Class Notes for EWSCS'06 - March 2006

A SHORT COURSE IN SEPARATION LOGIC

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

March 2006

The Problem — An Example

A program for in-place list reversal:

$\mathsf{j} := \mathbf{nil}\ ;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$

An inadequate invariant:

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

where

$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}.$$

An adequate invariant:

$$\begin{aligned}&(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta)\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}),\end{aligned}$$

where

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

A stronger invariant:

$$\begin{aligned}&(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta)\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})\\ &\quad \wedge \mathsf{list}\ \gamma\ \mathsf{x}\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{x},\mathsf{k})\\ &\qquad\qquad \wedge (\mathbf{reachable}(\mathsf{i},\mathsf{k}) \vee \mathbf{reachable}(\mathsf{j},\mathsf{k})) \Rightarrow \mathsf{k} = \mathbf{nil}).\end{aligned}$$

The Solution

Instead of

$$(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta)$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}),$$

or

$$\begin{aligned}
&(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta)\\
&\quad \wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})\\
&\quad \wedge\ \mathsf{list}\,\gamma\,\mathsf{x}\\
&\quad \wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{x},\mathsf{k})\\
&\qquad\qquad \wedge\ (\mathbf{reachable}(\mathsf{i},\mathsf{k}) \vee \mathbf{reachable}(\mathsf{j},\mathsf{k})) \Rightarrow \mathsf{k} = \mathbf{nil}),
\end{aligned}$$

in separation logic we can write

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

or

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

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

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)
- Information Hiding (O'Hearn, Yang, Reynolds 2003)
- Shared-Variable Concurrency (O'Hearn, Brookes 2004)
- Fractional and Accounting Permissions (Bornat 2005, following Boyland)

States

Without address arithmetic (old version):

$$\text{Values} = \text{Integers} \cup \text{Atoms} \cup \text{Addresses}$$

where Integers, Atoms, and Addresses are disjoint

$$\mathbf{nil} \in \text{Atoms}$$

$$\text{Stores}_V = V \to \text{Values}$$

$$\text{Heaps} = \bigcup_{A \overset{\text{fin}}{\subseteq} \text{Addresses}} (A \to \text{Values}^+)$$

$$\text{States}_V = \text{Stores}_V \times \text{Heaps}$$

where V is a finite set of variables.

With address arithmetic (new version):

$$\text{Values} = \text{Integers}$$

$$\text{Atoms} \cup \text{Addresses} \subseteq \text{Integers}$$

where Atoms and Addresses are disjoint

$$\mathbf{nil} \in \text{Atoms}$$

$$\text{Stores}_V = V \to \text{Values}$$

$$\text{Heaps} = \bigcup_{A \overset{\text{fin}}{\subseteq} \text{Addresses}} (A \to \text{Values})$$

$$\text{States}_V = \text{Stores}_V \times \text{Heaps}$$

where V is a finite set of variables.

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

The Programming Language: An Informal View

The simple imperative language:

$:=$ **skip**   ;   **if** – **then** – **else** –   **while** – **do** –

plus:

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

Note that:

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

Memory Faults

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

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

Assertions

Standard predicate calculus:

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

plus:

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

More Precisely

When s is a store, h is a heap, and p is an assertion whose free variables all 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:

$$\begin{aligned}
s, h \vDash b &\text{ iff } [\![b]\!]_{\text{boolexp}} s = \mathbf{true}, \\
s, h \vDash \neg p &\text{ iff } s, h \vDash p \text{ is false}, \\
s, h \vDash p_0 \wedge p_1 &\text{ iff } s, h \vDash p_0 \text{ and } s, h \vDash p_1 \\
&\qquad\qquad\qquad\qquad \text{(and similarly for } \vee, \Rightarrow, \Leftrightarrow\text{)}, \\
s, h \vDash \forall v.\, p &\text{ iff } \forall x \in \mathbf{Z}.\ [\, s \mid v\colon x \,], h \vDash p, \\
s, h \vDash \exists v.\, p &\text{ iff } \exists x \in \mathbf{Z}.\ [\, s \mid v\colon x \,], h \vDash p, \\
s, h \vDash \mathbf{emp} &\text{ iff } \operatorname{dom} h = \{\}, \\
s, h \vDash e \mapsto e' &\text{ iff } \operatorname{dom} h = \{[\![e]\!]_{\text{exp}} s\} \text{ and } h([\![e]\!]_{\text{exp}} s) = [\![e']\!]_{\text{exp}} s, \\
s, h \vDash p_0 \ast p_1 &\text{ iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\
&\qquad\qquad\qquad\qquad s, h_0 \vDash p_0 \text{ and } s, h_1 \vDash p_1, \\
s, h \vDash p_0 \mathrel{-\!\!\ast} p_1 &\text{ iff } \forall h'.\ (h' \perp h \text{ and } s, h' \vDash p_0) \text{ implies} \\
&\qquad\qquad\qquad\qquad s, h \cdot h' \vDash p_1.
\end{aligned}$$

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

For Instance

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

Some Abbreviations

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

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

$$e \mapsto e_0, \ldots, e_{n-1} \stackrel{\text{def}}{=} e \mapsto e_0 \ * \ \cdots \ * \ e + n - 1 \mapsto e_{n-1}$$

$$\begin{aligned} e \hookrightarrow e_0, \ldots, e_{n-1} &\stackrel{\text{def}}{=} e \hookrightarrow e_0 \ * \ \cdots \ * \ e + n - 1 \hookrightarrow e_{n-1} \\ &\quad \text{iff } e \mapsto e_0, \ldots, e_{n-1} \ * \ \mathbf{true}. \end{aligned}$$

Examples

$$s, h \vDash \mathsf{x} \mapsto \mathsf{y} \ \text{ iff } \ \mathrm{dom}\, h = \{s\,\mathsf{x}\} \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y}$$

$$s, h \vDash \mathsf{x} \mapsto - \ \text{ iff } \ \mathrm{dom}\, h = \{s\,\mathsf{x}\}$$

$$s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y} \ \text{ iff } \ s\,\mathsf{x} \in \mathrm{dom}\, h \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y}$$

$$s, h \vDash \mathsf{x} \hookrightarrow - \ \text{ iff } \ s\,\mathsf{x} \in \mathrm{dom}\, h$$

$$s, h \vDash \mathsf{x} \mapsto \mathsf{y}, \mathsf{z} \ \text{ iff } \ h = [\, s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z}\,]$$

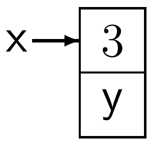
$$s, h \vDash \mathsf{x} \mapsto -, - \ \text{ iff } \ \mathrm{dom}\, h = \{s\,\mathsf{x}, s\,\mathsf{x} + 1\}$$

$$s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{z} \ \text{ iff } \ h \supseteq [\, s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z}\,]$$

$$s, h \vDash \mathsf{x} \hookrightarrow -, - \ \text{ iff } \ \mathrm{dom}\, h \supseteq \{s\,\mathsf{x}, s\,\mathsf{x} + 1\}.$$

Examples of Separating Conjunction

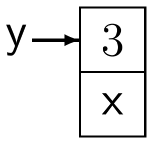
1. $\mathsf{x} \mapsto 3, \mathsf{y}$



   Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

   Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta$

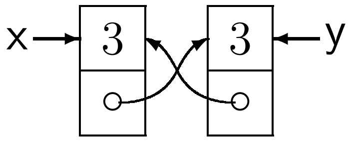
2. $\mathsf{y} \mapsto 3, \mathsf{x}$



   Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

   Heap : $\beta\colon 3,\ \beta{+}1\colon \alpha$

3. $\mathsf{x} \mapsto 3, \mathsf{y} \ * \ \mathsf{y} \mapsto 3, \mathsf{x}$

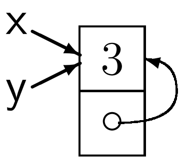


   Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

   Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta,\ \beta\colon 3,\ \beta{+}1\colon \alpha$

   where α, $\alpha + 1$, β, $\beta + 1$ are distinct

4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$



   Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \alpha$

   Heap : $\alpha\colon 3,\ \alpha{+}1\colon \alpha$

5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$

   Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

   Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta,\ \beta\colon 3,\ \beta{+}1\colon \alpha,\ \ldots$

As in (3) or (4), possibly with additional cells

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

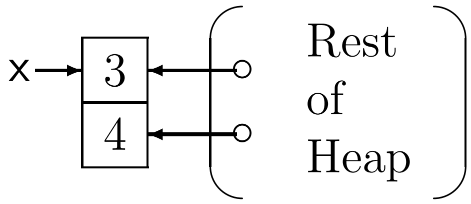
If p is:	then $s, h \vDash p$ iff:
$\mathsf{x} \mapsto 0$	$h = h_0$
$\mathsf{y} \mapsto 1$	$h = h_1$
$\mathsf{x} \mapsto 0 \;*\; \mathsf{y} \mapsto 1$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \;*\; \mathsf{x} \mapsto 0$	**false**
$\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1$	$h = h_0$ or $h = h_1$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

An Example of Separating Implication

Suppose p holds for

Store : $\mathsf{x}\colon \alpha, \ldots$

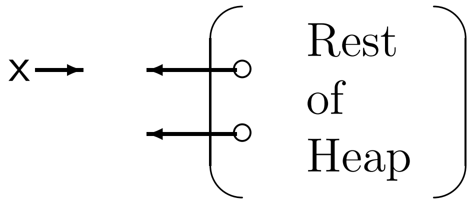
Heap : $\alpha\colon 3, \alpha + 1\colon 4, \text{rest of heap}$



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

Store : $\mathsf{x}\colon \alpha, \ldots$

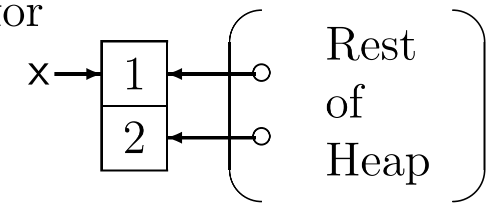
Heap : rest of heap



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

Store : $\mathsf{x}\colon \alpha, \ldots$

Heap : $\alpha\colon 1, \alpha + 1\colon 2, \text{rest of heap}$



Inference

Inference Rules	Instances
$$\frac{p_0 \qquad p_0 \Rightarrow p_1}{p_1}$$	$$\frac{\mathsf{x}+0=\mathsf{x} \qquad \mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0}{\mathsf{x}=\mathsf{x}+0}$$
$$\frac{}{e_2 = e_1 \Rightarrow e_1 = e_2}$$	$$\frac{}{\mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0}$$
$$\mathsf{x}+0=\mathsf{x}$$	$$\mathsf{x}+0=\mathsf{x}$$

A Proof

$$\begin{array}{c} \mathsf{x}+0=\mathsf{x} \\ \mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0 \\ \mathsf{x}=\mathsf{x}+0. \end{array}$$

Notice:

- Metavariables are in italics (or Greek), object variables are in sans serif.
- An inference rule is *sound* iff, for every instance, if the premisses are all valid, then the conclusion is valid.
- For example,

  $$\frac{p}{\forall v.\ p}$$

  is sound, but

  $$\frac{}{p \Rightarrow \forall v.\ p}$$

  is not sound (since, for instance, $\mathsf{x}=0 \Rightarrow \forall \mathsf{x}.\ \mathsf{x}=0$ is not valid).

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

$$
\begin{aligned}
p_0 * p_1 &\Leftrightarrow p_1 * p_0 \\
(p_0 * p_1) * p_2 &\Leftrightarrow p_0 * (p_1 * p_2) \\
p * \mathbf{emp} &\Leftrightarrow p \\
(p_0 \vee p_1) * q &\Leftrightarrow (p_0 * q) \vee (p_1 * q) \\
(p_0 \wedge p_1) * q &\Rightarrow (p_0 * q) \wedge (p_1 * q) \\
(\exists x.\, p_0) * p_1 &\Leftrightarrow \exists x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1 \\
(\forall x.\, p_0) * p_1 &\Rightarrow \forall x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1
\end{aligned}
$$

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

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

Some Axiom Schemata for $\mapsto$

$$
\begin{gathered}
e_0 \mapsto e_0' \wedge e_1 \mapsto e_1' \Leftrightarrow e_0 \mapsto e_0' \wedge e_0 = e_1 \wedge e_0' = e_1' \\
e_0 \hookrightarrow e_0' \ * \ e_1 \hookrightarrow e_1' \Rightarrow e_0 \neq e_1 \\
\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -) \\
(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).
\end{gathered}
$$

To show that the last axiom is sound, is sound, suppose $s, h \vDash (e \hookrightarrow e') \wedge p$. Then:

- $s, h \vDash (e \mapsto e') \ * \ \mathbf{true}$ and $s, h \vDash p$.
- There are heaps h_0 and h_1 such that $h_0 \perp h_1$, $h_0 \cdot h_1 = h$, and $s, h_0 \vDash e \mapsto e'$.
- Then $s, h_0 \vDash e \mapsto e'$ determines $h_0 = [\, [\![e]\!]s \colon [\![e']\!]s \,]$ uniquely.
- To see that $s, h_1 \vDash (e \mapsto e') \mathrel{-\!\!*} p$, let h' be any heap such that $h' \perp h_1$ and $s, h' \vDash e \mapsto e'$. Then $h' = [\, [\![e]\!]s \colon [\![e']\!]s \,] = h_0$, so that $h' \cdot h_1 = h_0 \cdot h_1 = h$, and thus $s, h' \cdot h_1 \vDash p$.
- Then $s, h_0 \cdot h_1 \vDash (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)$, so that $s, h \vDash (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)$.

