Separation Logic

Slides modified from Reynolds' mini-course CS 818A3

Review of Hoare Logic

- Method of reasoning mathematically about *imperative* programs
  - Programming language
  - Assertion language
    - Predicate logic rules + soundness
  - Specification language
    - Hoare logic rules + soundness
- Basis of automated program verification systems

Hoare Logic Rules

$$\frac{}{\{p[e/x]\}\ x := e\ \{p\}}\ (\text{AS}) \qquad \frac{}{\{p\}\mathbf{skip}\{p\}}\ (\text{SK})$$

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1\ ;c_2\{q\}}\ (\text{SC}) \qquad \frac{\{p \wedge b\}c_1\{q\} \qquad \{p \wedge \neg b\}c_2\{q\}}{\{p\}\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{q\}}\ (\text{CD})$$

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i \wedge \neg b\}}\ (\text{WHP})$$

$$\frac{[i \wedge b \wedge (e = x_0)]\ c\ [i \wedge (e < x_0)] \qquad i \wedge b \Rightarrow e \geq 0}{[i]\ \mathbf{while}\ b\ \mathbf{do}\ c\ [i \wedge \neg b]}\ (\text{WHT})$$

$$\frac{p \Rightarrow q \qquad \{q\}c\{r\}}{\{p\}c\{r\}}\ (\text{SP}) \qquad \frac{\{p\}c\{q\} \qquad q \Rightarrow r}{\{p\}c\{r\}}\ (\text{WC})$$

This Class: Separation Logic

- Extension of Hoare logic for reasoning about pointers
- Details took 30 years to evolve
  - 1970s ~ around 2000
- Very active research area

Motivating example: list reversal

- $LREV \stackrel{\text{def}}{=} j := \mathbf{nil}; \mathbf{while}\ i \neq \mathbf{nil}\ \mathbf{do}$
  $(k := [i+1]; [i+1] := j; j := i; i := k)$

- Invariant: $\exists \alpha, \beta.\, \text{list}\ \alpha\ i \wedge \text{list}\ \beta\ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta$
  - $\text{list}\ \epsilon\ i \stackrel{\text{def}}{=} i = nil \qquad \text{list}\ (\text{a} \cdot \alpha)\ i \stackrel{\text{def}}{=} \exists j.\, i \hookrightarrow \text{a}, j \wedge \text{list}\ \alpha\ j$

- NOT enough: may malfunction if a node is shared between i and j

Motivating example: list reversal

- A stronger invariant I:
  $(\exists \alpha, \beta.\, \mathrm{list}\ \alpha\ i \wedge \mathrm{list}\ \beta\ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta)$
  $\wedge\ (\forall k.\, \mathbf{reachable}(i,k) \wedge \mathbf{reachable}(j,k) \Rightarrow k = \mathbf{nil})$

  - $\mathbf{reachable}(i,j) \stackrel{\text{def}}{=} \exists n \geq 0.\, \mathbf{reachable}_n(i,j)$
  - $\mathbf{reachable}_0(i,k) \stackrel{\text{def}}{=} i = j$
  - $\mathbf{reachable}_{n+1}(i,j) \stackrel{\text{def}}{=} \exists \mathrm{a}, k.\, i \hookrightarrow \mathrm{a}, k \wedge \mathbf{reachable}_n(k,j)$

- What if there is another list in the storage?
  - $I \wedge \mathrm{list}\ \gamma\ x \wedge\ (\forall k.\, \mathbf{reachable}(x,k) \wedge$
    $(\mathbf{reachable}(i,k) \vee \mathbf{reachable}(j,k)) \Rightarrow k = \mathbf{nil})$

Motivating example: list reversal

- A stronger invariant I:
  $(\exists \alpha, \beta.\, \mathrm{list}\ \alpha\ i \wedge \mathrm{list}\ \beta\ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta)$
  $\wedge\ (\forall k.\, \mathbf{reachable}(i, k) \wedge \mathbf{reachable}(j, k) \Rightarrow k = \mathbf{nil})$
- What if there is another list in the storage?
  - $I \wedge \mathrm{list}\ \gamma\ x \wedge\ (\forall k.\, \mathbf{reachable}(x, k) \wedge$
    $(\mathbf{reachable}(i, k) \vee \mathbf{reachable}(j, k)) \Rightarrow k = \mathbf{nil})$
- In separation logic:
  - $\exists \alpha, \beta.\, \mathrm{list}\ \alpha\ i * \mathrm{list}\ \beta\ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta$
  - $\exists \alpha, \beta.\, \mathrm{list}\ \alpha\ i * \mathrm{list}\ \beta\ j * \mathrm{list}\ \gamma\ x \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta$

Motivating example: list reversal

- Local reasoning:
  - Prove a ***local*** specification $\{\text{list } \alpha\ i\} LREV \{\text{list } \alpha^\dagger\ j\}$
  - It implies that this list is the ***only*** addressable storage touched by executing $LREV$ (often called the ***footprint*** of $LREV$)

  - Infer directly using the ***frame rule***:
    $\{\text{list } \alpha\ i * \text{list } \gamma\ x\} LREV \{\text{list } \alpha^\dagger\ j * \text{list } \gamma\ x\}$

Overview of Separation Logic

- Low-level programming language
  - Extension of simple imperative language
  - Commands for allocating, accessing, mutating, and deallocating data structures
  - Dangling pointer faults (if pointer is dereferenced)
- Program specification and proof
  - Extension of Hoare logic
  - Separating (independent, spatial) conjunction ($*$) and implication ($-\!\!*$)
- Inductive definitions over abstract structures

Early History

- Distinct Nonrepeating Tree Systems
  (Burstall 1972)
- Adding Separating Conjunction to Hoare Logic
  (Reynolds 1999, with flaws)
- Bunched Implication (BI) Logics
  (O'Hearn and Pym 1999)
- Intuitionistic Separation Logic
  (Ishtiaq and O'Hearn 2001, Reynolds 2000)
- Classical Separation Logic (Ishtiaq and O'Hearn 2001)
- Adding Address Arithmetic (Reynolds 2001)
- Concurrent Separation Logic (O'Hearn and Brookes 2004)

1. Programming Language

The Programming Language

$$
\begin{array}{lll}
(comm)\ c & ::= & \ldots \\
& | & x := \mathbf{cons}(e_1, \ldots, e_n) \\
& | & \mathbf{dispose}(e) \\
& | & x := [e] \\
& | & [e] := e
\end{array}
$$

States

With address arithmetic (new version):

$$\begin{aligned}
\text{Values} &= \text{Integers} \\
\text{Atoms} \cup \text{Addresses} &\subseteq \text{Integers} \\
&\quad \text{where Atoms and Addresses are disjoint} \\
\mathbf{nil} &\in \text{Atoms} \\
\text{Stores}_V &= V \to \text{Values} \\
\text{Heaps} &= \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \to \text{Values}) \\
\text{States}_V &= \text{Stores}_V \times \text{Heaps} \\
&\quad \text{where } V \text{ is a finite set of variables.}
\end{aligned}$$

(We assume that all but a finite number of nonnegative integers are addresses.)

		Store : x: 3, y: 4
		Heap : empty
Allocation	$x := \mathbf{cons}(1, 2)$;	⇓
		Store : x: 37, y: 4
		Heap : 37: 1, 38: 2
Lookup	$y := [x]$;	⇓
		Store : x: 37, y: 1
		Heap : 37: 1, 38: 2
Mutation	$[x + 1] := 3$;	⇓
		Store : x: 37, y: 1
		Heap : 37: 1, 38: 3
Deallocation	$\mathbf{dispose}(x + 1)$	⇓
		Store : x: 37, y: 1
		Heap : 37: 1

Note that:

- Expressions depend only upon the store.
  - no side effects or nontermination.
  - $\mathbf{cons}$ and $[-]$ are parts of commands.
- Allocation is nondeterminate.

Memory Faults

		Store :	x: 3, y: 4
		Heap :	empty
Allocation	$x := \mathbf{cons}(1, 2)$;		$\Downarrow$
		Store :	x: 37, y: 4
		Heap :	37: 1, 38: 2
Lookup	$y := [x]$;		$\Downarrow$
		Store :	x: 37, y: 1
		Heap :	37: 1, 38: 2
Mutation	$[x + 2] := 3$;		$\Downarrow$
			$\mathbf{abort}$

Faults can also be caused by out-of-range lookup or deallocation.

Operational Semantics

$$\frac{h([\![e]\!]_{intexp}\, s) = n}{(x := [e], (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow n\}, h))}$$

$$\frac{[\![e]\!]_{intexp}\, s \notin \mathrm{dom}(h)}{(x := [e], (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{[\![e]\!]_{intexp}\, s = \ell \quad \ell \in \mathrm{dom}(h)}{([e] := e', (s, h)) \longrightarrow (\textbf{skip}, (s, h\{\ell \rightsquigarrow [\![e']\!]_{intexp}\, s\}))}$$

$$\frac{[\![e]\!]_{intexp}\, s \notin \mathrm{dom}(h)}{([e] := e', (s, h)) \longrightarrow \textbf{abort}}$$

$$\frac{[\![e_1]\!]_{intexp}\, s = n_1 \quad [\![e_2]\!]_{intexp}\, s = n_2 \quad \{\ell, \ell + 1\} \cap \mathrm{dom}(h) = \emptyset}{(x := \textbf{cons}(e_1, e_2), (s, h)) \longrightarrow (\textbf{skip}, (s\{x \rightsquigarrow \ell\}, h\{\ell \rightsquigarrow n_1, \ell{+}1 \rightsquigarrow n_2\}))}$$

2. Assertions

Assertions

Standard predicate calculus:

$$\wedge \qquad \vee \qquad \neg \qquad \Rightarrow \qquad \forall \qquad \exists$$

plus:

- $\mathbf{emp}$ (empty heap)
  The heap is empty.
- $e \mapsto e'$ (singleton heap)
  The heap contains one cell, at address e with contents e'.
- $p_1 * p_2$ (separating conjunction)
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
- $p_1 -\!\!* p_2$ (separating implication)
  If the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

Some Abbreviations

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'.\, e \mapsto x' \quad \text{where } x' \text{ not free in } e$$

$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$$

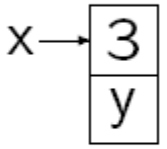
$$e \mapsto e_1, \ldots, e_n \stackrel{\text{def}}{=} e \mapsto e_1 * \cdots * e + n - 1 \mapsto e_n$$

$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e + n - 1 \hookrightarrow e_n$$

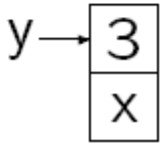
$$\text{iff } e \mapsto e_1, \ldots, e_n * \mathbf{true}$$

Examples of Separating Conjunction

1. $x \mapsto 3, y$ asserts that x points to an adjacent pair of cells containing 3 and y.

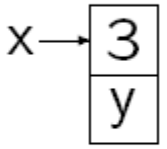


2. $y \mapsto 3, x$ asserts that y points to an adjacent pair of cells containing 3 and x.

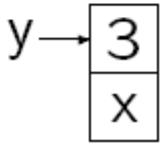


Examples of Separating Conjunction

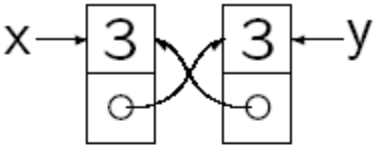
1. $x \mapsto 3, y$ asserts that x points to an adjacent pair of cells containing 3 and y.



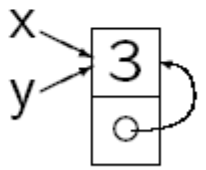
2. $y \mapsto 3, x$ asserts that y points to an adjacent pair of cells containing 3 and x.



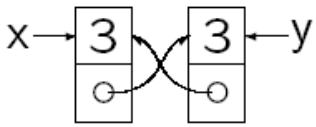
3. $x \mapsto 3, y \ * \ y \mapsto 3, x$ asserts that situations (1) and (2) hold for separate parts of the heap.



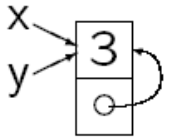
4. $x \mapsto 3, y \wedge y \mapsto 3, x$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



3. $x \mapsto 3, y \ * \ y \mapsto 3, x$ asserts that situations (1) and (2) hold for separate parts of the heap.



4. $x \mapsto 3, y \wedge y \mapsto 3, x$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



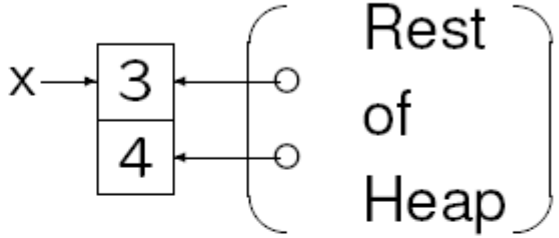
5. $x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

An Example of Separating Implication

Suppose p holds for

Store : x: $\alpha, \ldots$

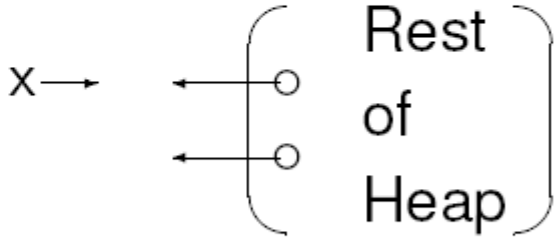
Heap : α: 3, $\alpha + 1$: 4, ...



Then $(x \mapsto 3, 4) \mathrel{-\!\!*} p$ holds for

Store : x: $\alpha, \ldots$

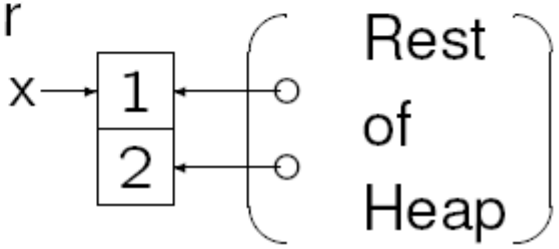
Heap : ...

