\(X := \text{cons}(0, 1, 2); [X] := Y + 1; [X + 1] := Z; Y := [Y + Z]\)

- \(X := \text{cons}(0, 1, 2)\) allocates three new pointers, say \(l, l+1, l+2\)
  - \(l\) initialised with contents 0, \(l+1\) with 1 and \(l+2\) with 2
  - variable \(X\) is assigned \(l\) as its value in store

- \([X] := Y + 1\) changes the contents of \(l\)
  - \(l\) gets value of \(Y + 1\) as new contents in heap

- \([X + 1] := Z\) changes the contents of \(l+1\)
  - \(l+1\) gets the value of \(Z\) as new contents in heap

- \(Y := [Y + Z]\) changes the value of \(Y\) in the store
  - \(Y\) assigned in the store the contents of \(Y + Z\) in the heap
  - faults if the value of \(Y + Z\) is not in the heap
Local Reasoning and Separation Logic

- Want to just reason about just those locations being modified
  - assume all other locations unchanged

- Solution: separation logic
  - `small` and `forward` assignment axioms + `separating conjunction`
  - `small` means just applies to fragment of heap (footprint)
  - `forward` means Floyd-style forward rules that support symbolic execution
  - `non-faulting semantics` of Hoare triples
  - symbolic execution used by tools like smallfoot
  - `separating conjunction` solves frame problem - like rule of constancy for heap

- Need new kinds of assertions to state separation logic axioms
The frame rule

\[ \vdash \{P\} C \{Q\} \]

\[ \vdash \{P \star R\} C \{Q \star R\} \]

where no variable modified by \( C \) occurs free in \( R \).

- **Separating conjunction** \( P \star Q \)
  - heap can be split into two disjoint components
  - \( P \) is true of one component and \( Q \) of the other
  - \( \star \) is commutative and associative
Local Reasoning and Separation Logic

- Want to just reason about just those locations being modified
  - assume all other locations unchanged

- Solution: separation logic
  - **small** and **forward** assignment axioms + **separating conjunction**
  - **small** means just applies to fragment of heap (*footprint*)
  - **forward** means Floyd-style forward rules that support symbolic execution
  - **non-faulting semantics** of Hoare triples
  - symbolic execution used by tools like smallfoot
  - **separating conjunction** solves frame problem - like rule of constancy for heap

- Need new kinds of assertions to state separation logic axioms
Separation logic assertions: \textit{emp}

- \textit{emp} is an atomic statement of separation logic
- \textit{emp} is true iff the heap is empty
- The semantics of \textit{emp} is:
  \[ \text{emp} (s, h) \iff h = \{\} \quad (\text{where } \{\} \text{ is the empty heap}) \]
- Abbreviation: \[ E_1 \defeq E_2 = def (E_1 = E_2) \land \text{emp} \]
- From the semantics: \[ (E_1 \defeq E_2) (s, h) \iff E_1(s) = E_2(s) \land h = \{\} \]
- \[ E_1 = E_2 \] is independent of the heap and only depends on the store
- Semantics of \[ E_1 = E_2 \] is:
  \[ (E_1 = E_2)(s, h) \iff E_1(s) = E_2(s) \]
  no constraint on the heap – any \( h \) will do
Separation logic: small axioms and faulting

- One might expect a heap assignment axiom to entail:
  \[ \vdash \{ T \} [0] := 0 \{ 0 \mapsto 0 \} \]
  i.e. after executing \([0] := 0\) the contents of location 0 in the heap is 0

- Recall the sneak preview of the frame rule:

  \[
  \begin{array}{c}
  \vdash \{ P \} C \{ Q \} \\
  \vdash \{ P \star R \} C \{ Q \star R \}
  \end{array}
  \]
  where no variable modified by \(C\) occurs free in \(R\).