Pure Assertions

- An assertion is *pure* iff, for any store, it is independent of the heap.
- Syntactically, an assertion is pure if it does not contain **emp**, $\mapsto$, or $\hookrightarrow$.

Axiom Schemata for Purity

$$\begin{array}{rl}
p_0 \wedge p_1 \Rightarrow p_0 \ast p_1 & \text{when } p_0 \text{ or } p_1 \text{ is pure} \\
p_0 \ast p_1 \Rightarrow p_0 \wedge p_1 & \text{when } p_0 \text{ and } p_1 \text{ are pure} \\
(p \wedge q) \ast r \Leftrightarrow (p \ast r) \wedge q & \text{when } q \text{ is pure} \\
(p_0 \mathrel{-\!\!*} p_1) \Rightarrow (p_0 \Rightarrow p_1) & \text{when } p_0 \text{ is pure} \\
(p_0 \Rightarrow p_1) \Rightarrow (p_0 \mathrel{-\!\!*} p_1) & \text{when } p_0 \text{ and } p_1 \text{ are pure.}
\end{array}$$

Two Unsound Axiom Schemata

$$\begin{array}{ll}
p \not\Rightarrow p \ast p & \text{(Contraction)} \\
 & \text{e.g. } p : \mathsf{x} \mapsto 1 \\
p \ast q \not\Rightarrow p & \text{(Weakening)} \\
 & \text{e.g. } p : \mathsf{x} \mapsto 1 \\
 & \quad q : \mathsf{y} \mapsto 2
\end{array}$$

Exercise 1

Give a formal proof of the valid assertion

$$\big((\mathsf{x} \mapsto \mathsf{y} \ * \ \mathsf{x}' \mapsto \mathsf{y}') \ * \ \mathbf{true}\big) \Rightarrow \\ \big(((\mathsf{x} \mapsto \mathsf{y} \ * \ \mathbf{true}) \wedge (\mathsf{x}' \mapsto \mathsf{y}' \ * \ \mathbf{true})) \wedge \mathsf{x} \neq \mathsf{x}'\big)$$

from the rules on page **??**, and (some of) the following inference rules for predicate calculus:

$$p \Rightarrow \mathbf{true} \qquad p \Rightarrow p \qquad p \wedge \mathbf{true} \Rightarrow p$$

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r} \quad \text{(trans impl)}$$

$$\frac{p \Rightarrow q \qquad p \Rightarrow r}{p \Rightarrow q \wedge r} \quad (\wedge\text{-introduction})$$

Your proof will be easier to read if you write it as a sequence of steps rather than a tree. In the inference rules, you should regard $*$ as left associative, e.g.,

$$e_1 \mapsto e_1' \ * \ e_2 \mapsto e_2' \ * \ \mathbf{true} \Rightarrow e_1 \neq e_2$$

stands for

$$(e_1 \mapsto e_1' \ * \ e_2 \mapsto e_2') \ * \ \mathbf{true} \Rightarrow e_1 \neq e_2.$$

For brevity, you may weaken $\Leftrightarrow$ to $\Rightarrow$ when it is the main operator of an axiom. You may also omit instances of the axiom schema $p \Rightarrow p$ when it is used as a premiss of the monotonicity rule.

Exercise 2

None of the following axiom schemata are sound. For each, given an instance which is not valid, along with a description of a state in which the instance is false.

$$p_1 \ * \ p_2 \not\Rightarrow p_1 \wedge p_2$$

$$p_1 \wedge p_2 \not\Rightarrow p_1 \ * \ p_2$$

$$(p_1 \ * \ p_2) \vee q \not\Rightarrow (p_1 \vee q) \ * \ (p_2 \vee q)$$

$$(p_1 \vee q) \ * \ (p_2 \vee q) \not\Rightarrow (p_1 \ * \ p_2) \vee q$$

$$(p_1 \ * \ q) \wedge (p_2 \ * \ q) \not\Rightarrow (p_1 \wedge p_2) \ * \ q$$

$$(p_1 \ * \ p_2) \wedge q \not\Rightarrow (p_1 \wedge q) \ * \ (p_2 \wedge q)$$

$$(p_1 \wedge q) \ * \ (p_2 \wedge q) \not\Rightarrow (p_1 \ * \ p_2) \wedge q$$

$$\Big(\forall x.\ (p_1 \ * \ p_2)\Big) \not\Rightarrow (\forall x.\ p_1) \ * \ p_2 \quad \text{when } x \text{ not free in } p_2$$

$$(p_1 \Rightarrow p_2) \not\Rightarrow \Big((p_1 \ * \ q) \Rightarrow (p_2 \ * \ q)\Big)$$

$$(p_1 \Rightarrow p_2) \not\Rightarrow (p_1 \mathrel{-\!\!*} p_2)$$

$$(p_1 \mathrel{-\!\!*} p_2) \not\Rightarrow (p_1 \Rightarrow p_2)$$

Specifications

- A *partial correctness specification*

$$\{p\}\ c\ \{q\}$$

  is *valid* iff, 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.

- A *total correctness specification*

$$[\,p\,]\ c\ [\,q\,]$$

  is *valid* iff, 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.

Thus:

• • • Well-specified programs don't go wrong. • • •

— and memory-fault checking is unnecessary.

(In this course, we will only consider partial correctness.)

Examples of Valid Specifications

$$\{\mathsf{x} - \mathsf{y} > 3\}\ \mathsf{x} := \mathsf{x} - \mathsf{y}\ \{\mathsf{x} > 3\}$$

$$\{\mathsf{x} + \mathsf{y} \geq 17\}\ \mathsf{x} := \mathsf{x} + 10\ \{\mathsf{x} + \mathsf{y} \geq 27\}$$

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

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

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

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

$$\{\mathsf{x} \leq 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\}$$

$$\{\mathbf{true}\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\}$$

$$\{\mathsf{x} > 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathbf{false}\}$$

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

Hoare's Inference Rules for Specifications:

Assignment (AS)

$$\frac{}{\{p/v \to e\}\ v := e\ \{p\}} \qquad \frac{}{[\,p/v \to e\,]\ v := e\ [\,p\,]}$$

Instances

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

Sequential Composition (SQ)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\, ;\, c_2\ \{r\}} \qquad \frac{[\,p\,]\ c_1\ [\,q\,] \qquad [\,q\,]\ c_2\ [\,r\,]}{[\,p\,]\ c_1\, ;\, c_2\ [\,r\,]}$$

An Instance

$$\frac{\begin{array}{l}\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \qquad\qquad\qquad\qquad \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\, ;\, \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

Strengthening Precedent (SP)

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

An Instance

$$\frac{\begin{array}{c}\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n} \\ \{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

Since they are applicable to arbitrary commands, the rules (SP) and (WC) (to be introduced later) are called *structural rules.* One premiss of each of these rules is an assertion, which is called a *verification condition* (VC). The verification conditions are used to introduce mathematical facts about data types into proofs of specifications.

Partial Correctness of **while** (WHP)

$$\frac{\{i \wedge b\}\; c\; \{i\}}{\{i\}\; \textbf{while}\; b\; \textbf{do}\; c\; \{i \wedge \neg\, b\}}$$

Here i is the *invariant*.

An Instance

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\; \mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}\; \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while}\; \mathsf{k} \neq \mathsf{n}\; \textbf{do}\; (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \end{array}}$$

Weakening Consequent (WC)

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

An Instance

$$\frac{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\, ;\, \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \\ \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}\end{array}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\, ;\, \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Conditional (CD)

$$\frac{\{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \neg\, b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

skip (SK)

$$\frac{}{\{p\}\ \textbf{skip}\ \{p\}}$$

Variable Declaration (DC)

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\}}$$

when v does not occur free in p or q.

Here the requirement on the declared variable v formalizes the concept of *locality*, i.e., that the value of v when c begins execution has no effect on this execution, and that the value of v when c finishes execution has no effect on the rest of the program.

Annotated Specifications

In an *annotated specification*, additional assertions called *annotations* are placed within the command in such a way that the proof of the specification can be constructed from the annotations. For example:

$$
\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
\{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\
\mathsf{k} := 0\,; \\
\{1 = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\mathsf{y} := 1\,; \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do} \\
\quad \big(\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\} \\
\quad \mathsf{k} := \mathsf{k} + 1\,; \\
\quad \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\quad \mathsf{y} := 2 \times \mathsf{y}\big) \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \\
\{\mathsf{y} = 2^{\mathsf{n}}\}
\end{array}
$$

Why Annotations Are Needed

Without annotations, it is not straightforward to construct a proof of a specification from the specification itself. For example, if we try to use the rule for sequential composition,

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ c_2\ \{r\},}$$

to obtain the main step of a proof of the specification

$$\begin{array}{l}\{\mathsf{n} \geq 0\} \\ (\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ; \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\},\end{array}$$

there is no indication of what assertion should replace the metavariable q.

Why Annotations Are Needed (continued)

But if we change the rule to

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\underline{\{q\}}\ c_2\ \{r\}},$$

then the new rule requires the annotation q to occur in the conclusion:

$\{\mathsf{n} \geq 0\}$
$(\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ;$
$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
$\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})$
$\{\mathsf{y} = 2^{\mathsf{n}}\}.$

Then, once q is determined, the premisses must be

$\{\mathsf{n} \geq 0\}$
$(\mathsf{k} := 0\ ;\ \mathsf{y} := 1);$
$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$

and

$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
$\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})$
$\{\mathsf{y} = 2^{\mathsf{n}}\}.$

The basic trick is to add annotations to the conclusions of the inference rules so that the conclusion of each rule completely determines its premisses.

Rules for Annotated Specifications

To obtain annotated specifications, we must change the following rules:

Sequential Composition (SQAN)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ \underline{\{q\}}\ c_2\ \{r\}}$$

Strengthening Precedent (SPAN)

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\ \underline{\{q\}}\ c\ \{r\}}$$

Weakening Consequent (WCAN)

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\ c\ \underline{\{q\}}\ \{r\}}$$

Rules for Reducing Annotations

In their present form, our inference rules lead to many more annotations than necessary. To reduce the number of annotations, it is useful to change further rules. (Each of the following rules can be derived from earlier rules.)

Assignment (ASAN)

$$\frac{p_0 \Rightarrow (p/v \to e)}{\{p_0\}\ v := e\ \{p\}} \qquad \frac{\{p_0\}\ c\ \{p/v \to e\}}{\{p_0\}\ c\, ;\, v := e\ \{p\}}$$

skip (SKAN)

$$\frac{p_0 \Rightarrow p}{\{p_0\}\ \mathbf{skip}\ \{p\}}$$

Partial Correctness of **while** (WHPAN)

$$\frac{\{i \wedge b\}\ c\ \{i\} \qquad (i \wedge \neg\, b) \Rightarrow p}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{p\}}$$

A Minimally Annotated Specification

$$\{\mathsf{n} \geq 0\}$$
$$\mathsf{k} := 0\,;\mathsf{y} := 1\,;$$
$$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$$
$$\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y})$$
$$\{\mathsf{y} = 2^{\mathsf{n}}\}$$

Structural Inference Rules

In addition to Strengthening Precedent and Weakening Consequent, there are a number of other *structural* rules of Hoare logic that are applicable to all commands.

Renaming (RN)

$$\frac{\{p\}\ c\ \{q\}}{\{p'\}\ c'\ \{q'\}},$$

where p', c', and q' are obtained from p, c, and q by zero or more renamings of bound variables.

It is sometimes necessary to use renaming in combination with the rule for variable declarations, e.g.

$$\cfrac{\cfrac{\{\mathsf{x} = 0\}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}}{\{\mathsf{x} = 0\}\ \mathbf{newvar}\ \mathsf{y}\ \mathbf{in}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}}}{\{\mathsf{x} = 0\}\ \mathbf{newvar}\ \mathsf{x}\ \mathbf{in}\ \mathsf{x} := 1\ \{\mathsf{x} = 0\}.}$$

Substitution (SUB)

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots,\ v_n \to e_n,}$$

where $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

For example, in

$$\{\mathsf{x} = \mathsf{y}\}\ \mathsf{x} := \mathsf{x} + \mathsf{y}\ \{\mathsf{x} = 2 \times \mathsf{y}\},$$

one can substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{w} - 1$ to infer

$$\{\mathsf{z} = 2 \times \mathsf{w} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{w} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{w} - 1)\}.$$

But one cannot substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{z} - 1$ to infer the invalid

$$\{\mathsf{z} = 2 \times \mathsf{z} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{z} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{z} - 1)\}.$$

The rule for substitution will become important when we consider procedures.

A special case of this rule occurs for an *identity* substitution, in which each e_i is v_i. Such substitutions still permit renaming (by choosing the v_{new}). Thus the Renaming rule is a special case of the substitution rule.