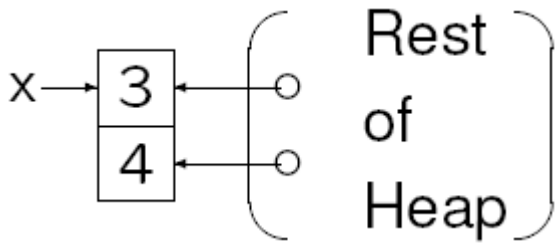


and $x \mapsto 1, 2 \ * \ ((x \mapsto 3, 4) \mathrel{-\!\!*} p)$ holds for

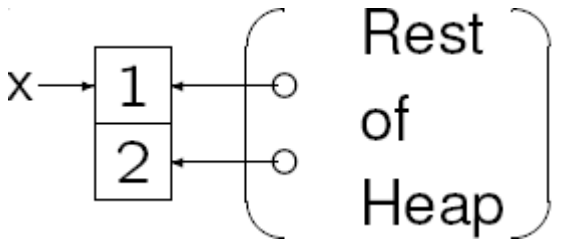
Store : x: $\alpha, \ldots$

Heap : α: 1, $\alpha + 1$: 2, ...





p



$\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)$

In particular,

$$\{\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\}\ [\mathsf{x}] := 3\ ;\ [\mathsf{x} + 1] := 4\ \{p\},$$

and more generally,

$$\{\mathsf{x} \mapsto -, - \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\}\ [\mathsf{x}] := 3\ ;\ [\mathsf{x} + 1] := 4\ \{p\}.$$

Next: Formal Semantics of Assertions

Recall States

With address arithmetic (new version):

$$\begin{aligned}
\text{Values} &= \text{Integers} \\
\text{Atoms} \cup \text{Addresses} &\subseteq \text{Integers} \\
&\quad \text{where Atoms and Addresses are disjoint} \\
\mathbf{nil} &\in \text{Atoms} \\
\text{Stores}_V &= V \to \text{Values} \\
\text{Heaps} &= \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \to \text{Values}) \\
\text{States}_V &= \text{Stores}_V \times \text{Heaps} \\
&\quad \text{where } V \text{ is a finite set of variables.}
\end{aligned}$$

(We assume that all but a finite number of nonnegative integers are addresses.)

Some Notation for Functions

We write

$$[\, x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$$

for the function with domain $\{x_1, \ldots, x_n\}$ that maps each x_i into y_i, and

$$[\, f \mid x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$$

for the function whose domain is the union of the domain of f with $\{x_1, \ldots, x_n\}$, that maps each x_i into y_i and all other members x of the domain of f into $f\, x$.

For heaps, we write

$$h_0 \perp h_1$$

when h_0 and h_1 have disjoint domains, and

$$h_0 \cdot h_1$$

to denote the union of heaps with disjoint domains.

The Meaning of Assertions

When s is a store, h is a heap, and p is an assertion whose free variables belong to the domain of s, we write

$$s, h \vDash p$$

to indicate that the state s, h *satisfies* p, or p *is true in* s, h, or p *holds in* s, h. Then:

$$s, h \vDash b \text{ iff } [\![b]\!]_{\mathsf{boolexp}} s = \mathbf{true},$$

$$s, h \vDash \neg\, p \text{ iff } s, h \vDash p \text{ is false},$$

$$s, h \vDash p_0 \wedge p_1 \text{ iff } s, h \vDash p_0 \text{ and } s, h \vDash p_1$$

(and similarly for $\vee$, $\Rightarrow$, $\Leftrightarrow$),

$$s, h \vDash \forall v.\ p \text{ iff } \forall x \in \mathbf{Z}.\ [\, s \mid v\colon x \,], h \vDash p,$$

$$s, h \vDash \exists v.\ p \text{ iff } \exists x \in \mathbf{Z}.\ [\, s \mid v\colon x \,], h \vDash p,$$

$$s, h \vDash \mathbf{emp} \text{ iff } \mathsf{dom}\, h = \{\},$$

$$s, h \vDash e \mapsto e' \text{ iff } \mathsf{dom}\, h = \{[\![e]\!]_{\mathsf{exp}} s\} \text{ and } h([\![e]\!]_{\mathsf{exp}} s) = [\![e']\!]_{\mathsf{exp}} s,$$

$$s, h \vDash p_0 * p_1 \text{ iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } s, h_0 \vDash p_0 \text{ and } s, h_1 \vDash p_1,$$

$$s, h \vDash p_0 \mathrel{-\!\!*} p_1 \text{ iff } \forall h'.\ (h' \perp h \text{ and } s, h' \vDash p_0) \text{ implies } s, h \cdot h' \vDash p_1.$$

When $s, h \vDash p$ holds for all states s, h (such that the domain of s contains the free variables of p), we say that p is *valid*.

When $s, h \vDash p$ holds for some state s, h, we say that p is *satisfiable*.

For Instance

$$
\begin{aligned}
s, h \vDash \mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1 \\
&\text{iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
&\quad \text{and } s, h_0 \vDash \mathsf{x} \mapsto 0 \\
&\quad \text{and } s, h_1 \vDash \mathsf{y} \mapsto 1 \\
&\text{iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
&\quad \text{and } \mathrm{dom}\, h_0 = \{s\,\mathsf{x}\} \text{ and } h_0(s\,\mathsf{x}) = 0 \\
&\quad \text{and } \mathrm{dom}\, h_1 = \{s\,\mathsf{y}\} \text{ and } h_1(s\,\mathsf{y}) = 1 \\
&\text{iff } s\,\mathsf{x} \neq s\,\mathsf{y} \\
&\quad \text{and } \mathrm{dom}\, h = \{s\,\mathsf{x}, s\,\mathsf{y}\} \\
&\quad \text{and } h(s\,\mathsf{x}) = 0 \text{ and } h(s\,\mathsf{y}) = 1 \\
&\text{iff } s\,\mathsf{x} \neq s\,\mathsf{y} \text{ and } h = [\,s\,\mathsf{x}\colon 0 \mid s\,\mathsf{y}\colon 1\,].
\end{aligned}
$$

Some Abbreviations

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'.\, e \mapsto x' \quad \text{where } x' \text{ not free in } e$$

$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$$

$$e \mapsto e_1, \ldots, e_n \stackrel{\text{def}}{=} e \mapsto e_1 * \cdots * e + n - 1 \mapsto e_n$$

$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e + n - 1 \hookrightarrow e_n$$

$$\text{iff } e \mapsto e_1, \ldots, e_n * \mathbf{true}$$

Examples

$$
\begin{aligned}
s, h \vDash \mathsf{x} \mapsto \mathsf{y} &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \vDash \mathsf{x} \mapsto - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \\
s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y} &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \vDash \mathsf{x} \hookrightarrow - &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \\
s, h \vDash \mathsf{x} \mapsto \mathsf{y}, \mathsf{z} &\text{ iff } h = [\,s\,\mathsf{x} \colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1 \colon s\,\mathsf{z}\,] \\
s, h \vDash \mathsf{x} \mapsto -, - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}, s\,\mathsf{x} + 1\} \\
s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{z} &\text{ iff } h \supseteq [\,s\,\mathsf{x} \colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1 \colon s\,\mathsf{z}\,] \\
s, h \vDash \mathsf{x} \hookrightarrow -, - &\text{ iff } \operatorname{dom} h \supseteq \{s\,\mathsf{x}, s\,\mathsf{x} + 1\}.
\end{aligned}
$$

More Examples of $*$

Suppose s x and s y are distinct addresses, so that

$$h_0 = [\, s\,\mathsf{x}\colon 0\,] \qquad \text{and} \qquad h_1 = [\, s\,\mathsf{y}\colon 1\,]$$

are heaps with disjoint domains. Then

If p is:	then $s, h \vDash p$ iff:
$\mathsf{x} \mapsto 0$	$h = h_0$
$\mathsf{y} \mapsto 1$	$h = h_1$
$\mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \ * \ \mathsf{x} \mapsto 0$	**false**
$\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1$	$h = h_0$ or $h = h_1$

More Examples of $*$

Suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct addresses, so that

$$h_0 = [\,s\,\mathsf{x}\colon 0\,] \qquad \text{and} \qquad h_1 = [\,s\,\mathsf{y}\colon 1\,]$$

are heaps with disjoint domains. Then

If p is:	then $s, h \vDash p$ iff:
$\mathsf{x} \mapsto 0 \;*\; (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$(\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1) \;*\; (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \;*\; \mathsf{y} \mapsto 1 \;*\; (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$\mathbf{false}$
$\mathsf{x} \mapsto 0 \;*\; \mathbf{true}$	$h_0 \subseteq h$

Next: Inference Rules for Assertions

Inference Rules

$$\frac{\mathcal{P}_1 \quad \cdots \quad \mathcal{P}_n}{\mathcal{C}}$$

(zero or more premisses)
(one conclusion)

Inference

Inference Rules

$$\frac{p_0 \qquad p_0 \Rightarrow p_1}{p_1}$$

$$e_2 = e_1 \Rightarrow e_1 = e_2$$

$$x + 0 = x$$

Instances

$$\frac{x + 0 = x \qquad x + 0 = x \Rightarrow x = x + 0}{x = x + 0}$$

$$x + 0 = x \Rightarrow x = x + 0$$

$$x + 0 = x$$

A Proof

$$x + 0 = x$$
$$x + 0 = x \Rightarrow x = x + 0$$
$$x = x + 0.$$

A Subtlety

$\dfrac{p}{q}$ is sound iff, for all instances,
if p is valid, then q is valid, i.e.,
if p holds in all states, then q holds in all states.

$\dfrac{}{p \Rightarrow q}$ is sound iff, for all instances,
$p \Rightarrow q$ is valid, i.e.,
for all states, if p holds, then q holds.

For example,

$$\frac{p}{\forall v.\ p} \quad \text{e.g.} \quad \frac{\mathrm{x} + \mathrm{y} = \mathrm{y} + \mathrm{x}}{\forall \mathrm{x}.\ \mathrm{x} + \mathrm{y} = \mathrm{y} + \mathrm{x}} \quad \text{or} \quad \frac{\mathrm{x} = 0}{\forall \mathrm{x}.\ \mathrm{x} = 0}$$

is sound, but

$$\frac{}{p \Rightarrow \forall v.\ p} \quad \text{e.g.} \quad \frac{}{\mathrm{x} = 0 \Rightarrow \forall \mathrm{x}.\ \mathrm{x} = 0}$$

is not sound.

Inference Rules for $*$ and $-\!\!*$

$$p_0 * p_1 \Leftrightarrow p_1 * p_0$$

$$(p_0 * p_1) * p_2 \Leftrightarrow p_0 * (p_1 * p_2)$$

$$p * \mathbf{emp} \Leftrightarrow p$$

$$(p_0 \vee p_1) * q \Leftrightarrow (p_0 * q) \vee (p_1 * q)$$

$$(p_0 \wedge p_1) * q \Rightarrow (p_0 * q) \wedge (p_1 * q)$$

$$(\exists x.\, p_0) * p_1 \Leftrightarrow \exists x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1$$

$$(\forall x.\, p_0) * p_1 \Rightarrow \forall x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1$$

$$\frac{p_0 \Rightarrow p_1 \qquad q_0 \Rightarrow q_1}{p_0 * q_0 \Rightarrow p_1 * q_1} \quad \text{(monotonicity)}$$

$$\frac{p_0 * p_1 \Rightarrow p_2}{p_0 \Rightarrow (p_1 -\!\!* p_2)} \quad \text{(currying)} \qquad \frac{p_0 \Rightarrow (p_1 -\!\!* p_2)}{p_0 * p_1 \Rightarrow p_2.} \quad \text{(decurrying)}$$

Two Unsound Axiom Schemata

$p \Rightarrow p * p$ (Contraction — unsound)

e.g. $p : \mathsf{x} \mapsto 1$

$p * q \Rightarrow p$ (Weakening — unsound)

e.g. $p : \mathsf{x} \mapsto 1$

$q : \mathsf{y} \mapsto 2$

Some Axiom Schemata for $\mapsto$

$$e_1 \mapsto e_1' \wedge e_2 \mapsto e_2' \Leftrightarrow e_1 \mapsto e_1' \wedge e_1 = e_2 \wedge e_1' = e_2'$$

$$e_1 \hookrightarrow e_1' \ * \ e_2 \hookrightarrow e_2' \Rightarrow e_1 \neq e_2$$

$$\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -)$$

$$(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).$$

(Regrettably, these are far from complete.)

Next

- We will introduce some important/interesting classes of assertions:
  - Pure assertions
  - Strictly exact assertions
  - Precise assertions
  - Intuitionistic assertions

Pure Assertions

An assertion p is *pure* iff, for all stores s and all heaps h and h',

$$s, h \vDash p \text{ iff } s, h' \vDash p.$$

A sufficient syntactic criteria is that an assertion is pure if it does not contain $\mathbf{emp}$, $\mapsto$, or $\hookrightarrow$.

Axiom Schemata for Purity

$p_0 \wedge p_1 \Rightarrow p_0 \ * \ p_1$	when p_0 or p_1 is pure
$p_0 \ * \ p_1 \Rightarrow p_0 \wedge p_1$	when p_0 and p_1 are pure
$(p \wedge q) \ * \ r \Leftrightarrow (p \ * \ r) \wedge q$	when q is pure
$(p_0 \mathrel{-\!\!*} p_1) \Rightarrow (p_0 \Rightarrow p_1)$	when p_0 is pure
$(p_0 \Rightarrow p_1) \Rightarrow (p_0 \mathrel{-\!\!*} p_1)$	when p_0 and p_1 are pure.

Strictly Exact Assertions (Yang)

An assertion is *strictly exact* iff, for all stores s and all heaps h and h',

$$s, h \vDash p \text{ and } s, h' \vDash p \text{ implies } h = h'.$$

Strictly Exact Assertions (Yang)

An assertion is *strictly exact* iff, for all stores s and all heaps h and h',

$$s, h \vDash p \text{ and } s, h' \vDash p \text{ implies } h = h'.$$

Examples of Strictly Exact Assertions

- $\mathbf{emp}$.
- $e \mapsto e'$.
- $p * q$, when p and q are strictly exact.
- $p \wedge q$, when p or q is strictly exact.
- p, when $p \Rightarrow q$ is valid and q is strictly exact.

