An Introduction
to Separation Logic

February 16, 2011

Chapter 6

Iterated Separating Conjunction

In this chapter, we introduce an iterative version of the separating conjunction that is useful in describing arrays, as well as certain properties of list structures.

6.1 A New Form of Assertion

We extend the language of assertions with an binding operator $\odot$, which is used to construct an assertion of the form

$$\odot_{v=e}^{e'} p,$$

where the occurrence of v in the subscript is a binder whose scope is p. Roughly speaking, this assertion describes the separating conjunction

$$(p/v \to e) \ * \ (p/v \to e+1) \ * \ \cdots \ * \ (p/v \to e').$$

More precisely, for a state s, h, let $m = [\![e]\!]_{\text{exp}} s$ and $n = [\![e']\!]_{\text{exp}} s$ be the *lower* and *upper bounds*, and $I = \{\, i \mid m \le i \le n \,\}$ be the set of *indices*. Then $s, h \models \odot_{v=e}^{e'} p$ iff there is a function $H \in I \to \text{Heaps}$ that partitions h into an indexed set of heaps,

$$h = \bigcup \{\, Hi \mid i \in I \,\} \text{ and } \forall i, j \in I.\ i \neq j \text{ implies } Hi \perp Hj,$$

such that, for all indices $i \in I$, $[\, s \mid v{:}\, i \,], Hi \models p$.

This new form satisfies the following axiom schemata, in which, for readability, we have written $p(e)$ for $p/i \to e$:

$$m > n \;\Rightarrow\; \left(\textstyle\bigodot_{i=m}^{n} p(i) \Leftrightarrow \mathbf{emp}\right) \tag{6.1}$$

$$m = n \;\Rightarrow\; \left(\textstyle\bigodot_{i=m}^{n} p(i) \Leftrightarrow p(m)\right) \tag{6.2}$$

$$k \le m \le n+1 \;\Rightarrow\; \left(\textstyle\bigodot_{i=k}^{n} p(i) \Leftrightarrow \left(\bigodot_{i=k}^{m-1} p(i) \;*\; \bigodot_{i=m}^{n} p(i)\right)\right) \tag{6.3}$$

$$\textstyle\bigodot_{i=m}^{n} p(i) \Leftrightarrow \bigodot_{i=m-k}^{n-k} p(i+k) \tag{6.4}$$

$$m \le n \;\Rightarrow\; \left(\left(\textstyle\bigodot_{i=m}^{n} p(i)\right) \;*\; q \Leftrightarrow \bigodot_{i=m}^{n} (p(i) \;*\; q)\right) \tag{6.5}$$
when q is pure and $i \notin \mathrm{FV}(q)$

$$m \le n \;\Rightarrow\; \left(\left(\textstyle\bigodot_{i=m}^{n} p(i)\right) \wedge q \Leftrightarrow \bigodot_{i=m}^{n} (p(i) \wedge q)\right) \tag{6.6}$$
when q is pure and $i \notin \mathrm{FV}(q)$

$$m \le j \le n \;\Rightarrow\; \left(\left(\textstyle\bigodot_{i=m}^{n} p(i)\right) \Rightarrow (p(j) \;*\; \mathbf{true})\right). \tag{6.7}$$

6.2 Arrays

The most obvious use of the iterated separating conjunction is to describe arrays that occur in the heap. To allocate such arrays, as discussed in Section 1.8, we introduce the command

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

The effect of $v := \mathbf{allocate}\ e$ is to allocate a block of size e, and to assign the address of the first element to v. The initialization of the array elements is not specified. For example:

	Store :	$\mathsf{x}\colon 3,\ \mathsf{y}\colon 4$
	Heap :	empty
$\mathsf{x} := \mathbf{allocate}\ \mathsf{y}$		$\Downarrow$
	Store :	$\mathsf{x}\colon 37,\ \mathsf{y}\colon 4$
	Heap :	$37\colon -,\ 38\colon -, 39\colon -,\ 40\colon -.$

The inference rules for this new command are similar to those for the ordinary allocation of records:

- The local nonoverwriting form (ALLOCNOL)

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

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

- The global nonoverwriting form (ALLOCNOG)

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

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

- The local form (ALLOCL)

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

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

- The global form (ALLOCG)

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

  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 (ALLOCBR)

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

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

Usually, (one-dimensional) arrays are used to represent sequences. We define

$$\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b}) \stackrel{\text{def}}{=} \#\alpha = \mathsf{b}-\mathsf{a}+1 \wedge \bigodot_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i} \mapsto \alpha_{\mathsf{i}-\mathsf{a}+1}.$$

When $\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b})$ holds, we say that a to b (more precisely, the heap from a to b) represents the sequence α.