Structural Inference Rules:

Conjunction (CONJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}}$$

Disjunction (DISJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \vee p_2\}\ c\ \{q_1 \vee q_2\}}$$

Universal Quantification (UQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\forall v.\ p\}\ c\ \{\forall v.\ q\},}$$

where v is not free in c.

Existential Quantification (EQ)

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

where v is not free in c.

The Failure of the Rule of Constancy

On the other hand, there is one rule in Hoare logic,

- 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.

that is *unsound* in separation logic, 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 $\mathsf{x} = \mathsf{y}$.

Instead, we have 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.

Local Specifications

- The *footprint* of a command is the variables and the parts of the heap that are actually used by the command.

- A specification of a command is *local* when it mentions only the footprint.

- By using the frame rule, one can move from local to non-local specifications.

For example,

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

Inference Rules for Mutation

The local form (MUL):

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

The global form (MUG):

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

The backward-reasoning form (MUBR):

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

One can derive (MUG) from (MUL) by using the frame rule; one can go in the opposite direction by taking r to be **emp**.

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

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

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

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

Exercise 3

Fill in the postconditions in

$$\{(e_1 \mapsto -) \ast (e_2 \mapsto -)\}\ [e_1] := e_1' \,; [e_2] := e_2'\ \{?\}$$

$$\{(e_1 \mapsto -) \wedge (e_2 \mapsto -)\}\ [e_1] := e_1' \,; [e_2] := e_2'\ \{?\}.$$

to give two sound inference rules describing a sequence of two mutations. Your postconditions should be as strong as possible. Give a derivation of each of these inference rules, exhibited as an annotated specification.

Inference Rules for Deallocation

The local form (DISL):

$$\overline{\{e \mapsto -\}\ \textbf{dispose}\ e\ \{\textbf{emp}\}.}$$

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

$$\overline{\{(e \mapsto -) \ast r\}\ \textbf{dispose}\ e\ \{r\}.}$$

One can derive (DISG) from (DISL) by using (FR); one can go in the opposite direction by taking r to be **emp**.

Inference Rules for Noninterfering Allocation

The local form (CONSNIL):

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{v \mapsto e_0, \ldots, e_{n-1}\},}$$

where $v \notin \mathrm{FV}(e_0, \ldots, e_{n-1})$.

The global form (CONSNIG):

$$\frac{}{\{r\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{(v \mapsto e_0, \ldots, e_{n-1})\ *\ r\},}$$

where $v \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, r)$.

One can derive (CONSNIG) from (CONSNIL) by using the frame rule (FR); one can go in the opposite direction by taking r to be **emp**.

Inference Rules for General Allocation

The local form (CONSL):

$$\overline{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{v \mapsto e'_0, \ldots, e'_{n-1}\}}.$$

where v' is distinct from v, and e'_i denotes $e_i/v \rightarrow v'$.

The global form (CONSG):

$$\overline{\{r\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{\exists v'.\ (v \mapsto e'_0, \ldots, e'_{n-1})\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, r)$, e'_i denotes $e_i/v \rightarrow v'$, and r' denotes $r/v \rightarrow v'$.

The backward-reasoning form (CONSBR):

$$\overline{\{\forall v''.\ (v'' \mapsto e_0, \ldots, e_{n-1}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{p\}},$$

where v'' is distinct from v, $v'' \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, p)$, and p'' denotes $p/v \rightarrow v''$.

Inference Rules for General Allocation (continued)

Let $\overline{e}$ abbreviate $e_0, \ldots, e_{n-1}$, $\overline{e}'$ abbreviate $e'_0, \ldots, e'_{n-1}$, and similarly for e''. One can derive (CONSG) from (CONSL) by using (FR) and (EQ):

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

To go in the other direction:

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

Inference Rules for General Allocation (continued)

In order to derive (CONSBR) from (CONSG), we choose $v' \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, p)$ to be distinct from v and v'', take r to be $\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''$, and use various laws about quantifiers, as well as $q \ * \ (q \mathrel{-\!\!*} p) \Rightarrow p$:

$$
\begin{array}{l}
\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v'.\ (v \mapsto \overline{e}') \ * \ (\forall v''.\ (v'' \mapsto \overline{e}') \mathrel{-\!\!*} p'')\} \\
\{\exists v'.\ (v \mapsto \overline{e}') \ * \ ((v \mapsto \overline{e}') \mathrel{-\!\!*} p)\} \\
\{\exists v'.\ p\} \\
\{p\}.
\end{array}
$$

To go in the other direction, we choose $v'' \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, r)$ to be distinct from v and v', take p to be $\exists v'.\ (v \mapsto \overline{e}') \ * \ r'$, and use various laws about quantifiers, as well as $r \Rightarrow (q \mathrel{-\!\!*} (q \ * \ r))$:

$$
\begin{array}{l}
\{r\} \\
\{\forall v''.\ r\} \\
\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} ((v'' \mapsto \overline{e}) \ * \ r)\} \\
\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (((v'' \mapsto \overline{e}') \ * \ r')/v' \to v)\} \\
\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (\exists v'.\ (v'' \mapsto \overline{e}') \ * \ r')\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v'.\ (v \mapsto \overline{e}') \ * \ r'\}.
\end{array}
$$

Inference Rules for Lookup

The local form (LKL):

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

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

The global form (LKG):

$$\overline{\begin{array}{r}\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \to v)\}\ v := [e] \qquad\qquad\qquad\qquad \\ \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \to v)\},\end{array}}$$

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

The backward-reasoning form (LKBR):

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

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

The global noninterfering form (LKNIG):

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

where $v \notin \mathrm{FV}(e)$, $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$. As a special case,

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

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

Inference Rules for Lookup (continued)

To derive (LKG) from (LKL), we use the frame rule (FR):

$$
\begin{aligned}
&\{(v = v' \wedge (e \mapsto v'')) \ * \ (r/v' \to v)\}\\
&\{(v = v' \wedge (e \mapsto v'')) \ * \ r\}\\
&v := [e]\\
&\{(v = v'' \wedge (e' \mapsto v'')) \ * \ r\}\\
&\{(v = v'' \wedge (e' \mapsto v)) \ * \ (r/v'' \to v)\},
\end{aligned}
$$

and the rule for existential quantification (EQ):

$$
\begin{aligned}
&\{\exists v''.\ (e \mapsto v'') \ * \ (r/v' \to v)\}\\
&\{\exists v', v''.\ (v = v' \wedge (e \mapsto v'')) \ * \ (r/v' \to v)\}\\
&v := [e]\\
&\{\exists v', v''.\ (v = v'' \wedge (e' \mapsto v)) \ * \ (r/v'' \to v)\}\\
&\{\exists v'.\ (e' \mapsto v) \ * \ (r/v'' \to v)\}.
\end{aligned}
$$

To derive (LKBR) from (LKG), we take r to be $(e' \mapsto v'') \mathrel{-\!\!*} p''$ and use the axiom schemata $(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)$ and $q \ * \ (q \mathrel{-\!\!*} p) \Rightarrow p$. (In the first line, we can rename v'' to be any variable not in $\mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$.)

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

Inference Rules for Lookup (continued)

To derive (LKL) from (LKBR), we take p to be $v = v'' \wedge (e' \mapsto v'')$, after renaming v'' to $\hat{v}$:

$$\begin{aligned}
&\{v = v' \wedge (e \mapsto v'')\}\\
&\{v = v' \wedge (e \hookrightarrow v'') \wedge (e' \mapsto v'')\}\\
&\{(e \hookrightarrow v'') \wedge (e' \mapsto v'')\}\\
&\{\exists \hat{v}.\ (e \hookrightarrow v'') \wedge \hat{v} = v'' \wedge (e' \mapsto v'')\}\\
&\{\exists \hat{v}.\ (e \hookrightarrow \hat{v}) \wedge \hat{v} = v'' \wedge (e' \mapsto v'')\}\\
&v := [e]\\
&\{v = v'' \wedge (e' \mapsto v'')\}.
\end{aligned}$$

To derive (LKNIG) from (LKG), suppose v and v'' are distinct variables, $v \notin \mathrm{FV}(e)$, and $v'' \notin \mathrm{FV}(e) \cup \mathrm{FV}(p)$. We take v' to be a variable distinct from v and v'' that does not occur free in e or p, and r to be $p'' = p/v \to v''$. Then

$$\begin{aligned}
&\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\\
&\{\exists v''.\ (e \mapsto v'')\ *\ (p''/v' \to v)\}\\
&v := [e]\\
&\{\exists v'.\ (e' \mapsto v)\ *\ (p''/v'' \to v)\}\\
&\{\exists v'.\ (e \mapsto v)\ *\ p)\}\\
&\{(e \mapsto v)\ *\ p)\}.
\end{aligned}$$

In the first line, we can rename v'' to be any variable not in $\mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, so that v'' does not need to be distinct from v.

Weakest Preconditions for the New Commands

For each of the new heap-manipulating commands, the backward-reasoning form gives both the weakest liberal precondition and (since these commands always terminate) the weakest precondition.

Mutation:

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

Allocation:

$$\mathbf{wp}(v := \mathbf{cons}(e_0, \ldots, e_{n-1}), p) = \forall v''.\ (v'' \mapsto e_0, \ldots, e_{n-1}) \mathrel{-\!\!*} p'',$$

where v'' is distinct from v, $v'' \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, p)$, and p'' denotes $p/v \rightarrow v''$.

Disposal:

$$\mathbf{wp}(\mathbf{dispose}\ e, r) = (e \mapsto -) \ast r.$$

Lookup:

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

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

Similar rules hold for **wlp**.

An Annotated Specification

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

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

$$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\} \quad \text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{a}\}$$

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

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{a} \ * \ \mathsf{y} \mapsto \mathsf{b} \ * \ \mathsf{y} + 1 \mapsto \mathsf{b}\}$$
$$(p/v \to e \Rightarrow \exists v.\ p)$$
$$\{(\mathsf{x} \mapsto \mathsf{a}, -) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ (\exists \mathsf{a}.\ \mathsf{x} + 1 \mapsto \mathsf{a}) \ * \ \mathsf{y} \mapsto \mathsf{b} \ * \ \mathsf{y} + 1 \mapsto \mathsf{b}\}$$

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

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ * \ \mathsf{y} \mapsto \mathsf{b} \ * \ \mathsf{y} + 1 \mapsto \mathsf{b}\}$$
$$(p/v \to e \Rightarrow \exists v.\ p)$$
$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, -)\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ * \ \mathsf{y} \mapsto \mathsf{b} \ * \ (\exists \mathsf{b}.\ \mathsf{y} + 1 \mapsto \mathsf{b})\}$$

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

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ * \ (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ * \ \mathsf{y} \mapsto \mathsf{b} \ * \ \mathsf{y} + 1 \mapsto \mathsf{x} - \mathsf{y}\}$$
$$(p/v \to e \Rightarrow \exists v.\ p)$$
$$\{\exists \mathsf{o}.\ (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \ * \ (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b},\ -\mathsf{o})\}$$
$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ * \ \mathsf{x} + 1 \mapsto \mathsf{o} \ * \ \mathsf{x} + \mathsf{o} \mapsto \mathsf{b} \ * \ \mathsf{x} + \mathsf{o} + 1 \mapsto -\mathsf{o}\}$$

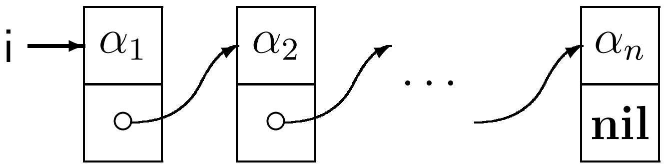
Notation for Sequences

When α and β are sequences, we write

- ϵ for the empty sequence.
- $[x]$ for the single-element sequence containing x. (We will omit the brackets when x 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.