Proposition 4 *When q is strictly exact,*

$$((q \ * \ \mathbf{true}) \wedge p) \Rightarrow (q \ * \ (q \mathrel{-\!\!*} p))$$

is valid.

PROOF Suppose $s, h \vDash (q \ * \ \mathbf{true}) \wedge p$, so that $s, h \vDash q \ * \ \mathbf{true}$ and $s, h \vDash p$. Then there are heaps h_0 and h_1 such that $h_0 \perp h_1$, $h_0 \cdot h_1 = h$, and $s, h_0 \vDash q$.

To see that $s, h_1 \vDash q \mathrel{-\!\!*} p$, let h' be any heap such that $h' \perp h_1$ and $s, h' \vDash q$. Since q is strictly exact, $h' = h_0$, so that $h' \cdot h_1 = h_0 \cdot h_1 = h$, and thus $s, h' \cdot h_1 \vDash p$.

Then $s, h_0 \cdot h_1 \vDash q \ * \ (q \mathrel{-\!\!*} p)$, so that $s, h \vDash q \ * \ (q \mathrel{-\!\!*} p)$.

END OF PROOF

Proposition 4 *When q is strictly exact,*

$$((q \ * \ \mathbf{true}) \wedge p) \Rightarrow (q \ * \ (q \mathrel{-\!\!*} p))$$

is valid.

For example, taking q to be the strictly exact assertion $e \mapsto e'$ gives the final axiom schema for $\mapsto$:

$$(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).$$

Precise Assertions

An assertion q is *precise* iff

For all s and h, there is at most one $h' \subseteq h$ such that

$$s, h' \vDash q.$$

Examples of Precise Assertions

- Strictly exact assertions.
- $e \mapsto -$.
- $p * q$, when p and q are precise.
- $p \wedge q$, when p or q is precise.
- p, when $p \Rightarrow q$ is valid and q is precise.
- list α e and $\exists \alpha$. list α e.

Examples of Imprecise Assertions

- $\mathbf{true}$
- $\mathbf{emp} \vee \mathsf{x} \mapsto 10$
- $\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 10$
- $\exists \mathsf{x}.\ \mathsf{x} \mapsto 10$

Preciseness and Distributivity

The semi-distributive laws

$$(p_0 \wedge p_1) * q \Rightarrow (p_0 * q) \wedge (p_1 * q)$$
$$(\forall x.\ p) * q \Rightarrow \forall x.\ (p * q) \quad \text{when } x \text{ not free in } q$$

are valid for all assertions. But their converses

$$(p_0 * q) \wedge (p_1 * q) \Rightarrow (p_0 \wedge p_1) * q$$
$$\forall x.\ (p * q) \Rightarrow (\forall x.\ p) * q \quad \text{when } x \text{ not free in } q$$

are not. For example, when

$$s(\mathrm{x}) = 1 \quad s(\mathrm{y}) = 2 \quad h = [\,1\colon 10 \mid 2\colon 20\,],$$

the assertion

$$(\mathrm{x} \mapsto 10 * (\mathrm{x} \mapsto 10 \vee \mathrm{y} \mapsto 20)) \wedge (\mathrm{y} \mapsto 20 * (\mathrm{x} \mapsto 10 \vee \mathrm{y} \mapsto 20))$$

is true, but

$$((\mathrm{x} \mapsto 10 \wedge \mathrm{y} \mapsto 20) * (\mathrm{x} \mapsto 10 \vee \mathrm{y} \mapsto 20))$$

is false.

Proposition 5 *When q is precise,*

$$(p_0 \ * \ q) \wedge (p_1 \ * \ q) \Rightarrow (p_0 \wedge p_1) \ * \ q$$

is valid. When q is precise and x is not free in q,

$$\forall x.\ (p \ * \ q) \Rightarrow (\forall x.\ p) \ * \ q$$

is valid.

PROOF (first law) Suppose $s, h \vDash (p_0 \ * \ q) \wedge (p_1 \ * \ q)$. Then there are:

- An $h_0 \subseteq h$ such that $s, h - h_0 \vDash p_0$ and $s, h_0 \vDash q$, and
- An $h_1 \subseteq h$ such that $s, h - h_1 \vDash p_1$ and $s, h_1 \vDash q$.

Thus, since q is precise,

$$h_0 = h_1$$
$$h - h_0 = h - h_1$$
$$s, h - h_0 \vDash p_0 \wedge p_1$$
$$s, h \vDash (p_0 \wedge p_1) \ * \ q.$$

END OF PROOF

Intuitionistic Assertions

An assertion i is *intuitionistic* iff, for all stores s and heaps h and h':

$$(h \subseteq h' \text{ and } s, h \vDash i) \text{ implies } s, h' \vDash i.$$

Assume i and i' are intuitionistic assertions, and p is any assertion. Then:

- The following assertions are intuitionistic:

Any pure assertion	$p * i$
$p \mathrel{-\!\!*} i$	$i \mathrel{-\!\!*} p$
$i \wedge i'$	$i \vee i'$
$\forall v.\ i$	$\exists v.\ i$

and as special cases:

$$p * \mathbf{true} \qquad \mathbf{true} \mathrel{-\!\!*} p \qquad e \hookrightarrow e'.$$

Assume i and i' are intuitionistic assertions, and p is any assertion. Then:

- The following inference rules are sound:

$$(i \ * \ i') \Rightarrow (i \wedge i')$$

$$(i \ * \ p) \Rightarrow i \qquad i \Rightarrow (p \mathrel{-\!\!*} i)$$

$$\frac{p \Rightarrow i}{(p \ * \ \mathbf{true}) \Rightarrow i} \qquad \frac{i \Rightarrow p}{i \Rightarrow (\mathbf{true} \mathrel{-\!\!*} p).}$$

The last two of these rules, in conjunction with the rules

$$p \Rightarrow (p \ * \ \mathbf{true}) \qquad (\mathbf{true} \mathrel{-\!\!*} p) \Rightarrow p,$$

which hold for all assertions, imply that

- $i \Leftrightarrow (i \ * \ \mathbf{true})$.
- $(\mathbf{true} \mathrel{-\!\!*} i) \Leftrightarrow i$.

Summary

- Assertions

$$
\begin{aligned}
s, h \vDash \mathbf{emp} \text{ iff } & \operatorname{dom} h = \{\}, \\
s, h \vDash e \mapsto e' \text{ iff } & \operatorname{dom} h = \{[\![e]\!]_{\mathsf{exp}} s\} \text{ and} \\
& \qquad h([\![e]\!]_{\mathsf{exp}} s) = [\![e']\!]_{\mathsf{exp}} s, \\
s, h \vDash p_0 * p_1 \text{ iff } & \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\
& \qquad s, h_0 \vDash p_0 \text{ and } s, h_1 \vDash p_1, \\
s, h \vDash p_0 \mathrel{-\!\!*} p_1 \text{ iff } & \forall h'.\ (h' \perp h \text{ and } s, h' \vDash p_0) \text{ implies} \\
& \qquad s, h \cdot h' \vDash p_1.
\end{aligned}
$$

- Next: Specifications and Inference Rules

3. Specifications and Inference Rules

Specifications

- $\{p\}\ c\ \{q\}$ (partial correctness)

  Starting in any state in which p holds:
  - No execution of c aborts.
  - When some execution of c terminates in a final state, then q holds in the final state.

- $[\, p \,]\ c\ [\, q \,]$ (total correctness)

  Starting in any state in which p holds:
  - No execution of c aborts.
  - Every execution of c terminates.
  - When some execution of c terminates in a final state, then q holds in the final state.

The Differences with Hoare Logic

- Specifications are universally quantified implicitly over both stores and heaps,
- Specifications are universally quantified implicitly over all possible executions.
- Any execution (starting in a state satisfying p) that gives a memory fault falsifies both partial and total specifications. Thus:

$\bullet\bullet\bullet$ Well-specified programs don't go wrong. $\bullet\bullet\bullet$

Enforcing Record Boundaries

The fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. For example, in

$$c_0 \,;\, \mathsf{x} := \mathbf{cons}(1,2) \,;\, c_1 \,;\, [\mathsf{x}+2] := 7,$$

no allocation performed by the subcommand c_0 or c_1 can be guaranteed to allocate the location $\mathsf{x}+2$.

It follows that there is no postcondition that makes the specification

$$\{\mathbf{true}\}\; c_0 \,;\, \mathsf{x} := \mathbf{cons}(1,2) \,;\, c_1 \,;\, [\mathsf{x}+2] := 7\; \{?\}$$

valid.

On the Other Hand (Gluing Records)

$$\{x \mapsto - \ * \ y \mapsto -\}$$
$$\textbf{if } y = x + 1 \textbf{ then skip else}$$
$$\quad \textbf{if } x = y + 1 \textbf{ then } x := y \textbf{ else}$$
$$\quad\quad (\textbf{dispose } x \,;\, \textbf{dispose } y \,;\, x := \textbf{cons}(1, 2))$$
$$\{x \mapsto -, -\}.$$

Also valid total correctness spec.

Examples of Valid Specifications

$$\{\mathbf{emp}\}\ \mathsf{x} := \mathbf{cons}(1,2)\ \{\mathsf{x} \mapsto 1,2\}$$

$$\{\mathsf{x} \mapsto 1,2\}\ \mathsf{y} := [\mathsf{x}]\ \{\mathsf{x} \mapsto 1,2 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1,2 \wedge \mathsf{y} = 1\}\ [\mathsf{x}+1] := 3\ \{\mathsf{x} \mapsto 1,3 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1,3 \wedge \mathsf{y} = 1\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x}+1 \mapsto 3 \wedge \mathsf{y} = 1\}$$

Also valid total correctness spec.

Hoare's Inference Rules

The command-specific inference rules of Hoare logic remain sound, as do structural rules such as

- Strengthening Precedent

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\, c\, \{r\}.}$$

- Weakening Consequent

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\, c\, \{r\}.}$$

The Failure of the Rule of Constancy

On the other hand,

- Rule of Constancy

$$\frac{\{p\}\ c\ \{q\}}{\{p \wedge r\}\ c\ \{q \wedge r\},}$$

  where no variable occurring free in r is modified by c.

is *unsound*, since, for example

$$\frac{\{\mathsf{x} \mapsto -\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4\}}{\{\mathsf{x} \mapsto - \wedge \mathsf{y} \mapsto 3\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4 \wedge \mathsf{y} \mapsto 3\}}$$

fails when $\mathsf{x} = \mathsf{y}$.

The Frame Rule

Instead, we have the

- Frame Rule (O'Hearn)

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

  where no variable occurring free in r is modified by c.

The frame rule is the key to "local reasoning" about the heap:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged. (O'Hearn)

Local Reasoning

- The set of variables and heap cells that may actually be used by a command (starting from a given state) is called its *footprint*.
- If $\{p\}\ c\ \{q\}$ is valid, then p will assert that the heap contains all the cells in the footprint of c (excluding the cells that are freshly allocated by c).
- If p asserts that the heap contains *only* cells in the footprint of c, then $\{p\}\ c\ \{q\}$ is a *local specification*.
- If c' contains c, it may have a larger footprint described, say, by $p\ *\ r$. Then the frame rule is needed to move from $\{p\}\ c\ \{q\}$ to $\{p\ *\ r\}\ c\ \{q\ *\ r\}$.

The Frame Rule (O'Hearn) (FR)

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

where no variable occurring free in r is modified by c.

For instance,

$$\frac{\{\mathbf{list}\ \alpha\ \mathsf{i}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j}\}}{\{\mathbf{list}\ \alpha\ \mathsf{i}\ *\ \mathbf{list}\ \gamma\ \mathsf{x}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j}\ *\ \mathbf{list}\ \gamma\ \mathsf{x}\}},$$

(assuming “Reverse List” does not modify x or γ).

Soundness of the frame rule is sensitive to the semantics of programming language

- Suppose **dispose** x :
  - If heap does not contain location x, then do nothing
  - Otherwise, free the location x

Soundness of the frame rule is sensitive to the semantics of programming language

- Suppose **dispose** x :
  - If heap does not contain location x, then do nothing
  - Otherwise, free the location x

$$\{\mathbf{emp}\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathbf{emp}\}.$$

Then the frame rule would give

$$\{\mathbf{emp} * \mathsf{x} \mapsto 10\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathbf{emp} * \mathsf{x} \mapsto 10\},$$

and therefore

$$\{\mathsf{x} \mapsto 10\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x} \mapsto 10\},$$

which is patently false.

Why the Frame Rule is Sound

We define:

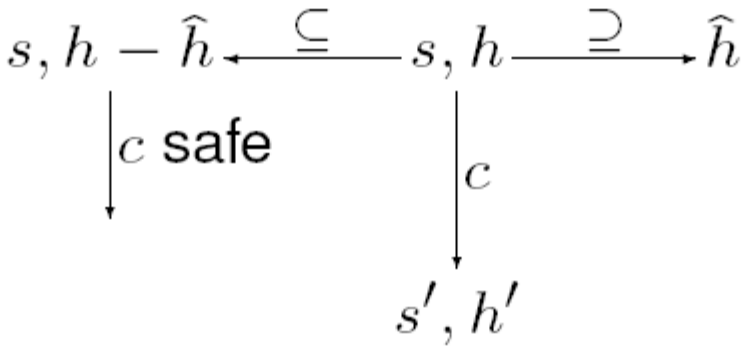
If, starting in the state s, h, no execution of a command c aborts, then c is *safe at* s, h.

If, starting in the state s, h, every execution of c terminates without aborting, then c *must terminate normally* at s, h.