- Taking \(R\) to be the points-to statement \(0 \mapsto 1\) yields:
  \[ \vdash \{ T \star 0 \mapsto 1 \} [0] := 0 \{ 0 \mapsto 0 \star 0 \mapsto 1 \} \]
  something is wrong with the conclusion!

- Solution: define Hoare triple so \(\vdash \{ T \} [0] := 0 \{ 0 \mapsto 0 \}\) is not true
Non-faulting interpretation of Hoare triples

- The *non-faulting semantics* of Hoare triples \( \{P\} C \{Q\} \) is:

  if \( P \) holds then
  
  (i) executing \( C \) doesn’t fault **and**
  
  (ii) if \( C \) terminates then \( Q \) holds

\[
\models \{P\} C \{Q\} = \\
\forall s \ h. \ P(s, h) \Rightarrow -(C(s, h)\text{fault}) \land \forall s' \ h'. \ C(s, h)(s', h') \Rightarrow Q(s', h')
\]

- Now \( \vdash \{T\}[0]:=0\{0\rightarrow 0\} \) is not true as \([0]:=0)(s, \{\})\text{fault}

- Recall the sneak preview of the frame rule:

  The frame rule

  \[
  \vdash \{P\} \vdash \{P \star R\} \vdash \{P \star R\} \quad \text{where no variable modified by} \ C \ \text{occurs free in} \ R.
  \]

- So can’t use frame rule to get \( \vdash \{T \star 0\rightarrow 1\}[0]:=0\{0\rightarrow 0 \star 0\rightarrow 1\} \)
Store assignment axiom

\[ \vdash \{ V \leftarrow v \} V := E \{ V \leftarrow E[v/V] \} \]

where \( v \) is an auxiliary variable not occurring in \( E \).

- \( E_1 \leftarrow E_2 \) means value of \( E_1 \) and \( E_2 \) equal in the store and heap is empty.
- In Hoare logic (no heap) this is equivalent to the assignment axiom

\[
\begin{align*}
\vdash \{ V=v \} V := E \{ V=E[v/V] \} & \quad \text{store assign. ax.} \\
\vdash \{ V=v \land Q\{E[v/V]/V\} \} V := E \{ V=E[v/V] \land Q\{E[v/V]/V\} \} & \quad \text{rule of constancy} \\
\vdash \{ \exists v. V=v \land Q\{E[v/V]/V\} \} V := E \{ \exists v. V=E[v/V] \land Q\{E[v/V]/V\} \} & \quad \text{exists introduction} \\
\vdash \{ \exists v. V=v \land Q\{E[V/V]/V\} \} V := E \{ \exists v. V=E[v/V] \land Q\{V/V\} \} & \quad \text{predicate logic} \\
\vdash \{ \exists v. V=v \land Q\{E/V\} \} V := E \{ \exists v. V=E[v/V] \land Q \} & \quad [V/V] \text{ is identity} \\
\vdash \{ (\exists v. V=v) \land Q\{E/V\} \} V := E \{ (\exists v. V=E[v/V]) \land Q \} & \quad \text{predicate logic: } v \text{ not in } E \\
\vdash \{ T \land Q\{E/V\} \} V := E \{ (\exists v. V=E[v/V]) \land Q \} & \quad \text{predicate logic} \\
\vdash \{ Q\{E/V\} \} V := E \{ Q \} & \quad \text{rules of consequence}
\end{align*}
\]

- Separation logic: exists introduction valid, rule of constancy invalid
Fetch assignment axiom

\[
\vdash \{(V = v_1) \land E \mapsto v_2\} \quad V := [E] \quad \{(V = v_2) \land E[v_1/V] \mapsto v_2\}
\]

where \(v_1, v_2\) are auxiliary variables not occurring in \(E\).