Singly-linked Lists

$\mathsf{list}\ \alpha\ \mathsf{i}$:



is defined by

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

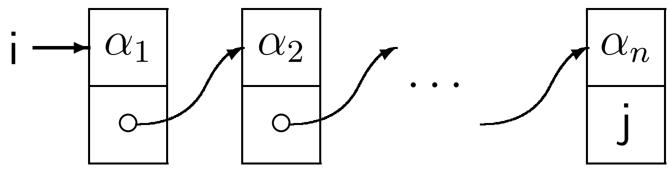
$$\mathsf{list}\ (\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}.$$

Reversing a List

$$
\begin{array}{l}
\{\mathsf{list}\,\alpha_0\,\mathsf{i}\} \\
\{\mathsf{list}\,\alpha_0\,\mathsf{i} \ast (\mathbf{emp} \wedge \mathbf{nil} = \mathbf{nil})\} \\
\mathsf{j} := \mathbf{nil}\,; \\
\{\mathsf{list}\,\alpha_0\,\mathsf{i} \ast (\mathbf{emp} \wedge \mathsf{j} = \mathbf{nil})\} \\
\{\mathsf{list}\,\alpha_0\,\mathsf{i} \ast \mathsf{list}\,\epsilon\,\mathsf{j}\} \\
\{\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} \\
\quad \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\} \\
\quad\ \ \{\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\} \\
\quad\ \ \mathsf{k} := [\mathsf{i} + 1]\,; \\
\quad\ \ \{\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\} \\
\quad\ \ [\mathsf{i} + 1] := \mathsf{j}\,; \\
\quad\ \ \{\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\} \\
\quad\ \ \{\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\} \\
\quad\ \ \{\exists \alpha, \beta.\ (\mathsf{list}\,\alpha\,\mathsf{k} \ast \mathsf{list}\,\beta\,\mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\} \\
\quad\ \ \mathsf{j} := \mathsf{i}\,;\, \mathsf{i} := \mathsf{k} \\
\quad\ \ \{\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}\}
\end{array}
$$

Singly-linked List Segments

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



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

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

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

Properties

$$\mathsf{lseg}\;\mathsf{a}\,(\mathsf{i},\mathsf{j}) \Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j}$$

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

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

$$\mathsf{list}\;\alpha\;\mathsf{i} \Leftrightarrow \mathsf{lseg}\;\alpha\,(\mathsf{i},\mathbf{nil}).$$

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:

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

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

$$\begin{aligned}
&\exists \mathsf{j}.\ \mathsf{lseg}\ \mathsf{a}{\cdot}\alpha'\,(\mathsf{i},\mathsf{j}) \ *\ \mathsf{lseg}\ \beta\,(\mathsf{j},\mathsf{k})\\
&\quad\Leftrightarrow \exists \mathsf{j},\mathsf{l}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{l} \ *\ \mathsf{lseg}\ \alpha'\,(\mathsf{l},\mathsf{j}) \ *\ \mathsf{lseg}\ \beta\,(\mathsf{j},\mathsf{k})\\
&\quad\Leftrightarrow \exists \mathsf{l}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{l} \ *\ \mathsf{lseg}\ \alpha'{\cdot}\beta\,(\mathsf{l},\mathsf{k}) \qquad \text{(induction hypothesis)}\\
&\quad\Leftrightarrow \mathsf{lseg}\ \mathsf{a}{\cdot}\alpha'{\cdot}\beta\,(\mathsf{i},\mathsf{k})
\end{aligned}$$

Emptyness Conditions

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

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

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

Nontouching List Segments

When

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

we have

$$\begin{aligned}&\exists \mathsf{i}_1, \ldots \mathsf{i}_{n-1}.\\ &\qquad (\mathsf{i}_0 \mapsto a_1, \mathsf{i}_1) \ast (\mathsf{i}_1 \mapsto a_2, \mathsf{i}_2) \ast \cdots \ast (\mathsf{i}_{n-1} \mapsto a_n, \mathsf{i}_n).\end{aligned}$$

Thus i_0, ..., i_{n-1} are distinct, but i_n is not constrained, and may equal any of the i_0, ..., i_{n-1}. In this case, we say that the list segment is *touching*.

We can define nontouching list segments in terms of lseg:

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

or we can define them inductively:

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

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

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:

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

Inserting a List Element

- At the beginning:

$$
\begin{aligned}
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\}\\
&\mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{i});\\
&\{\mathsf{k} \mapsto \mathsf{a},\mathsf{i}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\}\\
&\{\exists \mathsf{i}.\ \mathsf{k} \mapsto \mathsf{a},\mathsf{i}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\}\\
&\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{k},\mathsf{j})\}\\
&\mathsf{i} := \mathsf{k}\\
&\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j})\}
\end{aligned}
$$

- At the end of a nonempty segment:

$$
\begin{aligned}
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\}\\
&\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})\ ;\\
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto -\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&[\mathsf{j}+1] := \mathsf{l}\\
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}\\
&\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{k})\}
\end{aligned}
$$

Deleting a List Element

- At the beginning:

$$
\begin{array}{l}
\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}\\
\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}\\
\{\exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}\\
\mathsf{j} := [\mathsf{i}+1]\ ;\\
\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}\\
\textbf{dispose}\ \mathsf{i}\ ;\\
\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}\\
\textbf{dispose}\ \mathsf{i}+1\ ;\\
\{\mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}\\
\mathsf{i} := \mathsf{j}\\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{k})\}
\end{array}
$$

- At the end of a nonempty segment:

$$
\begin{array}{l}
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}\\
[\mathsf{j}+1] := \mathsf{l}\ ;\\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}\\
\textbf{dispose}\ \mathsf{k}\ ;\ \textbf{dispose}\ \mathsf{k}+1\\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\}\\
\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\}
\end{array}
$$

Exercise 4

When

$$\exists \alpha, \beta.\ (\mathbf{lseg}\, \alpha\, (\mathsf{i}, \mathsf{j})\ *\ \mathbf{lseg}\, \beta\, (\mathsf{j}, \mathsf{k})) \wedge \gamma = \alpha{\cdot}\beta,$$

we say that j is an *interior pointer* of the list segment described by $\mathbf{lseg}\, \gamma\, (\mathsf{i}, \mathsf{k})$.

1. Give an assertion describing a list segment with two interior pointers j_1 and j_2, such that j_1 comes before than, or at the same point as, j_2 in the ordering of the elements of the list segment.

2. Give an assertion describing a list segment with two interior pointers j_1 and j_2, where there is no constraint on the relative positions of j_1 and j_2.

3. Prove that the first assertion implies the second.

Exercise 5

A *braced list segment* is a list segment with an interior pointer j to its last element; in the special case where the list segment is empty, j is **nil**. Formally,

$$\mathbf{brlseg}\,\epsilon\,(\mathsf{i},\mathsf{j},\mathsf{k}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{k} \wedge \mathsf{j} = \mathbf{nil}$$

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

1. Prove the assertion

$$\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k}) \Rightarrow \mathbf{lseg}\,\alpha\,(\mathsf{i},\mathsf{k}).$$

2. Give an annotated specification of a command that changes the final pointer of a braced list segment from k to l. The command should not construct or deallocate any list structure, and it should not change the value of j. It should satisfy

$$\{\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\} \;\;\cdots\;\; \{\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{l})\}.$$

3. Give an annotated specification of a command that concatenates two braced list segments. The command should not construct or deallocate any list structure, and it should not change the value of k_2. It should satisfy

$$\{\mathbf{brlseg}\,\alpha\,(\mathsf{i}_1,\mathsf{j}_1,\mathsf{k}_1) \;*\; \mathbf{brlseg}\,\beta\,(\mathsf{i}_2,\mathsf{j}_2,\mathsf{k}_2)\}$$
$$\ldots$$
$$\{\mathbf{brlseg}\,\alpha{\cdot}\beta\,(\mathsf{i}_1,\mathsf{j},\mathsf{k}_2)\}.$$

Exercise 5 (continued)

4. Give an annotated specification of a command that appends an item a at the beginning of a braced list segment. The command should not change the value of k, and should satisfy

$$\{\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\}\ \cdots\ \{\mathbf{brlseg}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\}.$$

5. Give an annotated specification of a command that appends an item a at the end of a braced list segment. The command should not change the value of k, and should satisfy

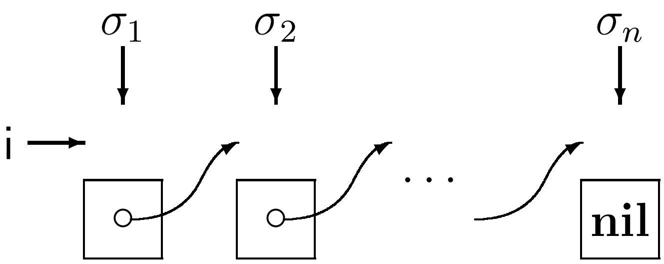
$$\{\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\}\ \cdots\ \{\mathbf{brlseg}\,\alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{j},\mathsf{k})\}.$$

6. Give an annotated specification of a command that deletes an item a from the beginning of a nonempty braced list segment. The command should not change the value of k, and should satisfy

$$\{\mathbf{brlseg}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\}\ \cdots\ \{\mathbf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k})\}.$$

Bornat Lists

$\mathsf{listN}\ \sigma\ \mathsf{i}$:



is defined by

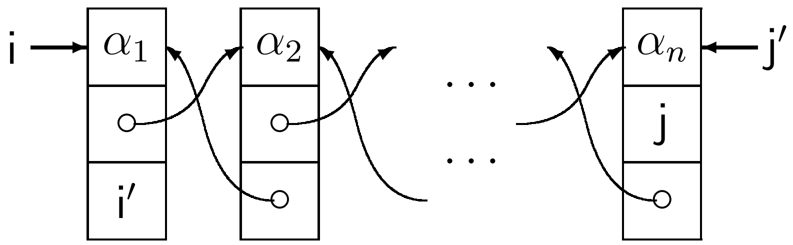
$$\mathsf{listN}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

$$\mathsf{listN}\ (\mathsf{a}{\cdot}\sigma)\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{a} = \mathsf{i} \wedge \exists \mathsf{j}.\ \mathsf{i} + 1 \mapsto \mathsf{j}\ *\ \mathsf{listN}\ \sigma\ \mathsf{j}.$$

Similarly, one can define Bornat list segments and nontouching Bornat list segments.

Doubly-Linked List Segments

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



is defined by

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

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

Properties

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

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

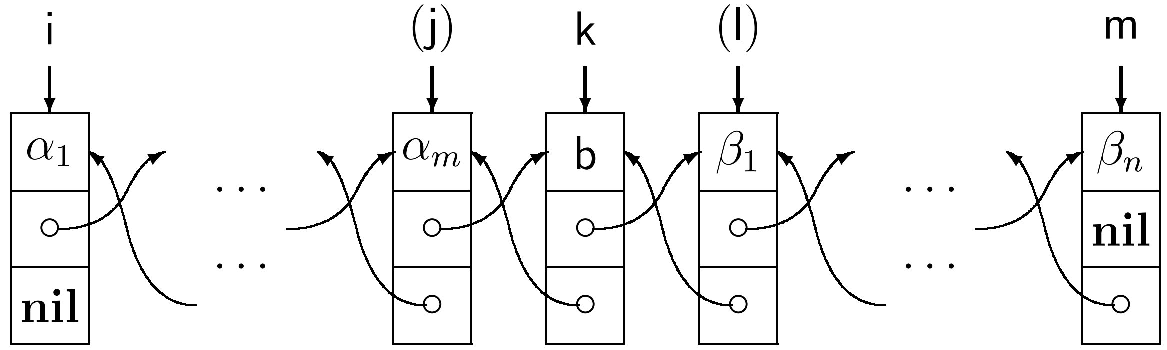
$$\mathsf{dlseg}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \Leftrightarrow \exists \mathsf{j}'.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{k}', \mathsf{j}') \ *\ \mathsf{k}' \mapsto \mathsf{b}, \mathsf{k}, \mathsf{j}'.$$

Emptyness Conditions

$$\begin{array}{lcccc}
\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') & \wedge & \mathsf{i} = \mathbf{nil} & \Rightarrow & (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil} \wedge \mathsf{i}' = \mathsf{j}') \\
\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') & \wedge & \mathsf{j}' = \mathbf{nil} & \Rightarrow & (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}) \\
\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') & \wedge & \mathsf{i} \neq \mathsf{j} & \Rightarrow & \alpha \neq \epsilon \\
\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') & \wedge & \mathsf{i}' \neq \mathsf{j}' & \Rightarrow & \alpha \neq \epsilon.
\end{array}$$

(One can also define doubly-linked lists and nontouching segments.)