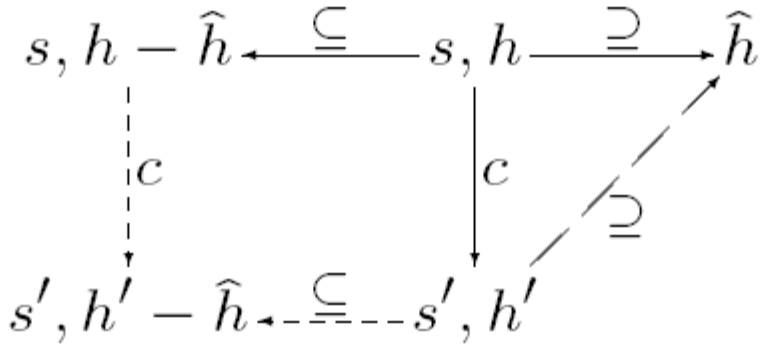
Then our programming language satisfies:

Safety Monotonicity If $\widehat{h} \subseteq h$ and c is safe at $s, h - \widehat{h}$, then c is safe at s, h. If $\widehat{h} \subseteq h$ and c must terminate normally at $s, h - \widehat{h}$, then c must terminate normally at s, h.

The Frame Property If $\widehat{h} \subseteq h$, c is safe at $s, h - \widehat{h}$, and some execution of c starting at s, h terminates normally in the state s', h',



then $\widehat{h} \subseteq h'$ and some execution of c starting at $s, h - \widehat{h}$, terminates normally in the state $s', h' - \widehat{h}$.



Proposition 11 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

Locality : Safety monotonicity + Frame property

Soundness

if $\vdash \{P\}\ C\ \{Q\}$
then $\vDash \{P\}\ C\ \{Q\}$

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P * R\}\ C\ \{Q * R\}}(\dagger)$$

†provided R doesn't mention any variable modified by C

assume $\vDash \{P\}\ C\ \{Q\}$
show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

assume $\models \{P\}\ C\ \{Q\}$

show $\models \{P * R\}\ C\ \{Q * R\}$

Soundness of Frame Rule

assume $\models \{P\}\ C\ \{Q\}$

If $\sigma \models P * R$, then:
(C, σ) does not abort, and $(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \models Q * R$

show $\models \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

(C, σ) does not abort, and $(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

(C, σ) does not abort

$(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

$\sigma = \sigma_1 \cdot \sigma_2$

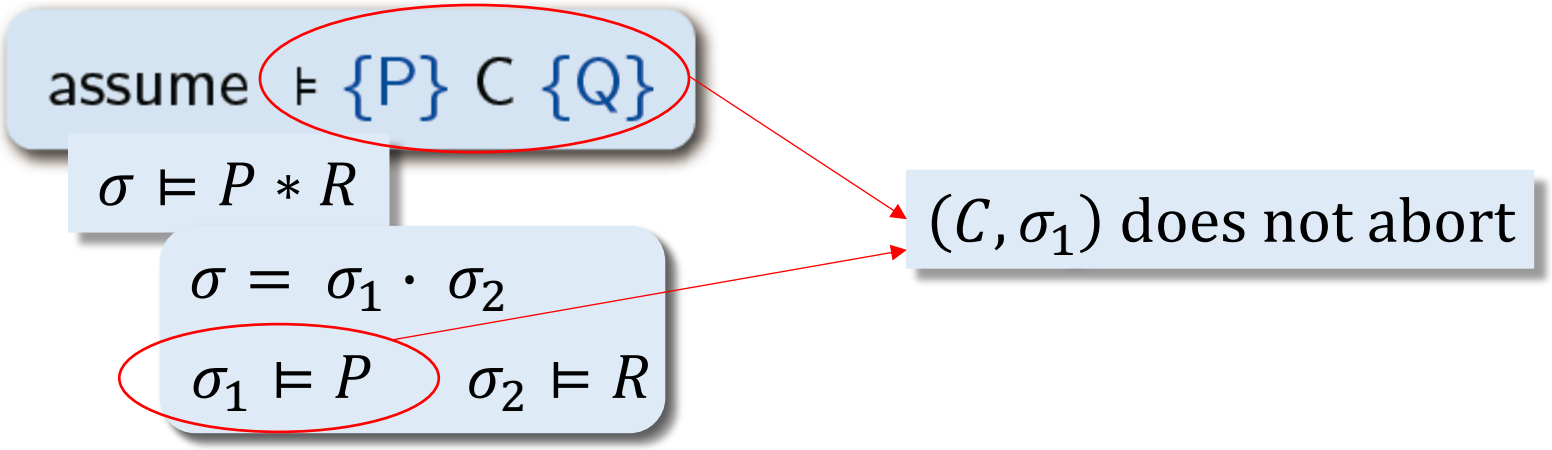
$\sigma_1 \vDash P \qquad \sigma_2 \vDash R$

(C, σ) does not abort

$(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

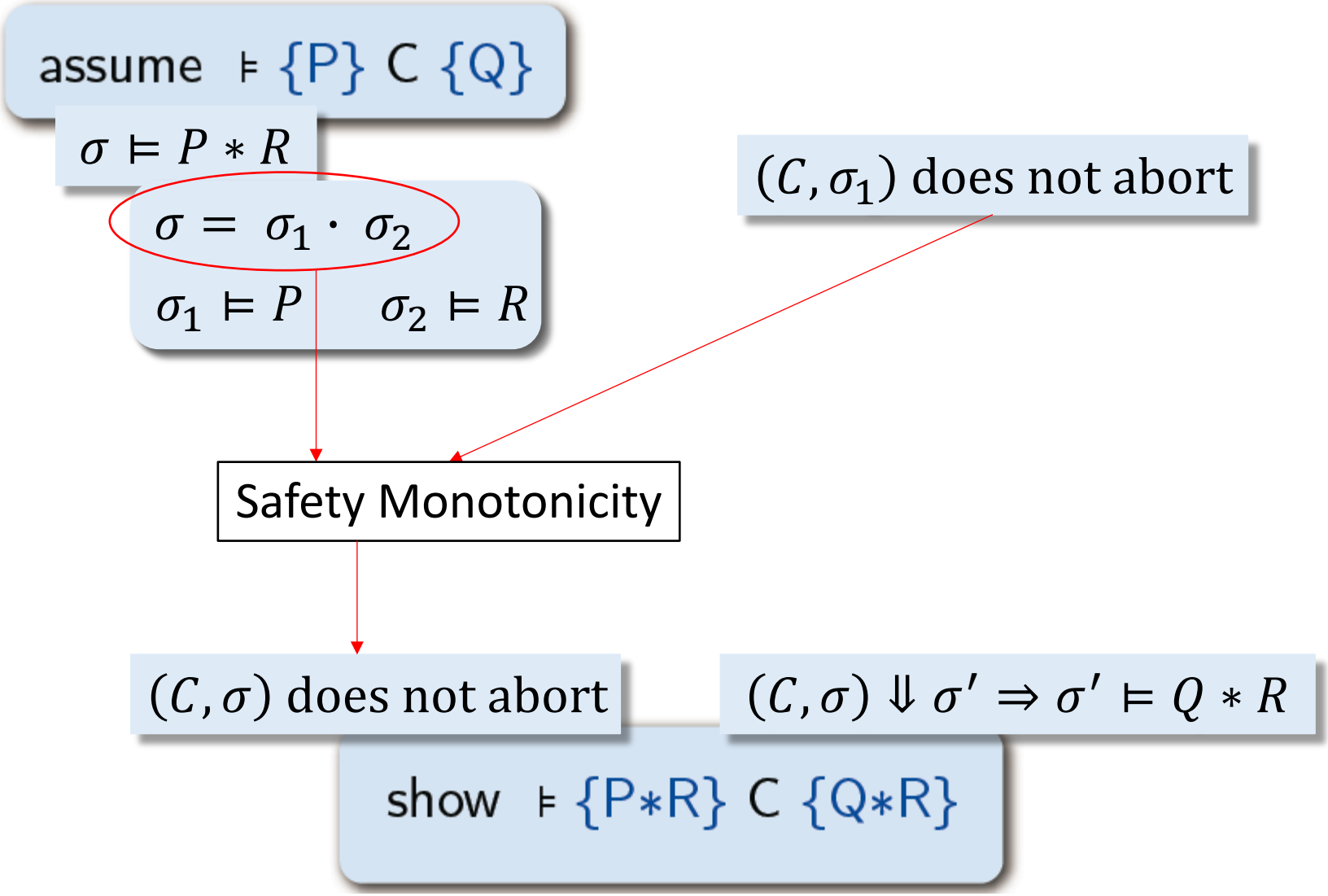


(C, σ) does not abort

$(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \vDash Q * R$

show ⊨ {P∗R} C {Q∗R}

Soundness of Frame Rule



Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

$\sigma = \sigma_1 \cdot \sigma_2$

$\sigma_1 \vDash P \qquad \sigma_2 \vDash R$

(C, σ_1) does not abort

$(C, \sigma) \Downarrow \sigma' \Rightarrow \sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