- Precondition guarantees the assignment doesn’t fault
- \(V\) is assigned the contents of \(E\) in the heap
- Small axiom: precondition and postcondition specify singleton heap
- If neither \(V\) nor \(v\) occur in \(E\) then the following holds:
  \[
  \vdash \{E \mapsto v\} \quad V := [E] \quad \{(V = v) \land E \mapsto v\}
  \]
  (proof: instantiate \(v_1\) to \(V\) and \(v_2\) to \(v\) and then simplify)
Heap assignment axiom (heap mutation)

\[ \vdash \{ E \mapsto \_ \} [E] := F \{ E \mapsto F \} \]

- Precondition guarantees the assignment doesn’t fault
- Contents of $E$ in heap is updated to be value of $F$
- Small axiom: precondition and postcondition specify singleton heap
Pointer allocation

Allocation assignment axiom

\[ \vdash \{ V \equiv v \} V := \text{cons}(E_1, \ldots, E_n) \{ V \mapsto E_1[v/V], \ldots, E_n[v/V] \} \]

where \( v \) is an auxiliary variable not equal to \( V \) or occurring in \( E_1, \ldots, E_n \)

- **Never faults**
- **If** \( V \) **doesn’t occur in** \( E_1, \ldots, E_n \) **then:**
  
  \[ \vdash \{ V \equiv v \} V := \text{cons}(E_1, \ldots, E_n) \{ V \mapsto E_1[v/V], \ldots, E_n[v/V] \} \]
  
  \[ \vdash \{ V \equiv v \} V := \text{cons}(E_1, \ldots, E_n) \{ V \mapsto E_1, \ldots, E_n \} \]
  
  \[ \vdash \{ \exists v. \ V \equiv v \} V := \text{cons}(E_1, \ldots, E_n) \{ \exists v. \ V \mapsto E_1, \ldots, E_n \} \]
  
  \[ \vdash \{ \exists v. \ V = v \land \text{emp} \} V := \text{cons}(E_1, \ldots, E_n) \{ \exists v. \ V \mapsto E_1, \ldots, E_n \} \]
  
  \[ \vdash \{ \text{emp} \} V := \text{cons}(E_1, \ldots, E_n) \{ V \mapsto E_1, \ldots, E_n \} \]

- **Which is a derivation of:**

  Derived allocation assignment axiom

  \[ \vdash \{ \text{emp} \} V := \text{cons}(E_1, \ldots, E_n) \{ V \mapsto E_1, \ldots, E_n \} \]

  where \( V \) doesn’t occur in \( E_1, \ldots, E_n \).
Pointer deallocation

Dispose axiom

\[ \vdash \{ E \mapsto \_ \} \text{dispose}(E) \{ \text{emp} \} \]

- Attempting to deallocate a pointer not in the heap faults
- Small axiom: singleton precondition heap, empty postcondition heap
- Sanity checking example proof:
  \[ \vdash \{ E_1 \mapsto \_ \} \text{dispose}(E_1) \{ \text{emp} \} \]
  \[ \vdash \{ \text{emp} \} V := \text{cons}(E_2) \{ V \mapsto E_2 \} \]
  \[ \vdash \{ E_1 \mapsto \_ \} \text{dispose}(E_1); V := \text{cons}(E_2) \{ V \mapsto E_2 \} \]

  dispose axiom

  derived allocation assignment axiom

  sequencing rule
Compound command rules

- Following rules apply to both Hoare logic and separation logic

<table>
<thead>
<tr>
<th>The sequencing rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \ \vdash { P } C_1 { Q }, \quad \vdash { Q } C_2 { R } ]</td>
</tr>
<tr>
<td>[ \vdash { P } C_1 ; C_2 { R } ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The conditional rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \ \vdash { P \land S } C_1 { Q }, \quad \vdash { P \land \neg S } C_2 { Q } ]</td>
</tr>
<tr>
<td>[ \vdash { P } \text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2 { Q } ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The WHILE-rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \ \vdash { P \land S } C { P } ]</td>
</tr>
<tr>
<td>[ \vdash { P } \text{WHILE } S \text{ DO } C { P \land \neg S } ]</td>
</tr>
</tbody>
</table>

- For separation logic, need to think about faulting