Deleting an Element from a Doubly-Linked List



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

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

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

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

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

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

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

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

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

$\mathbf{else}$

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

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

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

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

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

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

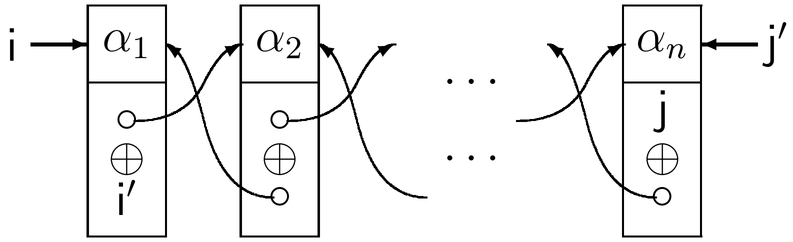
$\quad \vdots$

Deleting from a Doubly-Linked List (continued)

$$
\begin{array}{l}
\quad\quad \vdots \\
\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \;*\; \mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})\} \\
\mathbf{if}\ \mathsf{l} = \mathbf{nil}\ \mathbf{then} \\
\quad \{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{k} = \mathsf{m} \wedge \beta = \epsilon\} \\
\quad \mathsf{m} := \mathsf{j} \\
\quad \{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{j} = \mathsf{m} \wedge \beta = \epsilon\} \\
\mathbf{else} \\
\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \;*\; \mathsf{l} \mapsto \mathsf{a},\mathsf{n},\mathsf{k} \\
\quad\quad\quad *\; \mathsf{dlseg}\,\beta'\,(\mathsf{n},\mathsf{l},\mathbf{nil},\mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\} \\
\quad [\mathsf{l} + 2] := \mathsf{j} \\
\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \;*\; \mathsf{l} \mapsto \mathsf{a},\mathsf{n},\mathsf{j} \\
\quad\quad\quad *\; \mathsf{dlseg}\,\beta'\,(\mathsf{n},\mathsf{l},\mathbf{nil},\mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\} \\
\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j}) \;*\; \mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\} \\
\{\mathsf{dlseg}\,\alpha{\cdot}\beta\,(\mathsf{i},\mathbf{nil},\mathbf{nil},\mathsf{m})\}
\end{array}
$$

An Inductive Definition: Xor-Linked List Segments

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



is defined by

$$\mathsf{xlseg}\,\epsilon\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \stackrel{\mathrm{def}}{=} \mathbf{emp} \land \mathsf{i} = \mathsf{j} \land \mathsf{i}' = \mathsf{j}'$$

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

Properties

$$\mathsf{xlseg}\,\mathsf{a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \Leftrightarrow \mathsf{i} \mapsto \mathsf{a}, (\mathsf{j} \oplus \mathsf{i}') \land \mathsf{i} = \mathsf{j}'$$

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

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

Emptyness Conditions (as with dlseg)

$$\begin{array}{lcl}
\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ \land \ \mathsf{i} = \mathbf{nil} & \Rightarrow & (\alpha = \epsilon \land \mathsf{j} = \mathbf{nil} \land \mathsf{i}' = \mathsf{j}') \\
\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ \land \ \mathsf{j}' = \mathbf{nil} & \Rightarrow & (\alpha = \epsilon \land \mathsf{i}' = \mathbf{nil} \land \mathsf{i} = \mathsf{j}) \\
\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ \land \ \mathsf{i} \neq \mathsf{j} & \Rightarrow & \alpha \neq \epsilon \\
\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ \land \ \mathsf{i}' \neq \mathsf{j}' & \Rightarrow & \alpha \neq \epsilon.
\end{array}$$

Iterated Separating Conjunction

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

$$\begin{array}{l}
s, h \vDash \bigodot_{v=e}^{e'} p \text{ iff} \\
\qquad \textbf{let } m = [\![e]\!]_{\text{exp}} s,\ n = [\![e']\!]_{\text{exp}} s,\ I = \{\, i \mid m \leq i \leq n \,\} \textbf{ in} \\
\qquad\qquad \exists H \in I \to \text{Heaps}. \\
\qquad\qquad\qquad \forall i, j \in I.\ i \neq j \text{ implies } Hi \perp Hj \\
\qquad\qquad\qquad \text{and } h = \bigcup\{\, Hi \mid i \in I \,\} \\
\qquad\qquad\qquad \text{and } \forall i \in I.\ [\, s \mid v\colon i \,], Hi \vDash p.
\end{array}$$

Axiom Schemata

$$m > n \Rightarrow \Big(\bigodot_{i=m}^{n} p(i) \Leftrightarrow \textbf{emp}\Big)$$

$$m = n \Rightarrow \Big(\bigodot_{i=m}^{n} p(i) \Leftrightarrow p(m)\Big)$$

$$k \leq m \leq n+1 \Rightarrow \Big(\bigodot_{i=k}^{n} p(i) \Leftrightarrow \Big(\bigodot_{i=k}^{m-1} p(i) \ * \ \bigodot_{i=m}^{n} p(i)\Big)\Big)$$

$$\bigodot_{i=m}^{n} p(i) \Leftrightarrow \bigodot_{i=m-k}^{n-k} p(i+k)$$

$$m \leq n \Rightarrow \Big(\Big(\bigodot_{i=m}^{n} p(i)\Big) \ * \ q \Leftrightarrow \bigodot_{i=m}^{n} (p(i) \ * \ q)\Big)$$

when q is pure and $i \notin \mathrm{FV}(q)$

$$m \leq j \leq n \Rightarrow \Big(\Big(\bigodot_{i=m}^{n} p(i)\Big) \Rightarrow (p(j) \ * \ \textbf{true})\Big)$$

Array Allocation

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

$$
\begin{array}{lll}
 & \text{Store}: & \mathsf{x}\colon 3,\ \mathsf{y}\colon 4 \\
 & \text{Heap}: & \text{empty} \\
\mathsf{x} := \textbf{allocate}\ \mathsf{y} & & \\
 & \text{Store}: & \mathsf{x}\colon 37,\ \mathsf{y}\colon 4 \\
 & \text{Heap}: & 37\colon -,\ 38\colon -,\ 39\colon -,\ 40\colon -
\end{array}
$$

Inference Rules

Noninterfering:

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

where v does not occur free in r or e.

The local form:

$$\frac{}{\{v = v' \wedge \textbf{emp}\}\ v := \textbf{allocate}\ e\ \{\bigodot_{i=v}^{v+e'-1} i \mapsto -\},}$$

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

The global form:

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

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

The backward-reasoning form:

$$\frac{}{\{\forall v''.\ (\bigodot_{i=v''}^{v''+e-1} i \mapsto -) \mathrel{-\!\!*} p''\}\ v := \textbf{allocate}\ e\ \{p\},}$$

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

Exercise 6

Derive the axiom schemata

$$m \leq j \leq n \Rightarrow \left(\left(\bigodot_{i=m}^{n} p(i)\right) \Rightarrow (p(j) \ * \ \mathbf{true})\right)$$

from the axiom schematas for iterating separating conjunction given on page **??**.

A Cyclic Buffer Using an Array

We assume that an n-element array has been allocated at location l. Let $\phi(x)$ be the unique integer such that

$$\phi(x) \equiv_{\mathrm{mod}\ \mathsf{n}} x \qquad\qquad \mathsf{l} \leq \phi(x) < \mathsf{l} + \mathsf{n}.$$

We will use the variables

m	number of active elements
i	pointer to first active element
j	pointer to first inactive element

Let R abbreviate the assertion

$$R \stackrel{\mathrm{def}}{=} 0 \leq \mathsf{m} \leq \mathsf{n} \wedge \mathsf{l} \leq \mathsf{i} < \mathsf{l} + \mathsf{n} \wedge \mathsf{l} \leq \mathsf{j} < \mathsf{l} + \mathsf{n} \wedge \mathsf{j} \equiv_{\mathrm{mod}\ \mathsf{n}} \mathsf{i} + \mathsf{m}$$

It is easy to show that

$$\{R \wedge \mathsf{m} < \mathsf{n}\}$$
$$\mathsf{m} := \mathsf{m} + 1\,;\ \mathbf{if}\ \mathsf{j} = \mathsf{l} + \mathsf{n} - 1\ \mathbf{then}\ \mathsf{j} := \mathsf{l}\ \mathbf{else}\ \mathsf{j} := \mathsf{j} + 1$$
$$\{R\}$$

and

$$\{R \wedge \mathsf{m} > 0\}$$
$$\mathsf{m} := \mathsf{m} - 1\,;\ \mathbf{if}\ \mathsf{i} = \mathsf{l} + \mathsf{n} - 1\ \mathbf{then}\ \mathsf{i} := \mathsf{l}\ \mathbf{else}\ \mathsf{i} := \mathsf{i} + 1$$
$$\{R\}$$

When the buffer contains a sequence α, it should satisfy

$$((\textstyle\bigodot_{\mathsf{k}=0}^{\mathsf{m}-1} \phi(\mathsf{i}+\mathsf{k}) \mapsto \alpha_{\mathsf{k}+1}) \ast (\textstyle\bigodot_{\mathsf{k}=0}^{\mathsf{n}-\mathsf{m}-1} \phi(\mathsf{j}+\mathsf{k}) \mapsto -)) \wedge \mathsf{m} = \#\alpha \wedge R.$$

Inserting an Element

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto \alpha_{k+1}) * (\bigodot_{k=0}^{n-m-1} \phi(j+k) \mapsto -))$$
$$\quad \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto \alpha_{k+1}) * (\bigodot_{k=0}^{0} \phi(j+k) \mapsto -) *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto \alpha_{k+1}) * \phi(j) \mapsto - *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$[j] := x\,;$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto \alpha_{k+1}) * \phi(j) \mapsto x *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto \alpha_{k+1}) * \phi(i+m) \mapsto x *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) * \phi(i+m) \mapsto (\alpha{\cdot}x)_{m+1} *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) * (\bigodot_{k=m}^{m} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) *$$
$$\quad (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$$

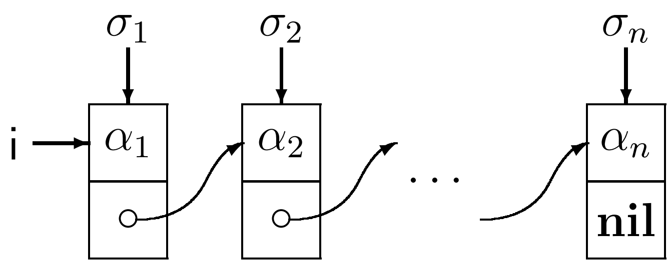
$$\{((\bigodot_{k=0}^{m} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) * (\bigodot_{k=1}^{n-m-1} \phi(j+k) \mapsto -))$$
$$\quad \wedge m+1 = \#(\alpha{\cdot}x) \wedge R \wedge m < n\}$$

$$\{((\bigodot_{k=0}^{m} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) * (\bigodot_{k=0}^{n-m-2} \phi(j+k+1) \mapsto -))$$
$$\quad \wedge m+1 = \#(\alpha{\cdot}x) \wedge R \wedge m < n\}$$

$m := m+1\,;$ **if** $j = l+n-1$ **then** $j := l$ **else** $j := j+1$

$$\{((\bigodot_{k=0}^{m-1} \phi(i+k) \mapsto (\alpha{\cdot}x)_{k+1}) * (\bigodot_{k=0}^{n-m-1} \phi(j+k) \mapsto -))$$
$$\quad \wedge m = \#(\alpha{\cdot}x) \wedge R\}$$

Connecting Two Views of Lists



If

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

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

and

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

$$\mathsf{listN}\ (\mathsf{b}{\cdot}\sigma)\ \mathsf{i} \stackrel{\mathrm{def}}{=} \mathsf{b} = \mathsf{i} \wedge \exists \mathsf{j}.\ \mathsf{i} + 1 \mapsto \mathsf{j}\ *\ \mathsf{listN}\ \sigma\ \mathsf{j},$$

then

$$\mathsf{list}\ \alpha\ \mathsf{i} \Leftrightarrow \exists \sigma.\ \#\sigma = \#\alpha \wedge (\mathsf{listN}\ \sigma\ \mathsf{i}\ *\ \bigodot_{\mathsf{k}=1}^{\#\alpha} \sigma_k \mapsto \alpha_k).$$

The proof is by induction on α.

Specifying Subset Lists

Let $\alpha, \beta \in \mathbf{Z}^*$ and $\sigma \in (\mathbf{Z}^*)^*$ in

$$\{\mathsf{list}\ \alpha\ \mathsf{i}\}$$
"Set j to list of lists of subsets of i"
$$\{\exists \sigma, \beta.\ \ \mathsf{ss}(\alpha^\dagger, \sigma) \wedge (\mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\},$$

where

$$\#\mathsf{ext}_\mathsf{a}\sigma \stackrel{\text{def}}{=} \#\sigma$$
$$(\mathsf{ext}_\mathsf{a}\sigma)_i \stackrel{\text{def}}{=} \mathsf{a}\cdot\sigma_i$$

$$\mathsf{ss}(\epsilon, \sigma) \stackrel{\text{def}}{=} \sigma = [\epsilon]$$
$$\mathsf{ss}(\mathsf{a}\cdot\alpha, \sigma) \stackrel{\text{def}}{=} \exists \sigma'.\ \ \mathsf{ss}(\alpha, \sigma') \wedge \sigma = (\mathsf{ext}_\mathsf{a}\sigma')^\dagger\cdot\sigma'$$

$$Q(\sigma, \beta) \stackrel{\text{def}}{=} \#\beta = \#\sigma \wedge \forall_{\mathsf{i}=1}^{\#\beta}(\mathsf{list}\ \sigma_\mathsf{i}\ \beta_\mathsf{i}\ *\ \mathbf{true})$$

$$R(\beta) \stackrel{\text{def}}{=} (\beta_{\#\beta} = \mathbf{nil} \wedge \mathbf{emp})\ *$$
$$\bigodot_{\mathsf{i}=1}^{\#\beta-1}(\exists \mathsf{a}, \mathsf{k}.\ \ \mathsf{i} < \mathsf{k} \leq \#\beta \wedge \beta_\mathsf{i} \mapsto \mathsf{a}, \beta_\mathsf{k}).$$

S-expressions (a la LISP)

$$\begin{aligned} \tau \in \text{S-exps iff} \\ & \tau \in \text{Atoms} \\ & \text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps.} \end{aligned}$$

Representing S-expressions by Trees (no sharing)

For $\tau \in$ S-exps, we define the assertion

$$\mathsf{tree}\, \tau\, (i)$$

by structural induction:

$$\mathsf{tree}\, a\, (i) \text{ iff } \mathbf{emp} \wedge i = a$$

$$\begin{aligned} &\mathsf{tree}\, (\tau_1 \cdot \tau_2)\, (i) \text{ iff} \\ &\quad \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ * \ \mathsf{tree}\, \tau_1\, (i_1) \ * \ \mathsf{tree}\, \tau_2\, (i_2). \end{aligned}$$

Representing S-expressions by Dags (with sharing)

For $\tau \in$ S-exps, we define

$$\mathsf{dag}\, \tau\, (i)$$

by:

$$\mathsf{dag}\, a\, (i) \text{ iff } i = a$$

$$\begin{aligned} &\mathsf{dag}\, (\tau_1 \cdot \tau_2)\, (i) \text{ iff} \\ &\quad \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ * \ (\mathsf{dag}\, \tau_1\, (i_1) \wedge \mathsf{dag}\, \tau_2\, (i_2)). \end{aligned}$$

Intuitionistic Assertions

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

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

An assertion p is intuitionistic iff

$$p \ * \ \mathbf{true} \Rightarrow p.$$

(The opposite implication always holds.)