$\sigma = \sigma_1 \cdot \sigma_2$

$\sigma_1 \vDash P \qquad \sigma_2 \vDash R$

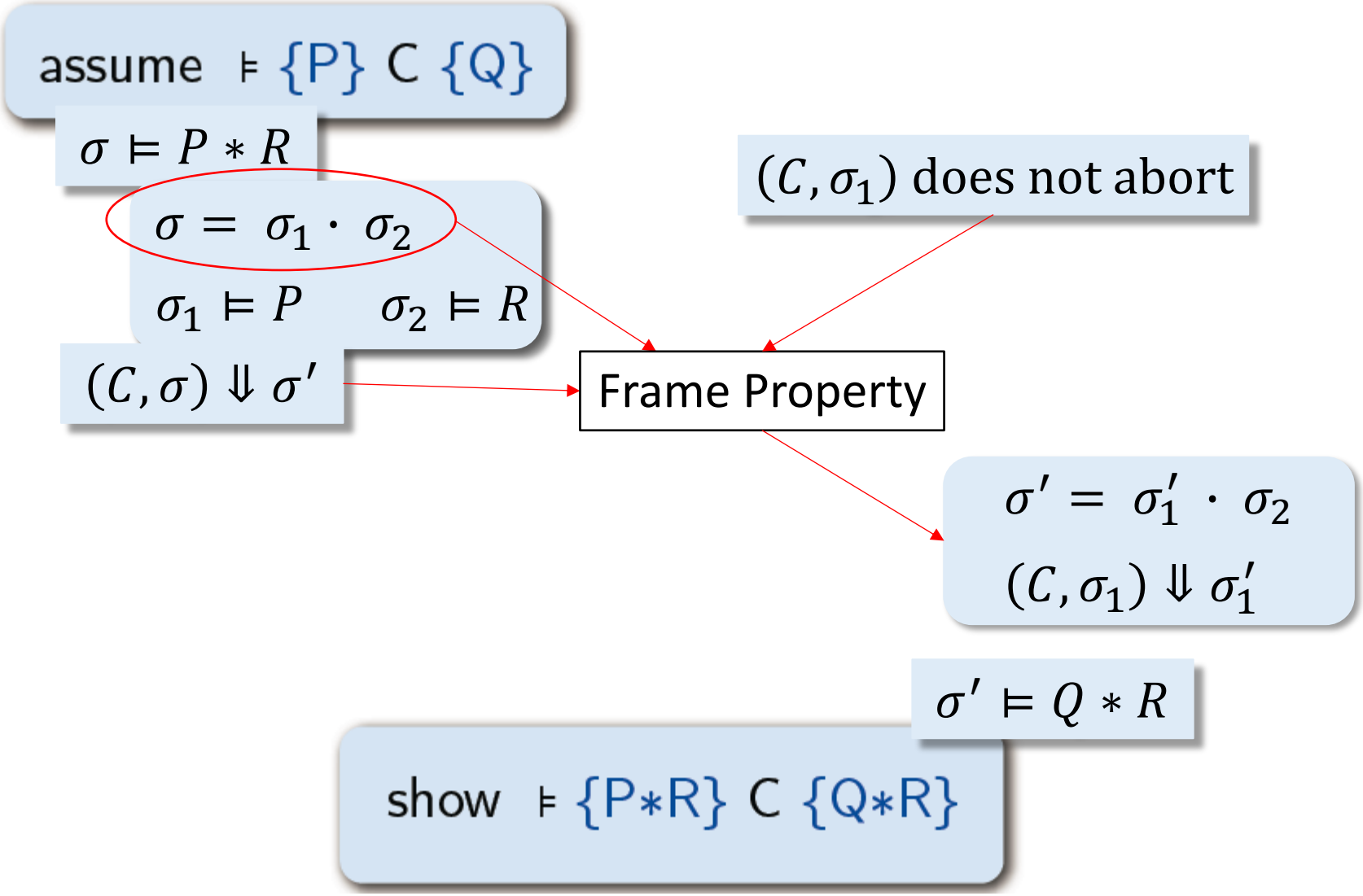
$(C, \sigma) \Downarrow \sigma'$

(C, σ_1) does not abort

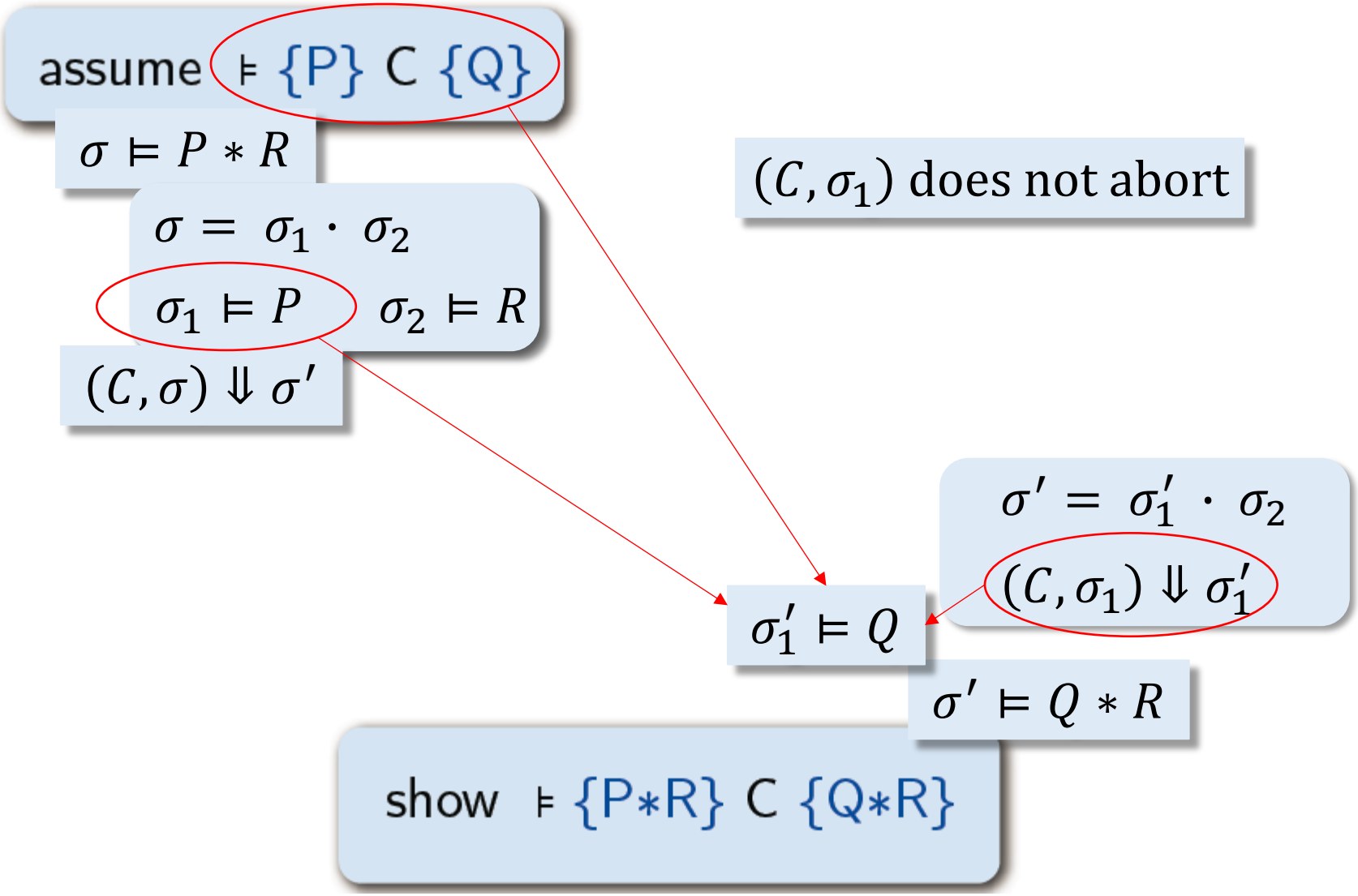
$\sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule



Soundness of Frame Rule



Soundness of Frame Rule

assume $\vDash \{P\}\ C\ \{Q\}$

$\sigma \vDash P * R$

$\sigma = \sigma_1 \cdot \sigma_2$

$\sigma_1 \vDash P \qquad \sigma_2 \vDash R$

$(C, \sigma) \Downarrow \sigma'$

(C, σ_1) does not abort

$\sigma' = \sigma_1' \cdot \sigma_2$

$(C, \sigma_1) \Downarrow \sigma_1'$

$\sigma_1' \vDash Q$

$\sigma' \vDash Q * R$

show $\vDash \{P*R\}\ C\ \{Q*R\}$

Soundness of Frame Rule

Proposition 11 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

$$
\begin{array}{rcl}
(comm)\ c & ::= & \mathbf{skip} \mid x := e \mid c_1; c_2 \\
& \mid & \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \\
& \mid & x := \mathbf{cons}(e_1, \ldots, e_n) \\
& \mid & \mathbf{dispose}(e) \\
& \mid & x := [e] \\
& \mid & [e] := e
\end{array}
$$

(You will need to prove this language satisfies locality in your homework.
For simplicity, you can use $x := \mathbf{cons}(e_1, e_2)$ instead of $x := \mathbf{cons}(e_1, \ldots, e_n)$.)

Inference Rules for Mutation

The local form (MUL):

$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

The global form (MUG):

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}.}$$

The backward-reasoning form (MUBR):

$$\frac{}{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}.}$$

Inference Rules for Mutation

The local form (MUL):

$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}}.$$

The global form (MUG):

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}}.$$

One can derive (MUG) from (MUL) by using the frame rule:

$$\begin{array}{l} \{(e \mapsto -)\ *\ r\} \\ \left.\begin{array}{l} \{e \mapsto -\} \\ [e] := e' \\ \{e \mapsto e'\} \end{array}\right\} *\ r \\ \{(e \mapsto e')\ *\ r\}, \end{array}$$

Inference Rules for Mutation

The local form (MUL):

$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

The global form (MUG):

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}.}$$

To go in the opposite direction it is only necessary to take r to be $\mathbf{emp}$:

$$\begin{array}{l}\{e \mapsto -\}\\ \{(e \mapsto -)\ *\ \mathbf{emp}\}\\ [e] := e'\\ \{(e \mapsto e')\ *\ \mathbf{emp}\}\\ \{e \mapsto e'\}.\end{array}$$

The global form (MUG):

$$\frac{}{\{(e \mapsto -) \ * \ r\} \ [e] := e' \ \{(e \mapsto e') \ * \ r\}.}$$

The backward-reasoning form (MUBR):

$$\frac{}{\{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)\} \ [e] := e' \ \{p\}.}$$

One can derive (MUBR) from (MUG) by taking r to be $(e \mapsto e') \mathrel{-\!\!*} p$ and using the law $q \ * \ (q \mathrel{-\!\!*} p) \Rightarrow p$:

$\{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)\}$
$[e] := e'$
$\{(e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)\}$
$\{p\}.$

The global form (MUG):

$$\frac{}{\{(e \mapsto -) \ * \ r\} \ [e] := e' \ \{(e \mapsto e') \ * \ r\}.}$$

The backward-reasoning form (MUBR):

$$\frac{}{\{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)\} \ [e] := e' \ \{p\}.}$$

One can go in the opposite direction by taking p to be $(e \mapsto e') \ * \ r$ and using $(p \ * \ r) \Rightarrow (p \ * \ (q \mathrel{-\!\!*} (q \ * \ r)))$:

$\{(e \mapsto -) \ * \ r\}$

$\{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} ((e \mapsto e') \ * \ r))\}$

$[e] := e'$

$\{(e \mapsto e') \ * \ r\}.$

Inference Rules for Deallocation

The local form (DISL):

$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}.$$

The global (and backward-reasoning) form (DISBR):

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}}.$$

One can derive (DISBR) from (DISL) by using (FR); one can go in the opposite direction by taking r to be $\mathbf{emp}$.

Inference Rules for Allocation and Lookup

These are *generalized assignment commands*, but they don't obey the assignment rule, e.g.

$$\{\mathbf{cons}(1,2) = \mathbf{cons}(1,2)\}\ \mathsf{x} := \mathbf{cons}(1,2)\ \{\mathsf{x} = \mathsf{x}\}$$

$\uparrow$

syntactically illegal

$\downarrow$

$$\{[\mathsf{y}] = [\mathsf{y}]\}\ \mathsf{x} := [\mathsf{y}]\ \{\mathsf{x} = \mathsf{x}\}$$

Inference Rules for Nonoverwriting Allocation

We abbreviate the sequence $e_1, \ldots, e_n$ of expressions by $\overline{e}$:

- The local form (CONSNOL)

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}\},}$$

  where $v \notin \mathsf{FV}(\overline{e})$.

- The global form (CONSNOG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\},}$$

  where $v \notin \mathsf{FV}(\overline{e}, r)$.

Again, one can derive the global form from the local by using the frame rule, and the local from the global by taking r to be $\mathbf{emp}$.

Inference Rules for General Allocation

- The local form (CONSL)

$$\frac{}{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\},}$$

  where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \rightarrow v'$ (i.e., each e_i' denotes $e_i/v \rightarrow v'$).

- The global form (CONSG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\},}$$

  where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \rightarrow v'$, and r' denotes $r/v \rightarrow v'$.

- The backward-reasoning form (CONSBR)

$$\frac{}{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\},}$$

  where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \rightarrow v''$.

The global form (CONSG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \rightarrow v'$, and r' denotes $r/v \rightarrow v'$.

Recall Forward Assignment Rule

$$\frac{}{\{p\}\ v := e\ \{\exists v'.\ v = e' \wedge p'\}}$$

where $v' \notin \{v\} \cup \mathsf{FV}(e) \cup \mathsf{FV}(p)$, e' is $e/v \rightarrow v'$, and p' is $p/v \rightarrow v'$. The quantifier can be omitted when v does not occur in e or p.

An Instance of (CONSG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\},}$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

An Instance:

$$\{\mathsf{list}\ \alpha\ \mathsf{i}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}\}.$$

An Inadequate Local Rule

From (CONSG):

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

we can derive

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\}},$$

where v' is distinct from v and $v' \notin \mathsf{FV}(\overline{e})$.

by taking r to be $\mathbf{emp}$.

An Inadequate Local Rule

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\}},$$

where v' is distinct from v and $v' \notin \mathsf{FV}(\overline{e})$.

But this rule is too weak. For instance,

$$\{\mathbf{emp}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\}$$

gives no information about the second component of the new record.

An *Adequate* Local Rule

- The local form (CONSL)

$$\frac{}{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\},}$$

where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$.

Here the existential quantifier is dropped and v' becomes a variable that is not modified by $v := \mathbf{cons}(\overline{e})$. For example,

$$\{\mathsf{i} = \mathsf{j} \wedge \mathbf{emp}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\mathsf{i} \mapsto 3, \mathsf{j}\}$$

shows that the value of j in the postcondition is value of i before the assignment.

From (CONSG):

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

we derive (CONSL):

$$\frac{}{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\}},$$

where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$.

$$\begin{array}{l}
\{v = v' \wedge \mathbf{emp}\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v''.\ (v \mapsto \overline{e}'')\ *\ (v'' = v' \wedge \mathbf{emp})\} \\
\{\exists v''.\ ((v \mapsto \overline{e}'') \wedge v'' = v')\ *\ \mathbf{emp}\} \\
\{\exists v''.\ (v \mapsto \overline{e}'') \wedge v'' = v'\} \\
\{\exists v''.\ (v \mapsto \overline{e}') \wedge v'' = v'\} \\
\{v \mapsto \overline{e}'\}.
\end{array}$$

From (CONSL) to (CONSG)

Existential Quantification (EQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\}}$$

where v is not free in c

$$
\begin{array}{l}
\{r\} \\
\{\exists v'.\ v = v' \wedge r\} \\
\left.\begin{array}{l}
\quad \{v = v' \wedge r\} \\
\quad \{v = v' \wedge r'\} \\
\quad \{v = v' \wedge (\mathbf{emp} \ * \ r')\} \\
\quad \{(v = v' \wedge \mathbf{emp}) \ * \ r'\} \\
\qquad \left.\begin{array}{l} \{(v = v' \wedge \mathbf{emp})\} \\ v := \mathbf{cons}(\overline{e}) \\ \{(v \mapsto \overline{e}')\} \end{array}\right\} \ * \ r' \\
\quad \{(v \mapsto \overline{e}') \ * \ r'\}
\end{array}\right\} \exists v' \\
\{\exists v'.\ (v \mapsto \overline{e}') \ * \ r'\}
\end{array}
$$

The backward-reasoning form (CONSBR)

$$\frac{}{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\},}$$

where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \to v''$.

The universal quantifier ∀v'' in the precondition expresses the nondeterminism of allocation

The most direct way to see this is by a semantic proof that the rule is sound

Soundness of the Backward Reasoning Rule (CONSBR)

$$\frac{}{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\},}$$

where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \to v''$.

Suppose the precondition holds in the state s, h, i.e., that

$$s, h \vDash \forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''.$$

Then the semantics of universal quantification gives

$$\forall \ell.\ [\, s \mid v'' \colon \ell \,], h \vDash (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p'',$$

and the semantics of separating implication gives

$$\forall \ell, h'.\ h \perp h' \text{ and } \underline{[\, s \mid v'' \colon \ell \,], h' \vDash (v'' \mapsto \overline{e})} \text{ implies } [\, s \mid v'' \colon \ell \,], h \cdot h' \vDash p'',$$

where the underlined formula is equivalent to

$$h' = [\, \ell \colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\mathsf{exp}} s \,].$$

The Soundness of (CONSBR) (continued)

Thus

$$\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \mathrm{dom}\, h \text{ implies}$$
$$[\, s \mid v'' \colon \ell \,], [\, h \mid \ell \colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\mathsf{exp}} s \,] \vDash p'' \big).$$

Then, by the substitution law for assertions, since p'' denotes $p/v \to v''$, we have

$$[\, s \mid v'' \colon \ell \,], h \cdot h' \vDash p'' \text{ iff } \widehat{s}, h \cdot h' \vDash p,$$

where

$$\widehat{s} = [\, s \mid v'' \colon \ell \mid v \colon [\![v'']\!]_{\mathsf{exp}} [\, s \mid v'' \colon \ell \,] \,] = [\, s \mid v'' \colon \ell \mid v \colon \ell \,].$$

Moreover, since v'' does not occur free in p, we can simplify $\widehat{s}, h \cdot h' \vDash p$ to $[\, s \mid v \colon \ell \,], h \cdot h' \vDash p$. Thus

$$\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \mathrm{dom}\, h \text{ implies}$$
$$[\, s \mid v \colon \ell \,], [\, h \mid \ell \colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\mathsf{exp}} s \,] \vDash p \big).$$

The Soundness of (CONSBR) (continued)

$$\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \mathsf{dom}\, h \text{ implies}$$
$$[\, s \mid v\colon \ell \,], [\, h \mid \ell\colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\mathsf{exp}} s \,] \models p\big).$$

Now execution of the allocation command $v := \mathbf{cons}(\overline{e})$, starting in the state s, h, will never abort, and will always termininate in a state

$$[\, s \mid v\colon \ell \,], [\, h \mid \ell\colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\mathsf{exp}} s \,]$$

for some ℓ such that $\ell, \ldots, \ell + n - 1 \notin \mathsf{dom}\, h$. Thus the condition displayed above insures that all possible terminating states satisfy the postcondition p. □

From (CONSG):

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \rightarrow v'$, and r' denotes $r/v \rightarrow v'$.

we derive (CONSBR):

$$\frac{}{\{\forall v''.\ (v'' \mapsto \overline{e})\ -\!\!*\ p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\}},$$

where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \rightarrow v''$.

$\{\forall v''.\ (v'' \mapsto \overline{e})\ -\!\!*\ p''\}$
$v := \mathbf{cons}(\overline{e})$
$\{\exists v'.\ (v \mapsto \overline{e}')\ *\ (\forall v''.\ (v'' \mapsto \overline{e}')\ -\!\!*\ p'')\}$
$\{\exists v'.\ (v \mapsto \overline{e}')\ *\ ((v \mapsto \overline{e}')\ -\!\!*\ p)\}$
$\{\exists v'.\ p\}$ $\qquad q\ *\ (q\ -\!\!*\ p) \Rightarrow p$
$\{p\}.$

From (CONSBR):

$$\overline{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\}},$$

where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \rightarrow v''$.

we derive (CONSG):

$$\overline{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \rightarrow v'$, and r' denotes $r/v \rightarrow v'$.

$\{r\}$
$\{\forall v''.\ r\}$
$\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} ((v'' \mapsto \overline{e})\ *\ r)\}$ $\qquad r \Rightarrow (q \mathrel{-\!\!*} (q\ *\ r))$
$\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (((v'' \mapsto \overline{e}')\ *\ r')/v' \rightarrow v)\}$
$\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (\exists v'.\ (v'' \mapsto \overline{e}')\ *\ r')\}$
$v := \mathbf{cons}(\overline{e})$
$\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}.$

Inference Rules for Nonoverwriting Lookup

- The local nonoverwriting form (LKNOL)

$$\frac{}{\{e \mapsto v''\}\ v := [e]\ \{v = v'' \wedge (e \mapsto v)\}},$$

  where $v \notin \mathsf{FV}(e)$.

- The global nonoverwriting form (LKNOG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\ v := [e]\ \{(e \mapsto v)\ *\ p\}},$$

  where $v \notin \mathsf{FV}(e)$, $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \rightarrow v''$.

In (LKNOG):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\ v := [e]\ \{(e \mapsto v)\ *\ p\},}$$

where $v \notin \mathsf{FV}(e)$, $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \rightharpoonup v''$.

there is no restriction preventing v'' from being the same variable as v. Thus, as a special case,

$$\frac{}{\{\exists v.\ (e \mapsto v)\ *\ p\}\ v := [e]\ \{(e \mapsto v)\ *\ p\},}$$

where $v \notin \mathsf{FV}(e)$. For example, if we take

v to be j
e to be $\mathsf{i} + 1$
p to be $\mathsf{i} \mapsto 3\ *\ \mathbf{list}\ \alpha\ \mathsf{j}$,

(and remember $\mathsf{i} \mapsto 3, \mathsf{j}$ abbreviates $(\mathsf{i} \mapsto 3) * (\mathsf{i} + 1 \mapsto \mathsf{j})$), then we obtain the instance

$$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ \mathsf{j}\}\ \mathsf{j} := [\mathsf{i} + 1]\ \{\mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ \mathsf{j}\}.$$

Inference Rules for General Lookup

- The local form (LKL)

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

where v, v', and v'' are distinct, and e' denotes $e/v \rightarrow v'$.

- The global form (LKG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \rightarrow v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \rightarrow v)\},}$$

where v, v', and v'' are distinct, $v', v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \rightarrow v'$.

Inference Rules for General Lookup

- The first backward-reasoning form (LKBR1)

$$\frac{}{\{\exists v''.\ (e \mapsto v'') \ * \ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\},}$$

  where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

- The second backward-reasoning form (LKBR2)

$$\frac{}{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\},}$$

  where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

From (LKL):

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

where v, v', and v'' are distinct, and e' denotes $e/v \rightarrow v'$.

we derive (LKG):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \rightarrow v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \rightarrow v)\},}$$

where v, v', and v'' are distinct, $v', v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \rightarrow v'$.

$$\begin{array}{l}
\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \rightarrow v)\} \\
\{\exists v', v''.\ (v = v' \wedge (e \mapsto v''))\ *\ (r/v' \rightarrow v)\} \\
\quad \left.\begin{array}{l}
\{(v = v' \wedge (e \mapsto v''))\ *\ (r/v' \rightarrow v)\} \\
\quad \left.\begin{array}{l}
\{v = v' \wedge (e \mapsto v'')\} \\
v := [e] \\
\{v = v'' \wedge (e' \mapsto v)\}
\end{array}\right\}\ *\ r \\
\{(v = v'' \wedge (e' \mapsto v))\ *\ (r/v'' \rightarrow v)\}
\end{array}\right\}\ \exists v', v'' \\
\{\exists v', v''.\ (v = v'' \wedge (e' \mapsto v))\ *\ (r/v'' \rightarrow v)\} \\
\{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \rightarrow v)\}.
\end{array}$$

An Instance of (LKG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'') \ *\ (r/v' \to v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v) \ *\ (r/v'' \to v)\},}$$

where v, v', and v'' are distinct, $v', v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \to v'$.

As an example of an instance, if we take

v to be j
v' to be m
v'' to be k

e to be $\mathsf{j}+1$
r to be $\mathsf{i}+1 \mapsto \mathsf{m} \ *\ \mathsf{k}+1 \mapsto \mathbf{nil}$,

then we obtain (using the commutivity of $*$)

$$\{\exists \mathsf{k}.\ \mathsf{i}+1 \mapsto \mathsf{j} \ *\ \mathsf{j}+1 \mapsto \mathsf{k} \ *\ \mathsf{k}+1 \mapsto \mathbf{nil}\}$$
$$\mathsf{j} := [\mathsf{j}+1]$$
$$\{\exists \mathsf{m}.\ \mathsf{i}+1 \mapsto \mathsf{m} \ *\ \mathsf{m}+1 \mapsto \mathsf{j} \ *\ \mathsf{j}+1 \mapsto \mathbf{nil}\}.$$

From (LKG):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \to v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \to v)\}}$$

where v, v', and v'' are distinct, $v', v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \to v'$.

we derive (LKL):

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

where v, v', and v'' are distinct, and e' denotes $e/v \to v'$.

$\{v = v' \wedge (e \mapsto v'')\}$

$\{\exists \widehat{v}''.\ v = v' \wedge \widehat{v}'' = v'' \wedge (e \mapsto \widehat{v}'')\}$

$\{\exists \widehat{v}''.\ (e \mapsto \widehat{v}'')\ *\ (v = v' \wedge \widehat{v}'' = v'' \wedge \mathbf{emp})\}$

$\{\exists \widehat{v}''.\ (e \mapsto \widehat{v}'')\ *\ ((\widehat{v}' = v' \wedge \widehat{v}'' = v'' \wedge \mathbf{emp})/\widehat{v}' \to v)\}$

$v := [e]$

$\{\exists \widehat{v}'.\ (\widehat{e}' \mapsto v)\ *\ ((\widehat{v}' = v' \wedge \widehat{v}'' = v'' \wedge \mathbf{emp})/\widehat{v}'' \to v)\}$

$\{\exists \widehat{v}'.\ (\widehat{e}' \mapsto v)\ *\ (\widehat{v}' = v' \wedge v = v'' \wedge \mathbf{emp})\}$

$\{\exists \widehat{v}'.\ \widehat{v}' = v' \wedge v = v'' \wedge (\widehat{e}' \mapsto v)\}$ (where $\widehat{e}'$ denotes $e/v \to \widehat{v}'$)

$\{v = v'' \wedge (e' \mapsto v)\}$

From (LKG):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \to v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \to v)\},}$$

where v, v', and v'' are distinct, v', $v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \to v'$.

we derive (LKBR1):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\},}$$

where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

$\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}$
$v := [e]$
$\{\exists v'.\ (e' \mapsto v)\ *\ ((e' \mapsto v) \mathrel{-\!\!*} p)\}$
$\{\exists v'.\ p\}$
$\{p\}.$

$q\ *\ (q \mathrel{-\!\!*} p) \Rightarrow p$

From (LKBR1):

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\},}$$

where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

we derive (LKBR2):

$$\frac{}{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\},}$$

where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

$$(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e')\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)$$

$\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}$
$\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}$
$v := [e]$
$\{p\}.$

Example: Gluing records

$$
\begin{array}{l}
\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\textbf{if } \mathsf{y} = \mathsf{x} + 1 \textbf{ then} \\
\quad \{\mathsf{x} \mapsto -, -\} \\
\quad \textbf{skip} \\
\textbf{else if } \mathsf{x} = \mathsf{y} + 1 \textbf{ then} \\
\quad \{\mathsf{y} \mapsto -, -\} \\
\quad \mathsf{x} := \mathsf{y} \\
\textbf{else} \\
\quad \big(\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\quad \textbf{dispose } \mathsf{x} \ ; \\
\quad \{\mathsf{y} \mapsto -\} \\
\quad \textbf{dispose } \mathsf{y} \ ; \\
\quad \{\textbf{emp}\} \\
\quad \mathsf{x} := \textbf{cons}(1, 2)\big) \\
\{\mathsf{x} \mapsto -, -\}.
\end{array}
$$

Another Example: Relative Pointers

$$\{\mathbf{emp}\}$$

$$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a})\ ; \qquad \text{(CONSNOL)}$$

$$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\}$$

$$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b})\ ; \qquad \text{(CONSNOG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, -) \ * \ (\mathsf{y} \mapsto \mathsf{b}, -)\} \qquad (p/v \rightarrow e \Rightarrow \exists v.\ p)$$

$$[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x}\ ; \qquad \text{(MUG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

$$[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y}\ ; \qquad \text{(MUG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{x} + (\mathsf{y} - \mathsf{x}) \mapsto \mathsf{b}, -(\mathsf{y} - \mathsf{x}))\}$$

$$\{\exists \mathsf{o}.\ (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \ * \ (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b}, \ -\mathsf{o})\}$$

Array Allocation

$$\langle \text{comm} \rangle ::= \cdots \mid \langle \text{var} \rangle := \mathbf{allocate}\ \langle \text{exp} \rangle$$

$\text{x} := \mathbf{allocate}\ \text{y}$

Store : x: 3, y: 4
Heap : empty
$\Downarrow$
Store : x: 37, y: 4
Heap : 37: −, 38: −, 39: −, 40: −

Iterated Separating Conjunction

$$\langle \text{assert} \rangle ::= \cdots \mid \bigodot_{\langle \text{var} \rangle = \langle \text{exp} \rangle}^{\langle \text{exp} \rangle} \langle \text{assert} \rangle$$

Let I be the contiguous set

$$I = \{ v \mid e \leq v \leq e' \}$$

of integers between the values of e and e'. Then $\bigodot_{v=e}^{e'} p(v)$ is true iff the heap can be partitioned into a family of disjoint subheaps, indexed by I, such that $p(v)$ is true for the vth subheap.

An Inference Rule

$$\frac{}{\{r\}\ v := \mathbf{allocate}\ e\ \{(\bigodot_{i=v}^{v+e-1} i \mapsto -)\ *\ r\},}$$

where v does not occur free in r or e.

Summary

- Non-faulting semantics of Hoare triples
- Local reasoning: just fragment of heap (footprint)

Summary

The Frame Rule (O'Hearn) (FR)

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

where no variable occurring free in r is modified by c.

Proposition 11 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

Summary

$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}}\ \text{(MUL)}$$

$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}\ \text{(DISL)}$$

$$\frac{}{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\},}\ \text{(CONSL)}$$

where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}\ \text{(LKL)}$$

where v, v', and v'' are distinct, and e' denotes $e/v \to v'$.

4. Lists and List Segments

Notation for Sequences

When α and β are sequences, we write

- ϵ for the empty sequence.
- $[a]$ for the single-element sequence containing a. (We will omit the brackets when a is not a sequence.)
- $\alpha \cdot \beta$ for the composition of α followed by β.
- $\alpha^\dagger$ for the reflection of α.
- $\#\alpha$ for the length of α.
- α_i for the ith component of α.