The assertion $\mathsf{dag}\,\tau\,(i)$ is intuitionistic. The proof is by induction on the structure of τ. If τ is an atom a,

$$\begin{aligned} &\mathsf{dag}\,a\,(i) \ * \ \mathbf{true} \\ &\quad \Rightarrow i = a \ * \ \mathbf{true} \\ &\quad \Rightarrow i = a \\ &\quad \Rightarrow \mathsf{dag}\,a\,(i). \end{aligned}$$

since $i = a$ is pure. Otherwise, $\tau = (\tau_1 \cdot \tau_2)$, and

$$\begin{aligned} &\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \ * \ \mathbf{true} \\ &\quad \Rightarrow \exists i_1, i_2.\ i \mapsto i_1, i_2 \ * \ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)) \ * \ \mathbf{true} \\ &\quad \Rightarrow \exists i_1, i_2.\ i \mapsto i_1, i_2 \ * \\ &\qquad\quad ((\mathsf{dag}\,\tau_1\,(i_1) \ * \ \mathbf{true}) \wedge (\mathsf{dag}\,\tau_2\,(i_2) \ * \ \mathbf{true})) \\ &\quad \Rightarrow \exists i_1, i_2.\ i \mapsto i_1, i_2 \ * \ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)) \\ &\quad \Rightarrow \mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i), \end{aligned}$$

by the induction hypothesis for τ_1 and τ_2.

Inference for Simple Procedures

A simple procedure definition has the form

$$h(x_1, \cdots, x_m; y_1, \cdots, y_n) = c,$$

where $y_1, \cdots, y_n$ are the free variables modified by c, and $x_1, \cdots, x_m$ are the other free variables of c.

If one can prove the specification $\{p\}\ c\ \{q\}$, then one can assume the specification $\{p\}\ h(x_1, \cdots, x_m; y_1, \cdots, y_n)\ \{q\}$ in proving a specification of a command that contains calls of h:

Simple Procedures (SPROC)

When $h(x_1, \cdots, x_m; y_1, \cdots, y_n) = c$,

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ h(x_1, \cdots, x_m; y_1, \cdots, y_n)\ \{q\}.}$$

From the conclusion of this rule, one can reason about other calls by using the rule for free variable substitution (FVS), assuming that the variables modified by $h(x_1, \cdots, x_m; y_1, \cdots, y_n)$ are $y_1, \cdots, y_n$.

For partial correctness, if c contains recursive calls, one can assume $\{p\}\ h(x_1, \cdots, x_m; y_1, \cdots, y_n)\ \{q\}$ in proving $\{p\}\ c\ \{q\}$:

Simple Recursive Procedures (SRPROC)

When $h(x_1, \cdots, x_m; y_1, \cdots, y_n) = c$,

$$\frac{\begin{array}{c}\{p\}\ h(x_1, \cdots, x_m; y_1, \cdots, y_n)\ \{q\} \\ \vdots \\ \{p\}\ c\ \{q\}\end{array}}{\{p\}\ h(x_1, \cdots, x_m; y_1, \cdots, y_n)\ \{q\}.}$$

Copying Trees

We assume that the recursive procedure copytree satisfies

$$\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.$$

Then

$$
\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\} \\
\textbf{if } \mathsf{isatom}(\mathsf{i}) \textbf{ then} \\
\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\} \\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{i} = \tau))\} \\
\quad \mathsf{j} := \mathsf{i} \\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{j} = \tau))\} \\
\textbf{else} \\
\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{tree}\,(\tau_1 \cdot \tau_2)(\mathsf{i})\} \\
\quad \textbf{newvar } \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2 \textbf{ in} \\
\qquad \big(\mathsf{i}_1 := [\mathsf{i}]\,;\, \mathsf{i}_2 := [\mathsf{i}+1]\,; \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\qquad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2))\} \\
\qquad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\,; \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\qquad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\} \\
\qquad \vdots \\
\{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.
\end{array}
$$

Copying Trees (continued)

$$
\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\} \\
\qquad\quad \vdots \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2 \ * \\
\qquad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ * \ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2) \ * \ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\} \\
\qquad \mathbf{copytree}(\mathsf{i}_2; \mathsf{j}_2)\ ; \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2 \ * \ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ * \ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2) \ * \\
\qquad\quad \mathsf{tree}\,\tau_1\,(\mathsf{j}_1) \ * \ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2))\} \\
\qquad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2) \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2 \ * \ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ * \ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2) \ * \\
\qquad\quad \mathsf{j} \mapsto \mathsf{j}_1, \mathsf{j}_2 \ * \ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1) \ * \ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2))\} \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad (\mathsf{tree}\,(\tau_1 {\cdot} \tau_2)\,(\mathsf{i}) \ * \ \mathsf{tree}\,(\tau_1 {\cdot} \tau_2)\,(\mathsf{j}))\}\big) \\
\{\mathsf{tree}\,\tau(\mathsf{i}) \ * \ \mathsf{tree}\,\tau(\mathsf{j})\}.
\end{array}
$$

so that we may define

$$
\begin{array}{l}
\mathsf{copytree}(\mathsf{i}; \mathsf{j}) = \\
\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\
\qquad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in} \\
\qquad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ; \mathsf{i}_2 := [\mathsf{i} + 1]\ ; \\
\qquad\quad \mathbf{copytree}(\mathsf{i}_1; \mathsf{j}_1)\ ; \mathbf{copytree}(\mathsf{i}_2; \mathsf{j}_2)\ ; \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2)\big)
\end{array}
$$

Proof of the First Recursive Call

$$\dfrac{\dfrac{\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}}{\{\mathsf{tree}\,\tau_1(\mathsf{i}_1)\}\ \mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ \{\mathsf{tree}\,\tau_1(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\}}}{\begin{array}{l}\{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2)\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\}\\ \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2)\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\}\end{array}}$$

$$\dfrac{\begin{array}{l}\{\tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2))\}\\ \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\\ \{\tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\}\end{array}}{\begin{array}{l}\{\exists\tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2))\}\\ \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\\ \{\exists\tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\},\end{array}}$$

using the substitution rule, the frame rule, the rule of consequence (with the axiom that $(p \wedge q)\ *\ r \Leftrightarrow p \wedge (q\ *\ r)$ when p is pure), and the existential rule.

Exercise 7

If τ is an S-expression, then $|\tau|$, called the *flattening* of τ, is the sequence defined by:

$$\begin{aligned} |a| &= [a] \quad \text{when } a \text{ is an atom} \\ |(t_1 \cdot t_2)| &= |\tau_1| \cdot |\tau_2|. \end{aligned}$$

Here $[a]$ denotes the sequence whose only element is a, and the "$\cdot$" on the right of the last equation denotes the concatenation of sequences.

Define and prove correct (by an annotated specification of its body) a recursive procedure $\mathsf{flatten}$ that mutates a tree denoting an S-expression τ into a singly-linked list segment denoting the flattening of τ. This procedure should not do any allocation or disposal of heap storage. However, since a list segment representing $|\tau|$ contains one more two-cell than a tree representing τ, the procedure should be given as input, in addition to the tree representing τ, a single two-cell, which will become the initial cell of the list segment that is constructed.

More precisely, the procedure should satisfy

$$\begin{aligned} &\{\mathsf{tree}\,\tau\,(\mathsf{i}) \ * \ \mathsf{j} \mapsto -, -\} \\ &\mathsf{flatten}(\mathsf{i}, \mathsf{j}, \mathsf{k}) \\ &\{\mathsf{lseg}\,|\tau|\,(\mathsf{j}, \mathsf{k})\}. \end{aligned}$$

(Note that $\mathsf{flatten}$ must not assign to the variables i, j, or k.)

A Problem

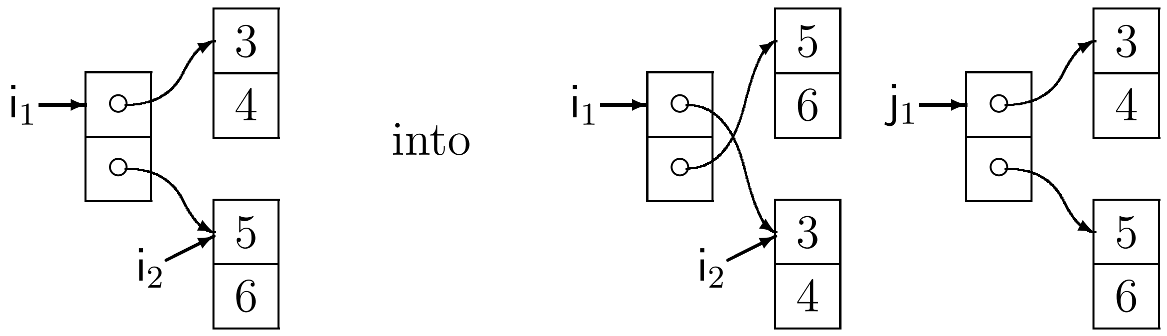
Suppose we wish to prove that

$$\{\mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{dag}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}$$

Then, we must use this specification as a hypothesis in proving that the recursive calls in the procedure body satisfy:

$$\begin{array}{l}
\{\mathsf{dag}\,\tau_1(\mathsf{i}_1) \land \mathsf{dag}\,\tau_2(\mathsf{i}_2)\} \\
\mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1) \\
\{(\mathsf{dag}\,\tau_1(\mathsf{i}_1) \land \mathsf{dag}\,\tau_2(\mathsf{i}_2))\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\} \\
\mathsf{copytree}(\mathsf{i}_2;\mathsf{j}_2) \\
\{(\mathsf{dag}\,\tau_1(\mathsf{i}_1) \land \mathsf{dag}\,\tau_2(\mathsf{i}_2))\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\ *\ \mathsf{tree}\,\tau_2(\mathsf{j}_2)\}
\end{array}$$

But the hypothesis is not strong enough to imply this. For example, suppose $\tau_1 = ((3 \cdot 4) \cdot (5 \cdot 6))$ and $\tau_2 = (5 \cdot 6)$. Then $\mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)$ might change the state from



where $\mathsf{dag}\,\tau_2\,(\mathsf{i}_2)$ is false.

Possible Solutions

1. Introduce ghost variables denoting heaps, e.g.

$$\begin{aligned}&\{\mathbf{this}(\mathsf{h}) \wedge \mathsf{dag}\,\tau(\mathsf{i})\}\\&\mathsf{copytree}(\mathsf{i};\mathsf{j})\\&\{\mathbf{this}(\mathsf{h}) \ * \ \mathsf{tree}\,\tau(\mathsf{j})\}\end{aligned}$$

2. Introduce ghost variables denoting assertions, e.g.

$$\begin{aligned}&\{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\}\\&\mathsf{copytree}(\mathsf{i};\mathsf{j})\\&\{\mathsf{p} \ * \ \mathsf{tree}\,\tau(\mathsf{j})\}\end{aligned}$$

3. Introduce passivity, i.e., the ability to assert directly that portions of the heap remain unchanged throughout the execution of a command.

We will explore the second approach.

Assertion Variables

$$\langle\text{assert}\rangle ::= \langle\text{avar}\rangle \mid \cdots$$

We generalize the concept of a state to:

$$\text{AStores}_A = A \rightarrow (\text{Heaps} \rightarrow \mathbf{B})$$

$$\text{States}_{AV} = \text{AStores}_A \times \text{Stores}_V \times \text{Heaps}.$$

Instead of $s, h \vDash p$, we now write $as, s, h \vDash p$ (when the assertion variables in p occur in the domain of as and the free ordinary variables in p occur in the domain of s).

When a is an assertion variable,

$$as, s, h \vDash a \text{ iff } \; as(a)(h).$$

The Substitution Rule (SUB) Revisited

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/a_1 \rightarrow p_1, \ldots, a_m \rightarrow p_m, v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n}$$

where $a_1, \ldots, a_m$ are the assertion variables occurring in p or q, and $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j or in any p_j.