Some Laws for Sequences

$$\alpha \cdot \epsilon = \alpha \qquad \epsilon \cdot \alpha = \alpha \qquad (\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$$

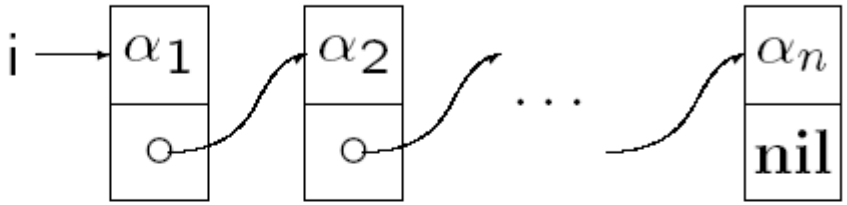
$$\epsilon^\dagger = \epsilon \qquad [a]^\dagger = [a] \qquad (\alpha \cdot \beta)^\dagger = \beta^\dagger \cdot \alpha^\dagger$$

$$\#\epsilon = 0 \qquad \#[a] = 1 \qquad \#(\alpha \cdot \beta) = (\#\alpha) + (\#\beta)$$

$$\alpha = \epsilon \vee \exists a, \alpha'.\ \alpha = [a] \cdot \alpha' \qquad \alpha = \epsilon \vee \exists \alpha', a.\ \alpha = \alpha' \cdot [a].$$

Singly-linked Lists

list α i:



is defined by

$$\text{list } \epsilon \text{ i} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \text{i} = \mathbf{nil}$$

$$\text{list } (\text{a}{\cdot}\alpha) \text{ i} \stackrel{\text{def}}{=} \exists \text{j}.\ \text{i} \mapsto \text{a}, \text{j} \ * \ \text{list } \alpha \text{ j},$$

where

- ϵ is the empty sequence.
- $\alpha{\cdot}\beta$ is the concatenation of α followed by β.

$$\text{list } \alpha \text{ i} \Rightarrow (\text{i} = \mathbf{nil} \Leftrightarrow \alpha = \epsilon)$$

List Reversal

$$LREV \stackrel{\text{def}}{=} \mathsf{j} := \mathbf{nil};$$
$$\quad \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k})$$

$$\{\text{list}\ \alpha_0\ \mathsf{i}\}\ LREV\ \{\text{list}\ \alpha_0^{\dagger}\ \mathsf{j}\}$$

Loop invariant:

$$\exists \alpha, \beta.\ (\text{list}\ \alpha\ \mathsf{i}\ *\ \text{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta$$

$$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\}$$

$$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathbf{nil} = \mathbf{nil})\}$$

$$\mathsf{j} := \mathbf{nil}\ ;$$

$$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathsf{j} = \mathbf{nil})\}$$

$$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ \mathsf{list}\ \epsilon\ \mathsf{j}\}$$

$$\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta\}$$

$\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$

$\big($ $\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ (\mathsf{a}\cdot\alpha)\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\mathsf{k} := [\mathsf{i} + 1]\ ;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$[\mathsf{i} + 1] := \mathsf{j}\ ;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ (\mathsf{a}\cdot\beta)\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \mathsf{a} \cdot \beta\}$

$\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}$

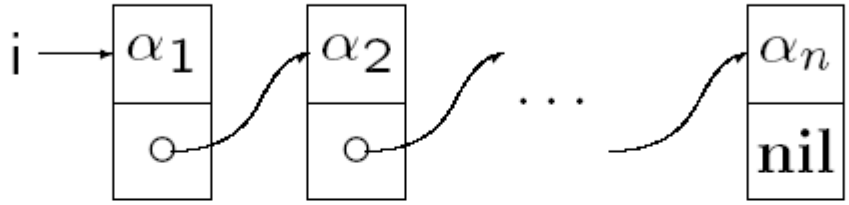
$\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}\big)$

$\{\exists \alpha, \beta.\ \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon\}$

$\{\mathsf{list}\ \alpha_0^\dagger\ \mathsf{j}\}$

Singly-linked Lists

list α i:



is defined by induction on the length of the sequence α (i.e., by structural induction on α):

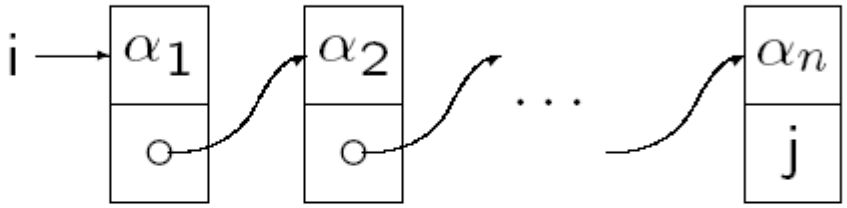
$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\mathsf{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

$$\mathsf{list}\ (\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \stackrel{\mathsf{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}.$$

This was sufficient for specifying and proving a program for reversing a list, but for many programs that deal with lists, it is necessary to reason about parts of lists that we will call list "segments".

Singly-linked List Segments

$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})$:



is defined by

$$\mathsf{lseg}\ \epsilon\ (\mathsf{i}, \mathsf{j}) \stackrel{\mathsf{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{k}) \stackrel{\mathsf{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j}, \mathsf{k}).$$

Properties

$$\begin{aligned}
\mathsf{lseg}\ \mathsf{a}\ (\mathsf{i},\mathsf{j}) &\Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j} \\
\mathsf{lseg}\ \alpha{\cdot}\beta\ (\mathsf{i},\mathsf{k}) &\Leftrightarrow \exists \mathsf{j}.\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j}) \ * \ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k}) \\
\mathsf{lseg}\ \alpha{\cdot}\mathsf{b}\ (\mathsf{i},\mathsf{k}) &\Leftrightarrow \exists \mathsf{j}.\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j}) \ * \ \mathsf{j} \mapsto \mathsf{b},\mathsf{k} \\
\mathsf{list}\ \alpha\ \mathsf{i} &\Leftrightarrow \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathbf{nil}).
\end{aligned}$$

Proof of the Composition Property

$$\mathsf{lseg}\ \alpha{\cdot}\beta\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k})$$

The proof is by induction on the length of α.

When α is empty:

$$\begin{aligned}
&\exists \mathsf{j}.\ \mathsf{lseg}\ \epsilon\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k}) \\
&\quad \Leftrightarrow \exists \mathsf{j}.\ (\mathbf{emp} \wedge \mathsf{i} = \mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k}) \\
&\quad \Leftrightarrow \exists \mathsf{j}.\ (\mathbf{emp}\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k})) \wedge \mathsf{i} = \mathsf{j} \\
&\quad \Leftrightarrow \exists \mathsf{j}.\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k}) \wedge \mathsf{i} = \mathsf{j} \\
&\quad \Leftrightarrow \mathsf{lseg}\ \beta\ (\mathsf{i},\mathsf{k}) \\
&\quad \Leftrightarrow \mathsf{lseg}\ \epsilon{\cdot}\beta\ (\mathsf{i},\mathsf{k})
\end{aligned}$$

Proof of the Composition Property

$$\mathsf{lseg}\ \alpha{\cdot}\beta\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k})$$

The proof is by induction on the length of α.

When $\alpha = \mathsf{a}{\cdot}\alpha'$:

$\exists \mathsf{j}.\ \mathsf{lseg}\ \mathsf{a}{\cdot}\alpha'\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k})$

$\Leftrightarrow \exists \mathsf{j},\mathsf{l}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{lseg}\ \alpha'\ (\mathsf{l},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j},\mathsf{k})$

$\Leftrightarrow \exists \mathsf{l}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{lseg}\ \alpha'{\cdot}\beta\ (\mathsf{l},\mathsf{k})$ (induction hypothesis)

$\Leftrightarrow \mathsf{lseg}\ \mathsf{a}{\cdot}\alpha'{\cdot}\beta\ (\mathsf{i},\mathsf{k})$

Emptyness Conditions

For lists, one can derive a law that shows clearly when a list represents the empty sequence:

$$\mathsf{list}\ \alpha\ \mathsf{i} \Rightarrow (\mathsf{i} = \mathbf{nil} \Leftrightarrow \alpha = \epsilon).$$

For list segments, however, the situation is more complex. One can derive

$$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \Rightarrow (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil}))$$
$$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon).$$

But these formulas do not say whether α is empty when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$.

Nontouching List Segments

When

$$\mathsf{lseg}\, a_1 \cdots a_n\, (\mathsf{i}_0, \mathsf{i}_n),$$

we have

$$\exists \mathsf{i}_1, \ldots \mathsf{i}_{n-1}.\ (\mathsf{i}_0 \mapsto a_1, \mathsf{i}_1) \ * \ (\mathsf{i}_1 \mapsto a_2, \mathsf{i}_2) \ * \ \cdots \ * \ (\mathsf{i}_{n-1} \mapsto a_n, \mathsf{i}_n).$$

Thus $\mathsf{i}_0, \ldots, \mathsf{i}_{n-1}$ are distinct, but i_n is not constrained, and may equal any of the $\mathsf{i}_0, \ldots, \mathsf{i}_{n-1}$. In this case, we say that the list segment is *touching*.

list segments
- nontouching
- cyclic
  - touching
  - overlapping (forbidden by $*$).

Nontouching List Segments (continued)

We can define nontouching list segments inductively by:

$$\mathsf{ntlseg}\ \epsilon\ (\mathsf{i},\mathsf{j}) \stackrel{\mathsf{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{ntlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k}) \stackrel{\mathsf{def}}{=} \mathsf{i} \neq \mathsf{k} \wedge \mathsf{i}{+}1 \neq \mathsf{k} \wedge (\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j} * \mathsf{ntlseg}\ \alpha\ (\mathsf{j},\mathsf{k})),$$

or equivalently, we can define them in terms of lseg:

$$\mathsf{ntlseg}\ \alpha\ (\mathsf{i},\mathsf{j}) \stackrel{\mathsf{def}}{=} \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j}) \wedge \neg\, \mathsf{j} \hookrightarrow -.$$

The obvious advantage of knowing that a list segment is nontouching is that it is easy to test whether it is empty:

$$\mathsf{ntlseg}\ \alpha\ (\mathsf{i},\mathsf{j}) \Rightarrow (\alpha = \epsilon \Leftrightarrow \mathsf{i} = \mathsf{j}).$$

Fortunately, there are common situations where list segments must be nontouching:

$$\begin{aligned}
\mathsf{list}\ \alpha\ \mathsf{i} &\Rightarrow \mathsf{ntlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}) \\
\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \ *\ \mathsf{list}\ \beta\ \mathsf{j} &\Rightarrow \mathsf{ntlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \ *\ \mathsf{list}\ \beta\ \mathsf{j} \\
\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \ *\ \mathsf{j} \hookrightarrow - &\Rightarrow \mathsf{ntlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \ *\ \mathsf{j} \hookrightarrow -.
\end{aligned}$$

Preciseness of List Assertions

The assertions

$$\mathsf{list}\ \alpha\ \mathsf{i} \qquad \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \qquad \mathsf{ntlseq}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

are all precise.. On the other hand, although

$$\exists \alpha.\ \mathsf{list}\ \alpha\ \mathsf{i} \qquad \exists \alpha.\ \mathsf{ntlseq}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

are precise,

$$\exists \alpha.\ \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

is not precise.

Proposition 12 (1) $\exists\alpha.$ list α i *is a precise assertion.* (2) list α i *is a precise assertion.*

PROOF (1) We begin with two preliminary properties of the list predicate:

(a) Suppose $[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \vDash \mathsf{list}\ \alpha\ \mathsf{i}$. Then

$$[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \vDash \mathsf{list}\ \alpha\ \mathsf{i} \wedge \alpha = \epsilon$$
$$[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \vDash \mathsf{list}\ \epsilon\ \mathsf{i}$$
$$[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \vDash \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil},$$

so that h is the empty heap and $i = \mathbf{nil}$.

Proposition 12 (1) $\exists\alpha.\ \mathsf{list}\ \alpha\ \mathsf{i}$ *is a precise assertion.* (2) $\mathsf{list}\ \alpha\ \mathsf{i}$ *is a precise assertion.*

PROOF (1) We begin with two preliminary properties of the list predicate:

(b) On the other hand, suppose $[\mathsf{i}\colon i \mid \alpha\colon a{\cdot}\alpha']\,, h \vDash \mathsf{list}\ \alpha\ \mathsf{i}$. Then

$$
\begin{aligned}
[\mathsf{i}\colon i \mid \alpha\colon a{\cdot}\alpha' \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha']\,, h &\vDash \mathsf{list}\ \alpha\ \mathsf{i} \wedge \alpha = \mathsf{a}{\cdot}\alpha' \\
[\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha']\,, h &\vDash \mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i} \\
[\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha']\,, h &\vDash \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ *\ \mathsf{list}\ \alpha'\ \mathsf{j} \\
\exists j.\ [\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \mathsf{j}\colon j \mid \alpha'\colon \alpha']\,, h &\vDash \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ *\ \mathsf{list}\ \alpha'\ \mathsf{j},
\end{aligned}
$$

so that there are j and h' such that

$$i \neq \mathbf{nil} \qquad h = [i\colon a \mid i+1\colon j]\cdot h' \qquad [\mathsf{j}\colon j \mid \alpha'\colon \alpha']\,, h' \vDash \mathsf{list}\ \alpha'\ \mathsf{j},$$

and by the substitution theorem,

$$[\mathsf{i}\colon j \mid \alpha\colon \alpha']\,, h' \vDash \mathsf{list}\ \alpha\ \mathsf{i}.$$

To prove (1), we assume s, h, h_0, and h_1 are such that h_0, $h_1 \subseteq h$ and

$$s, h_0 \vDash \exists \alpha.\ \mathsf{list}\ \alpha\ \mathsf{i} \qquad\qquad s, h_1 \vDash \exists \alpha.\ \mathsf{list}\ \alpha\ \mathsf{i}.$$

We must show that $h_0 = h_1$.

Since i is the only free variable of the above assertion, we can assume s is $[\,\mathsf{i}\colon i\,]$ for some i. Then we can use the semantic equation for the existential quantifier to show that there are sequences α_0 and α_1 such that

$$[\,\mathsf{i}\colon i \mid \alpha\colon \alpha_0\,], h_0 \vDash \mathsf{list}\ \alpha\ \mathsf{i} \qquad\qquad [\,\mathsf{i}\colon i \mid \alpha\colon \alpha_1\,], h_1 \vDash \mathsf{list}\ \alpha\ \mathsf{i}.$$

We will complete our proof by showing, by structural induction on α_0:

> For all α_0, α_1, i, h, h_0, and h_1, if h_0, $h_1 \subseteq h$ and the statements displayed above hold, then $h_0 = h_1$.

For the base case, suppose α_0 is empty. Then by (a), h_0 is the empty heap and $i = \mathbf{nil}$.

Moreover, if α_1 were not empty, then by (b) we would have the contradiction $i \neq \mathbf{nil}$. Thus α_1 must be empty, so by (a), h_1 is the empty heap, so that $h_0 = h_1$.

For the induction step suppose $\alpha_0 = a_0 \cdot \alpha'_0$. Then by (b), there are j_0 and h'_0 such that

$$i \neq \mathbf{nil}, \quad h_0 = [\, i\colon a_0 \mid i{+}1\colon j_0 \,] \cdot h'_0, \quad [\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha'_0\,], h'_0 \vDash \mathsf{list}\ \alpha\ \mathsf{i}.$$

Moreover, if α_1 were empty, then by (a) we would have the contradiction $i = \mathbf{nil}$. Thus α_1 must be $a_1 \cdot \alpha'_1$ for some a_1 and α'_1. Then by (b), there are j_1 and h'_1 such that

$$i \neq \mathbf{nil}, \quad h_1 = [\, i\colon a_1 \mid i{+}1\colon j_1 \,] \cdot h'_1, \quad [\,\mathsf{i}\colon j_1 \mid \alpha\colon \alpha'_1\,], h'_1 \vDash \mathsf{list}\ \alpha\ \mathsf{i}.$$

Since h_0 and h_1 are both subsets of h, they must map i and $i{+}1$ into the same value. Thus $[\, i\colon a_0 \mid i{+}1\colon j_0 \,] = [\, i\colon a_1 \mid i{+}1\colon j_1 \,]$, so that $a_0 = a_1$ and $j_0 = j_1$. Then, since

$$[\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha'_0\,], h'_0 \vDash \mathsf{list}\ \alpha\ \mathsf{i} \quad \text{and} \quad [\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha'_1\,], h'_1 \vDash \mathsf{list}\ \alpha\ \mathsf{i},$$

the induction hypothesis give $h'_0 = h'_1$. It follows that $h_0 = h_1$.

Proposition 12 (1) $\exists\alpha.$ list α i *is a precise assertion.* (2) list α i *is a precise assertion.*

(2) We use the law that p is precise whenever $p \Rightarrow q$ is valid and q is precise. Then, since $\mathbf{list}\ \alpha$ i $\Rightarrow \exists\alpha.\ \mathbf{list}\ \alpha$ i is valid, $\mathbf{list}\ \alpha$ i is precise. END OF PROOF

Example: Insertion at the Head

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\}$

$\mathsf{k} := \mathbf{cons}(\mathsf{a}, \mathsf{i})$; (CONSNOG)

$\{\mathsf{k} \mapsto \mathsf{a}, \mathsf{i} \ * \ \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\}$

$\{\exists \mathsf{i}.\ \mathsf{k} \mapsto \mathsf{a}, \mathsf{i} \ * \ \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\}$

$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{k}, \mathsf{j})\}$

$\mathsf{i} := \mathsf{k}$ (AS)

$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{j})\},$

or, more concisely:

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{k})\}$$
$$\mathsf{i} := \mathbf{cons}(\mathsf{a}, \mathsf{i})\ ; \qquad \text{(CONSG)}$$
$$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j}, \mathsf{k})\}$$
$$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{k})\}.$$

Example: Insertion at the End

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{k}\}$$

$$\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})\ ; \qquad \text{(CONSNOG)}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a}\ *\ \mathsf{j}+1\mapsto \mathsf{k}\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a}\ *\ \mathsf{j}+1\mapsto -\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$[\mathsf{j}+1] := \mathsf{l} \qquad \text{(MUG)}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a}\ *\ \mathsf{j}+1\mapsto \mathsf{l}\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\ (\mathsf{i},\mathsf{l})\ *\ \mathsf{l}\mapsto \mathsf{b},\mathsf{k}\}$$

$$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}{\cdot}\mathsf{b}\ (\mathsf{i},\mathsf{k})\}.$$

Example: Deletion at the Head

$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k})\}$

$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k})\}$

$\{\exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k}))\}$

$\mathsf{j} := [\mathsf{i}+1]\ ;$ (LKNOG)

$\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k}))\}$

$\{\mathsf{i} \mapsto \mathsf{a}\ *\ (\mathsf{i}+1 \mapsto \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k}))\}$

$\mathbf{dispose}\ \mathsf{i}\ ;$ (DISG)

$\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k})\}$

$\mathbf{dispose}\ \mathsf{i}+1\ ;$ (DISG)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k})\}$

$\mathsf{i} := \mathsf{j}$ (AS)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{k})\}.$

Example: Deletion at the End

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}\}$

$[\mathsf{j} + 1] := \mathsf{l}\ ;$ (MUG)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}\}$

$\mathbf{dispose}\ \mathsf{k}\ ;$ (DISG)

$\mathbf{dispose}\ \mathsf{k} + 1$ (DISG)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}\}$

$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\ (\mathsf{i}, \mathsf{l})\}.$

Example: List Reversal

$$LREV \stackrel{\text{def}}{=} \mathsf{j} := \mathbf{nil};$$
$$\quad \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k})$$

$$\{\text{list}\ \alpha_0\ \mathsf{i}\}\ LREV\ \{\text{list}\ \alpha_0^{\dagger}\ \mathsf{j}\}$$

Loop invariant:

$$\exists \alpha, \beta.\ (\text{list}\ \alpha\ \mathsf{i}\ *\ \text{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta$$

$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\}$

$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathbf{nil} = \mathbf{nil})\}$

$\mathsf{j} := \mathbf{nil}\ ;$

$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathsf{j} = \mathbf{nil})\}$

$\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ \mathsf{list}\ \epsilon\ \mathsf{j}\}$

$\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta\}$

$\{\exists \alpha, \beta.\ (\mathsf{list}\,\alpha\,\mathsf{i} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$

$\big(\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\,(\mathsf{a}\cdot\alpha)\,\mathsf{i} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k} \ast \mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\mathsf{k} := [\mathsf{i} + 1]\ ;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k} \ast \mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$[\mathsf{i} + 1] := \mathsf{j}\ ;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ast \mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,(\mathsf{a}\cdot\beta)\,\mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \mathsf{a} \cdot \beta\}$

$\{\exists \alpha, \beta.\ (\mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,\beta\,\mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}$

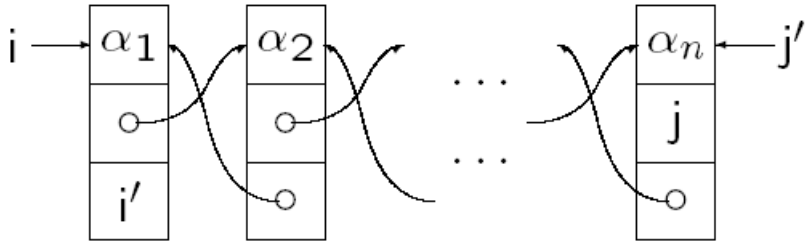
$\{\exists \alpha, \beta.\ (\mathsf{list}\,\alpha\,\mathsf{i} \ast \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}\big)$

$\{\exists \alpha, \beta.\ \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon\}$

$\{\mathsf{list}\,\alpha_0^\dagger\,\mathsf{j}\}$

Doubly-Linked List Segments

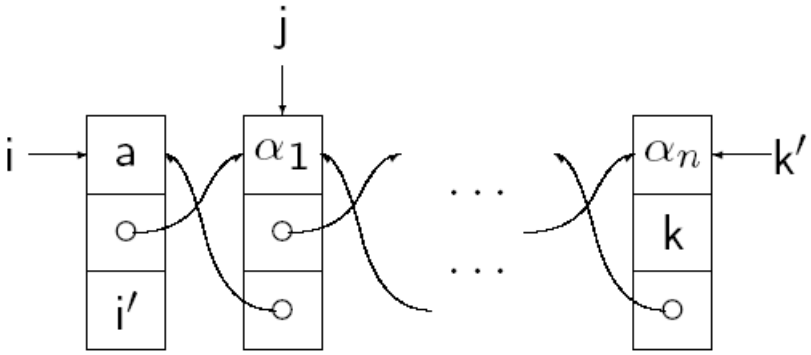
dlseg α (i, i′, j, j′):



is defined by

$$\text{dlseg } \epsilon \text{ (i, i′, j, j′)} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \text{i} = \text{j} \wedge \text{i′} = \text{j′}$$

$$\text{dlseg a}\cdot\alpha \text{ (i, i′, k, k′)} \stackrel{\text{def}}{=} \exists \text{j}.\ \text{i} \mapsto \text{a, j, i′} \ * \ \text{dlseg } \alpha \text{ (j, i, k, k′)}.$$



Properties

$$\mathsf{dlseg}\ \mathsf{a}\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Leftrightarrow \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}, \mathsf{i}' \wedge \mathsf{i} = \mathsf{j}'$$

$$\mathsf{dlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \Leftrightarrow \exists \mathsf{j}, \mathsf{j}'.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') * \mathsf{dlseg}\ \beta\ (\mathsf{j}, \mathsf{j}', \mathsf{k}, \mathsf{k}')$$

$$\mathsf{dlseg}\ \alpha{\cdot}\mathsf{b}\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \Leftrightarrow \exists \mathsf{j}'.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{k}', \mathsf{j}') * \mathsf{k}' \mapsto \mathsf{b}, \mathsf{k}, \mathsf{j}'.$$

One can also define a doubly-linked list by

$$\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathsf{j}') = \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{j}').$$

Emptyness Conditions

$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil} \wedge \mathsf{i}' = \mathsf{j}'))$$

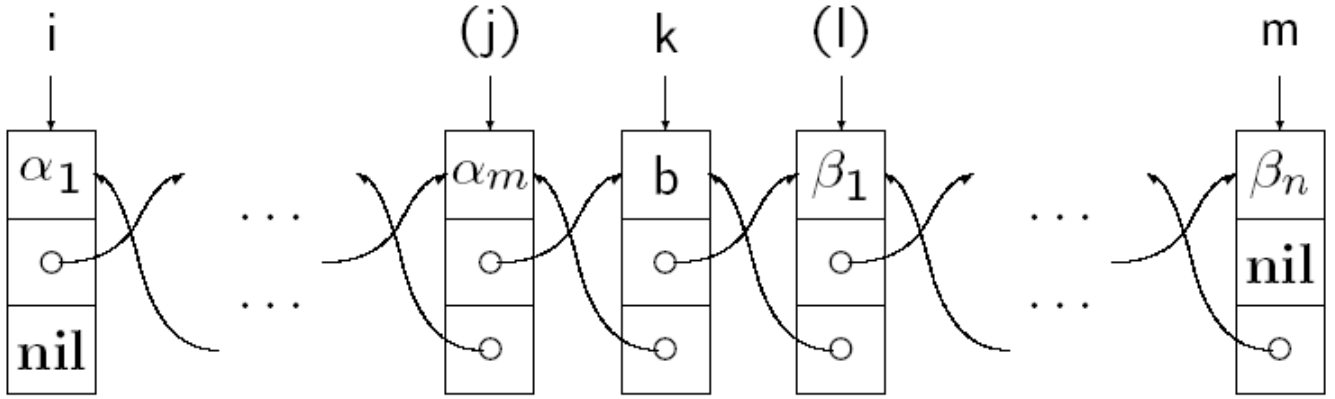
$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{j}' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}))$$

$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon)$$

$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i}' \neq \mathsf{j}' \Rightarrow \alpha \neq \epsilon).$$

(One can also define nontouching segments.)

Deleting an Element from a Doubly-Linked List



$\{\exists \mathsf{j}, \mathsf{l}.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathsf{l} := [\mathsf{k}+1]\ ;\ \mathsf{j} := [\mathsf{k}+2]\ ;$

$\mathbf{dispose}\ \mathsf{k}\ ;\ \mathbf{dispose}\ \mathsf{k}+1\ ;\ \mathbf{dispose}\ \mathsf{k}+2\ ;$

$\mathbf{if}\ \mathsf{j} = \mathbf{nil}\ \mathbf{then}\ \ \mathsf{i} := \mathsf{l}\ \ \ \mathbf{else}\ \ \ [\mathsf{j}+1] := \mathsf{l}\ ;$

$\mathbf{if}\ \mathsf{l} = \mathbf{nil}\ \mathbf{then}\ \ \mathsf{m} := \mathsf{j}\ \ \ \mathbf{else}\ \ [\mathsf{l}+2] := \mathsf{j}$

$\{\mathsf{dlseg}\ \alpha \cdot \beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}$

$\{\exists \mathsf{j}, \mathsf{l}.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathsf{l} := [\mathsf{k} + 1]\ ;\ \mathsf{j} := [\mathsf{k} + 2]\ ;$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{dispose}\ \mathsf{k}\ ;\ \mathbf{dispose}\ \mathsf{k} + 1\ ;\ \mathbf{dispose}\ \mathsf{k} + 2\ ;$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

...

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{if}\ \mathsf{j} = \mathbf{nil}\ \mathbf{then}$

$\quad \{\mathsf{i} = \mathsf{k} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\quad \mathsf{i} := \mathsf{l}$

$\quad \{\mathsf{i} = \mathsf{l} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{else}$

$\quad \{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}, \mathsf{n}$

$\qquad *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$\quad [\mathsf{j} + 1] := \mathsf{l}\ ;$

$\quad \{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}, \mathsf{n}$

$\qquad *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

...

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{if}\ \mathsf{l} = \mathbf{nil}\ \mathbf{then}$

$\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{k} = \mathsf{m} \wedge \beta = \epsilon\}$

$\quad \mathsf{m} := \mathsf{j}$

$\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{j} = \mathsf{m} \wedge \beta = \epsilon\}$

$\mathbf{else}$

$\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{k}$

$\qquad *\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a} \cdot \beta'\}$

$\quad [\mathsf{l} + 2] := \mathsf{j}$

$\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{j}$

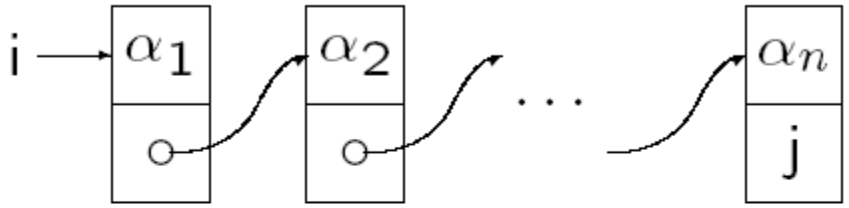
$\qquad *\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a} \cdot \beta'\}$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{j}, \mathbf{nil}, \mathsf{m})\}$

$\{\mathsf{dlseg}\ \alpha \cdot \beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}$

Summary

lseg α (i, j):



is defined by

$$\text{lseg } \epsilon \text{ (i, j)} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \text{i} = \text{j}$$

$$\text{lseg a}\cdot\alpha \text{ (i, k)} \stackrel{\text{def}}{=} \exists \text{j. i} \mapsto \text{a, j} \ * \ \text{lseg } \alpha \text{ (j, k)}.$$

We can also define trees and DAGs.

Summary of Separation Logic

- Programming language
- Assertion language
- Specification language

Current Research and the Future

- Extending separation logic to cover practical language features
  - various concurrency idioms
  - objects

- Building tools to mechanize separation logic
  - much work on shape analysis