In $\{\mathsf{a}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{a}\}$, for example, we can substitute $\mathsf{a} \rightarrow \mathsf{y} = \mathsf{z}, \mathsf{x} \rightarrow \mathsf{x}, \mathsf{y} \rightarrow \mathsf{y}$ to obtain

$$\{\mathsf{y} = \mathsf{z}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{y} = \mathsf{z}\},$$

but we cannot substitute $\mathsf{a} \rightarrow \mathsf{x} = \mathsf{z}, \mathsf{x} \rightarrow \mathsf{x}, \mathsf{y} \rightarrow \mathsf{y}$ to obtain

$$\{\mathsf{x} = \mathsf{z}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{x} = \mathsf{z}\}.$$

Copying Dags to Trees

We will prove that the procedure

$$
\begin{array}{l}
\mathsf{copytree}(\mathsf{i};\mathsf{j}) = \\
\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\
\quad\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in} \\
\quad\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\
\quad\quad\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\ \mathbf{copytree}(\mathsf{i}_2;\mathsf{j}_2)\ ;\ \mathsf{j} := \mathbf{cons}(\mathsf{j}_1,\mathsf{j}_2)\big)
\end{array}
$$

satisfies

$$\{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p} \ast \mathsf{tree}\,\tau(\mathsf{j})\}.$$

We can take p to be $\mathsf{dag}\,\tau(\mathsf{i})$, to obtain the specification

$$\{\mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{dag}\,\tau(\mathsf{i}) \ast \mathsf{tree}\,\tau(\mathsf{j})\},$$

but this is too weak to serve as a recursion hypothesis.

The Proof

$$
\begin{array}{l}
\{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\} \\
\mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then} \\
\quad \{\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = \mathsf{i}\} \\
\quad \{\mathsf{p} \ast (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp})\} \\
\quad \mathsf{j} := \mathsf{i} \\
\quad \{\mathsf{p} \ast (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{j} \wedge \mathbf{emp})\} \\
\mathbf{else} \\
\quad \vdots \\
\{\mathsf{p} \ast \mathsf{tree}\,\tau(\mathsf{j})\}
\end{array}
$$

Proof of copytree (continued)

$$
\begin{array}{l}
\{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\} \\
\quad \vdots \\
\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\,(\tau_1 \cdot \tau_2)(\mathsf{i})\} \\
\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in} \\
\qquad \big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i} + 1]\ ; \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad \mathsf{p} \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ (\mathsf{dag}\,\tau_1\,(\mathsf{i}_1) \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\} \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad \mathsf{p} \wedge (\mathbf{true}\ *\ (\mathsf{dag}\,\tau_1\,(\mathsf{i}_1) \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\} \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\quad \mathsf{p} \wedge ((\mathbf{true}\ *\ \mathsf{dag}\,\tau_1\,(\mathsf{i}_1)) \wedge (\mathbf{true}\ *\ \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\} \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2) \wedge \mathsf{dag}\,\tau_1\,(\mathsf{i}_1)\} \\
\qquad \mathbf{copytree}(\mathsf{i}_1; \mathsf{j}_1)\ ; \\
\qquad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2))\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\} \\
\qquad \mathbf{copytree}(\mathsf{i}_2; \mathsf{j}_2)\ ; \\
\qquad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2)\} \\
\qquad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2) \\
\qquad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ * \\
\qquad\quad \mathsf{j} \mapsto \mathsf{j}_1, \mathsf{j}_2\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2)\} \\
\qquad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ *\ \mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(\mathsf{j})\}\big) \\
\{\mathsf{p}\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}
\end{array}
$$

Proof of the First Recursive Call

Using the substitution rule, with the substitution

$$\begin{aligned}
\mathsf{p} &\to \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_2\, (\mathsf{i}_2) \\
\tau &\to \tau_1 \\
\mathsf{i} &\to \mathsf{i}_1 \\
\mathsf{j} &\to \mathsf{j}_1,
\end{aligned}$$

and the existential rule, we obtain

$$\dfrac{\dfrac{\{\mathsf{p} \wedge \mathsf{dag}\, \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p} \ast \mathsf{tree}\, \tau(\mathsf{j})\}}{\begin{array}{l}\{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_2\, (\mathsf{i}_2)) \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)\} \\ \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_2\, (\mathsf{i}_2)) \ast \mathsf{tree}\, \tau_1\, (\mathsf{j}_1)\}\end{array}}}{\begin{array}{l}\{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_2\, (\mathsf{i}_2)) \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)\} \\ \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\ \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_2\, (\mathsf{i}_2)) \ast \mathsf{tree}\, \tau_1\, (\mathsf{j}_1)\}.\end{array}}$$

Representing S-expressions by Back-Linked Trees

For $\tau \in$ S-exps, we define

$$\mathsf{bktree}\,\tau\,(i, b, t)$$

by structural induction:

$$\mathsf{bktree}\,a\,(i, b, t) \text{ iff } i \mapsto b, t, a, -$$

$$\begin{aligned}&\mathsf{bktree}\,(\tau_1 \cdot \tau_2)\,(i, b, t) \text{ iff}\\ &\qquad \exists i_1, i_2.\ \ i \mapsto b, t, i_1, i_2 \ * \ \mathsf{bktree}\,\tau_1\,(i_1, i, 0) \ * \ \mathsf{bktree}\,\tau_2\,(i_2, i, 1).\end{aligned}$$

Then

$$\begin{aligned}&\mathsf{copybktree}(\mathsf{i}, \mathsf{j}) =\\ &\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ [\mathsf{j} + 2] := \mathsf{i}\\ &\quad \mathbf{else\ newvar}\ \mathsf{i}', \mathsf{j}'\ \mathbf{in}\\ &\qquad \big(\mathsf{i}' := [\mathsf{i}]\ ;\ \mathsf{j}' := \mathbf{cons}(\mathsf{j}, 0, 0, 0)\ ;\ \mathsf{copybktree}(\mathsf{i}', \mathsf{j}')\ ;\ [\mathsf{j} + 2] := \mathsf{j}'\ ;\\ &\qquad \mathsf{i}' := [\mathsf{i} + 1]\ ;\ \mathsf{j}' := \mathbf{cons}(\mathsf{j}, 1, 0, 0)\ ;\ \mathsf{copybktree}(\mathsf{i}', \mathsf{j}')\ ;\ [\mathsf{j} + 3] := \mathsf{j}'\big)\end{aligned}$$

satisfies

$$\begin{aligned}&\{(\mathsf{p} \wedge \mathsf{dag}\,\tau\,(\mathsf{i}))\ *\ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\}\\ &\mathsf{copybktree}(\mathsf{i}, \mathsf{j})\\ &\{\mathsf{p}\ *\ \mathsf{bktree}\,\tau\,(\mathsf{j}, \mathsf{b}, \mathsf{t})\}\end{aligned}$$

since

$$
\begin{array}{l}
\{(\mathsf{p} \wedge \mathsf{dag}\,\tau\,(\mathsf{i})) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\} \\
\textbf{if } \mathsf{isatom}(\mathsf{i}) \textbf{ then} \\
\quad \{(\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = i) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\} \\
\quad [\mathsf{j} + 2] := \mathsf{i} \\
\quad \{(\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = i) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{i}, -\} \\
\textbf{else newvar } \mathsf{i}', \mathsf{j}' \textbf{ in} \\
\quad \Big(\{\exists \tau_1, \tau_2, \mathsf{i}_1, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2 \ * \ (\mathsf{dag}\,\tau_1\,(\mathsf{i}_1) \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\Big) * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\Big)\} \\
\quad \mathsf{i}' := [\mathsf{i}]\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto \mathsf{i}', \mathsf{i}_2 \ * \ (\mathsf{dag}\,\tau_1\,(\mathsf{i}') \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\Big) * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\Big)\} \\
\quad \mathsf{j}' := \mathbf{cons}(\mathsf{j}, 0, 0, 0)\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto \mathsf{i}', \mathsf{i}_2 \ * \ (\mathsf{dag}\,\tau_1\,(\mathsf{i}') \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)))\Big) * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, - \\
\qquad\qquad * \ \mathsf{j}' \mapsto \mathsf{j}, 0, -, -\Big)\} \\
\quad \{\exists \tau_1, \tau_2, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto -, \mathsf{i}_2 \ * \ \mathsf{dag}\,\tau_2\,(\mathsf{i}_2)) \wedge \mathsf{dag}\,\tau_1\,(\mathsf{i}')\Big) * \ \mathsf{j}' \mapsto \mathsf{j}, 0, -, - \\
\qquad\qquad * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\Big)\} \\
\quad \mathsf{copybktree}(\mathsf{i}', \mathsf{j}')\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\quad\ \Big(\Big(p \wedge (\mathsf{i} \mapsto -, \mathsf{i}_2 \ * \ \mathsf{dag}\,\tau_2\,(\mathsf{i}_2))\Big) * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}', \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\Big)\} \\
\quad [\mathsf{j} + 2] := \mathsf{j}'\ ; \\
\qquad \vdots \\
\{\mathsf{p} \ * \ \mathsf{bktree}\,\tau\,(\mathsf{j}, \mathsf{b}, \mathsf{t})\}.
\end{array}
$$

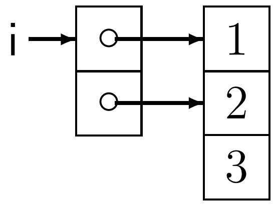
$$
\begin{array}{l}
\{(\mathsf{p} \wedge \mathsf{dag}\,\tau\,(\mathsf{i})) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, -, -\} \\
\qquad \vdots \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\quad\ \Big(\Big(p \wedge (\mathsf{i} \mapsto -, \mathsf{i}_2 \ * \ \mathsf{dag}\,\tau_2\,(\mathsf{i}_2))\Big) * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, -\Big)\} \\
\quad \mathsf{i}' := [\mathsf{i} + 1]\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto -, \mathsf{i}' \ * \ \mathsf{dag}\,\tau_2\,(\mathsf{i}'))\Big) * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, -\Big)\} \\
\quad \mathsf{j}' := \mathbf{cons}(\mathsf{j}, 1, 0, 0)\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge (\mathsf{i} \mapsto -, \mathsf{i}' \ * \ \mathsf{dag}\,\tau_2\,(\mathsf{i}'))\Big) * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, - \\
\qquad\qquad * \ \mathsf{j}' \mapsto \mathsf{j}, 1, -, -\Big)\} \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad \Big(\Big(p \wedge \mathsf{dag}\,\tau_2\,(\mathsf{i}')\Big) * \ \mathsf{j}' \mapsto \mathsf{j}, 1, -, - \\
\qquad\qquad * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, -\Big)\} \\
\quad \mathsf{copybktree}(\mathsf{i}', \mathsf{j}')\ ; \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (p \ * \ \mathsf{bktree}\,\tau_2\,(\mathsf{j}', \mathsf{j}, 1) \ * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, -)\} \\
\quad [\mathsf{j} + 3] := \mathsf{j}' \\
\quad \{\exists \tau_1, \tau_2, \mathsf{j}_1, \mathsf{j}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (p \ * \ \mathsf{bktree}\,\tau_2\,(\mathsf{j}_2, \mathsf{j}, 1) \ * \ \mathsf{bktree}\,\tau_1\,(\mathsf{j}_1, \mathsf{j}, 0) \ * \ \mathsf{j} \mapsto \mathsf{b}, \mathsf{t}, \mathsf{j}_1, \mathsf{j}_2)\}\Big) \\
\{\mathsf{p} \ * \ \mathsf{bktree}\,\tau\,(\mathsf{j}, \mathsf{b}, \mathsf{t})\}.
\end{array}
$$

Skewed Sharing

Our definition of dag permits *skewed sharing.* For example,

$$\mathsf{dag}\,((1 \cdot 2) \cdot (2 \cdot 3))\,(\mathsf{i})$$

holds when



This causes difficulties with algorithms for modifying or copying dags (while preserving structure).

A Possible Solution:

- We add to the state a mapping ϕ from the domain of the heap to natural numbers, called the field count.

- When $x := \mathbf{cons}(e_1, \ldots, e_n)$ sets x to the address a. the field count is extended so that

  $$\phi(a) = n \qquad \phi(a+1) = 0 \qquad \cdots \qquad \phi(a+n-1) = 0.$$

- We introduce $\stackrel{!}{\mapsto}$ such that $\phi, s, h \vDash e \stackrel{!}{\mapsto} e_1, \ldots, e_n$ iff

  $$s, h \vDash e \mapsto e_1, \ldots, e_n \text{ and } \phi([\![e]\!]_{\mathrm{exp}} s) = n \text{ and}$$
  $$\phi([\![e]\!]_{\mathrm{exp}} s + 1) = 0 \text{ and } \ldots \text{ and } \phi([\![e]\!]_{\mathrm{exp}} s + n - 1) = 0.$$

- Then

  $$e \stackrel{!}{\hookrightarrow} e_1, \ldots, e_m \wedge e' \stackrel{!}{\hookrightarrow} e'_1, \ldots, e'_n \wedge e \neq e' \Rightarrow$$
  $$e \stackrel{!}{\mapsto} e_1, \ldots, e_m \ast e' \stackrel{!}{\mapsto} e'_1, \ldots, e'_n \ast \mathbf{true}.$$

Proving the Schorr-Waite Marking Algorithm (Yang)

- We abandon address arithmetic, and require all records to contain two address fields and two boolean fields.
- Only reachable cells are in heap.

Let

$$\begin{aligned}
\mathsf{allocated}(\mathsf{x}) &\stackrel{\text{def}}{=} \mathsf{x} \hookrightarrow -,-,-,- \\
\mathsf{markedR} &\stackrel{\text{def}}{=} \forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \mathsf{x} \hookrightarrow -,-,-,\mathbf{true} \\
\mathsf{noDangling}(\mathsf{x}) &\stackrel{\text{def}}{=} (\mathsf{x} = \mathbf{nil}) \vee \mathsf{allocated}(\mathsf{x}) \\
\mathsf{noDanglingR} &\stackrel{\text{def}}{=} \forall \mathsf{x}, \mathsf{l}, \mathsf{r}.\ (\mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, -, -) \Rightarrow \\
&\qquad\qquad \mathsf{noDangling}(\mathsf{l}) \wedge \mathsf{noDangling}(\mathsf{r}).
\end{aligned}$$

Then the invariant of the program is

$$\begin{aligned}
&\mathsf{noDanglingR} \wedge \mathsf{noDangling}(\mathsf{t}) \wedge \mathsf{noDangling}(\mathsf{p}) \wedge \\
&\big(\mathsf{listMarkedNodesR}(\mathsf{stack}, \mathsf{p}) \ * \\
&\quad (\mathsf{restoredListR}(\mathsf{stack}, \mathsf{t}) \mathrel{-\!\!*} \mathsf{spansR}(\mathsf{STree}, \mathsf{root}))\big) \wedge \\
&\big(\mathsf{markedR} \ * \ \big(\mathsf{unmarkedR} \wedge \big(\forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \\
&\quad (\mathsf{reach}(\mathsf{t}, \mathsf{x}) \vee \mathsf{reachRightChildInList}(\mathsf{stack}, \mathsf{x}))\big)\big)\big).
\end{aligned}$$

Shared-Variable Concurrency (O'Hearn and Brookes)

We extend our programming language with:

$$
\begin{aligned}
\langle\text{comm}\rangle ::= {} & \langle\text{comm}\rangle \parallel \langle\text{comm}\rangle \\
& \mid \textbf{with}\ \langle\text{resource}\rangle\ \textbf{when}\ \langle\text{boolexp}\rangle\ \textbf{do}\ \langle\text{comm}\rangle \\
& \mid \textbf{resource}\ \langle\text{resource}\rangle\ \textbf{in}\ \langle\text{comm}\rangle
\end{aligned}
$$

and we generalize specifications by adding contexts:

$$\overbrace{r_1(X_1)\colon R_1, \ldots, r_n(X_n)\colon R_n}^{\Gamma} \vdash \{p\}\ c\ \{q\}.$$

Such a specification is *well-formed* iff:

- The r_i are distinct resources.
- The X_i are disjoint sets of *protected* variables.
- Each R_i is an assertion called the *invariant* of r_i.
- When $i \neq j$, no free variable of R_i belongs to X_j.
- The free variables of p and q do not belong to any X_i.
- Variables in an X_i can only occur free in c within a critical region for resource r_i.
- Variables occurring free in an R_i can only be modified by c within a critical region for resource r_i.
- Each R_i is a *precise* assertion, i.e., every heap contains at most one subheap that satisfies the assertion.

Inference Rules for Concurrency

In our previous rules, we prefix "$\Gamma \vdash$" to all specifications, and require their instances to be well-formed. Then we add:

$$\frac{\Gamma \vdash \{p_1\}\ c_1\ \{q_1\} \qquad \Gamma \vdash \{p_2\}\ c_2\ \{q_2\}}{\Gamma \vdash \{p_1 \ast p_2\}\ c_1 \parallel c_2\ \{q_1 \ast q_2\},}$$

when

- The free variables of p_1 and q_1 are not modified by c_2, and vice-versa,
- The variables that are free in c_1 and modified by c_2 (or vice-versa) belong to $\mathit{owned}(\Gamma)$.

$$\frac{\Gamma \vdash \{(p \ast R) \wedge b\}\ c\ \{q \ast R\}}{\Gamma, r(X) \colon R \vdash \{p\}\ \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c\ \{q\}.} \qquad \text{(CR)}$$

$$\frac{\Gamma, r(X) \colon R \vdash \{p\}\ c\ \{q\}}{\Gamma \vdash \{p \ast R\}\ \mathbf{resource}\ r\underline{(X) \colon R}\ \mathbf{in}\ c\ \{q \ast R\}.}$$

A Pointer Buffer

Let

$$R \stackrel{\text{def}}{=} (\mathsf{full} \land \mathsf{c} \mapsto -,-) \lor (\neg\, \mathsf{full} \land \mathbf{emp})$$

Then

$$
\begin{array}{l}
\{p_1 \ * \ p_2\} \\
\mathsf{full} := \mathbf{false}\ ; \\
\{p_1 \ * \ p_2 \ * \ R\} \\
\mathbf{resource}\ \mathsf{buff}(\mathsf{c}, \mathsf{full})\colon R\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \{p_1 \ * \ p_2\} \\
\qquad \left[\begin{array}{lcl}
\{p_1\} & & \{p_2\} \\
\quad\vdots & & \quad\vdots \\
\{p_1'\} & & \{p_2'\} \\
\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{b})\ ; & & \mathit{GETY}\ ; \\
\{p_1' \ * \ \mathsf{x} \mapsto -,-\} & \| & \{p_2' \ * \ \mathsf{y} \mapsto -,-\} \\
\mathit{PUTX}\ ; & & \mathbf{dispose}\ \mathsf{y}\ ; \\
\{p_1'\} & & \{p_2'\} \\
\quad\vdots & & \quad\vdots \\
\{q_1\} & & \{q_2\}
\end{array}\right] \\
\qquad\qquad\qquad\qquad \{q_1 \ * \ q_2\} \\
\{q_1 \ * \ q_2 \ * \ R\},
\end{array}
$$

A Pointer Buffer (continued)

where

$$
\begin{array}{rl}
& \{\mathsf{x} \mapsto -,-\} \\
PUTX \equiv & \textbf{with } \mathsf{buff} \textbf{ when } \neg\,\mathsf{full} \textbf{ do} \\
& \qquad \{(\mathsf{x} \mapsto -,- \ * \ R) \wedge \neg\,\mathsf{full}\} \\
& \qquad \{\mathsf{x} \mapsto -,- \ * \ (\neg\,\mathsf{full} \wedge \mathbf{emp})\} \\
& \qquad \{\mathsf{x} \mapsto -,-\} \\
& \qquad (\mathsf{c} := \mathsf{x}\,;\mathsf{full} := \mathbf{true})\,; \\
& \qquad \{\mathsf{full} \wedge \mathsf{c} \mapsto -,-\} \\
& \qquad \{R\} \\
& \qquad \{R \ * \ \mathbf{emp}\} \\
& \{\mathbf{emp}\},
\end{array}
$$

$$
\begin{array}{rl}
& \{\mathbf{emp}\} \\
GETY \equiv & \textbf{with } \mathsf{buff} \textbf{ when } \mathsf{full} \textbf{ do} \\
& \qquad \{(\mathbf{emp} \ * \ R) \wedge \mathsf{full}\} \\
& \qquad \{\mathsf{full} \wedge \mathsf{c} \mapsto -,-\} \\
& \qquad (\mathsf{y} := \mathsf{c}\,;\mathsf{full} := \mathbf{false})\,; \\
& \qquad \{\neg\,\mathsf{full} \wedge \mathsf{y} \mapsto -,-\} \\
& \qquad \{(\neg\,\mathsf{full} \wedge \mathbf{emp}) \ * \ \mathsf{y} \mapsto -,-\} \\
& \qquad \{R \ * \ \mathsf{y} \mapsto -,-\} \\
& \{\mathsf{y} \mapsto -,-\}.
\end{array}
$$

A Simple Memory Allocator

We write $\textbf{list}\,\mathsf{f}$ to abbreviate $\exists\alpha.\ \textbf{list}\,\alpha\,\mathsf{f}$.

$$
\begin{array}{l}
\{p_1 \ast p_2\} \\
\mathsf{f} := \textbf{nil}\ ; \\
\{p_1 \ast p_2 \ast \textbf{list}\,\mathsf{f}\} \\
\textbf{resource}\ \mathsf{mm}(\mathsf{f}) \colon \textbf{list}\,\mathsf{f}\ \textbf{in}
\end{array}
$$

$$
\begin{array}{c}
\{p_1 \ast p_2\} \\
\left[\begin{array}{lcl}
\{p_1\} & & \{p_2\} \\
\quad\vdots & & \quad\vdots \\
\{p_1'\} & & \{p_2'\} \\
\mathit{ALLOCX}\ ; & & \mathit{ALLOCY}\ ; \\
\{p_1' \ast \mathsf{x} \mapsto -,-\} & & \{p_2' \ast \mathsf{y} \mapsto -,-\} \\
\quad\vdots & \| & \quad\vdots \\
\{p_1'' \ast \mathsf{x} \mapsto -,-\} & & \{p_2'' \ast \mathsf{y} \mapsto -,-\} \\
\mathit{FREEX}\ ; & & \mathit{FREEY}\ ; \\
\{p_1''\} & & \{p_2''\} \\
\quad\vdots & & \quad\vdots \\
\{q_1\} & & \{q_2\}
\end{array}\right] \\
\{q_1 \ast q_2\}
\end{array}
$$

$$\{q_1 \ast q_2 \ast \textbf{list}\,\mathsf{f}\},$$

A Simple Memory Allocator (continued)

where

$$
\begin{aligned}
& \{\mathbf{emp}\} \\
\mathit{ALLOCX} \equiv\ & \mathbf{with}\ \mathsf{mm}\ \mathbf{when\ true\ do} \\
& \qquad \{\mathbf{emp} \ast \mathbf{list}\,\mathsf{f}\} \\
& \qquad \mathbf{if}\ \mathsf{f} = \mathbf{nil}\ \mathbf{then} \\
& \qquad\qquad \mathsf{x} := \mathbf{cons}(0, 0) \\
& \qquad \mathbf{else} \\
& \qquad\qquad (\mathsf{x} := \mathsf{f}\,;\mathsf{f} := [\mathsf{f} + 1]) \\
& \qquad \{\mathsf{x} \mapsto -, - \ast \mathbf{list}\,\mathsf{f}\} \\
& \{\mathsf{x} \mapsto -, -\},
\end{aligned}
$$

$$
\begin{aligned}
& \{\mathsf{x} \mapsto -, -\} \\
\mathit{FREEX} \equiv\ & \mathbf{with}\ \mathsf{mm}\ \mathbf{when\ true\ do} \\
& \qquad \{\mathsf{x} \mapsto -, - \ast \mathbf{list}\,\mathsf{f}\} \\
& \qquad ([\mathsf{x} + 1] := \mathsf{f}\,;\mathsf{f} := \mathsf{x}) \\
& \qquad \{\mathbf{emp} \ast \mathbf{list}\,\mathsf{f}\} \\
& \{\mathbf{emp}\},
\end{aligned}
$$

and similarly for *ALLOCY* and *FREEY*.

Why Resource Invariants must be Precise

Let **one** $\stackrel{\text{def}}{=} 10 \mapsto -$, and let r be a resource with the invariant $R \stackrel{\text{def}}{=}$ **true**, which is not precise. Then

1. $\{\mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{true}\}$ (SK)
2. $\{(\mathbf{emp} \vee \mathbf{one}) * \mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{true}\}$ (SP,1)
3. $\{(\mathbf{emp} \vee \mathbf{one}) * \mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{emp} * \mathbf{true}\}$ (WC,2)
4. $\{\mathbf{emp} \vee \mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{emp}\}$ (CR,3)
5. $\{\mathbf{emp}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{emp}\}$ (SP,4)
6. $\{\mathbf{emp} * \mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{emp} * \mathbf{one}\}$ (FR,5)
7. $\{\mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{emp} * \mathbf{one}\}$ (SP,6)
8. $\{\mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{one}\}$ (WC,7)
9. $\{\mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{emp}\}$ (SP,4)
10. $\{\mathbf{one} \wedge \mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{one} \wedge \mathbf{emp}\}$ (CONJ,8,9)
11. $\{\mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{one} \wedge \mathbf{emp}\}$ (SP,10)
12. $\{\mathbf{one}\}\ \mathbf{with}\ \mathsf{r}\ \mathbf{when\ true\ do\ skip}\ \{\mathbf{false}\}$ (WC,11)

Fractional Permissions (Bornat, following Boyland)

We write $e \mapsto_z e'$, where z is a real number such that $0 < z \leq 1$, to indicate that e points to e' with permission z.

- $e \mapsto_1 e'$ is the same as $e \mapsto e'$, so that a permission of one allows all operations.
- Only lookup is allowed when $z < 1$.

Then

$$e \mapsto_z e' \ * \ e \mapsto_{z'} e' \text{ iff } e \mapsto_{z+z'} e'$$

and

$$\begin{array}{ccc}
\{\mathbf{emp}\} & v := \mathbf{cons}(e_1, \ldots, e_n) & \{e \mapsto_1 e_1, \ldots, e_n\} \\
\{e \mapsto_1 -\} & \mathbf{dispose}(e) & \{\mathbf{emp}\} \\
\{e \mapsto_1 -\} & [e] := e' & \{e \mapsto_1 e'\} \\
\{e \mapsto_z e'\} & v := [e] & \{e \mapsto_z e' \wedge v = e'\},
\end{array}$$

with appropriate restrictions on variable occurrences.