Notice that, since the length of a sequence is never negative, the assertion $\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b})$ implies that $\mathsf{a} \leq \mathsf{b}+1$. In fact, it would be consistent to define a to b to represent the empty sequence when $\mathsf{a} > \mathsf{b}+1$ (as well as $\mathsf{a} = \mathsf{b}+1$), but we will not use such "irregular" representations in these notes. (An integrated approach to regular and irregular represetations is discussed in Reference [101, Chapter 2].)

This new form of assertion satisfies the following axioms:

$$\begin{aligned}
&\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b}) \Rightarrow \#\alpha = \mathsf{b}-\mathsf{a}+1\\
&\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b}) \Rightarrow \mathsf{i} \hookrightarrow \alpha_{\mathsf{i}-\mathsf{a}+1} \quad \text{when } \mathsf{a} \leq \mathsf{i} \leq \mathsf{b}\\
&\mathsf{array}\,\epsilon\,(\mathsf{a},\mathsf{b}) \Leftrightarrow \mathsf{b} = \mathsf{a}-1 \wedge \mathbf{emp}\\
&\mathsf{array}\,\mathsf{x}\,(\mathsf{a},\mathsf{b}) \Leftrightarrow \mathsf{b} = \mathsf{a} \wedge \mathsf{a} \mapsto \mathsf{x}\\
&\mathsf{array}\,\mathsf{x}{\cdot}\alpha\,(\mathsf{a},\mathsf{b}) \Leftrightarrow \mathsf{a} \mapsto \mathsf{x} \;*\; \mathsf{array}\,\alpha\,(\mathsf{a}+1,\mathsf{b})\\
&\mathsf{array}\,\alpha{\cdot}\mathsf{x}\,(\mathsf{a},\mathsf{b}) \Leftrightarrow \mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b}-1) \;*\; \mathsf{b} \mapsto \mathsf{x}\\
&\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{c}) \;*\; \mathsf{array}\,\beta\,(\mathsf{c}+1,\mathsf{b})\\
&\qquad\qquad \Leftrightarrow \mathsf{array}\,\alpha{\cdot}\beta\,(\mathsf{a},\mathsf{b}) \wedge \mathsf{c} = \mathsf{a}+\#\alpha-1\\
&\qquad\qquad \Leftrightarrow \mathsf{array}\,\alpha{\cdot}\beta\,(\mathsf{a},\mathsf{b}) \wedge \mathsf{c} = \mathsf{b}-\#\beta.
\end{aligned}$$

6.3 Partition

As an example, we present a program that, given an array representing a sequence, and a *pivot* value r, rearranges the sequence so that it splits into two contiguous parts, containing values smaller or equal to r and values larger than r, respectively. (This is a variant of the well-known program "Partition" by C. A. R. Hoare [108].)

$\{\mathsf{array}\ \alpha(\mathsf{a},\mathsf{b})\}$

$\mathbf{newvar}\ \mathsf{d},\mathsf{x},\mathsf{y}\ \mathbf{in}\ \big(\mathsf{c} := \mathsf{a}-1\ ;\ \mathsf{d} := \mathsf{b}+1\ ;$

$\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{d}-1)\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b}))$

$\quad\wedge\ \alpha_1{\cdot}\alpha_2{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r}\}$

$\mathbf{while}\ \mathsf{d} > \mathsf{c}+1\ \mathbf{do}\ \big(\mathsf{x} := [\mathsf{c}+1];$

$\quad\mathbf{if}\ \mathsf{x} \leq \mathsf{r}\ \mathbf{then}$

$\qquad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{c}+1 \mapsto \mathsf{x}\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+2,\mathsf{d}-1)$

$\qquad\quad *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b})) \wedge \alpha_1{\cdot}\mathsf{x}{\cdot}\alpha_2{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1{\cdot}\mathsf{x}\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r}\}$

$\qquad\mathsf{c} := \mathsf{c}+1$

$\quad\mathbf{else}\ \big(\mathsf{y} := [\mathsf{d}-1];$

$\qquad\mathbf{if}\ \mathsf{y} > \mathsf{r}\ \mathbf{then}$

$\qquad\quad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{d}-2)\ *\ \mathsf{d}-1 \mapsto \mathsf{y}$

$\qquad\quad\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b})) \wedge \alpha_1{\cdot}\alpha_2{\cdot}\mathsf{y}{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\mathsf{y}{\cdot}\alpha_3\} >^* \mathsf{r}\}$

$\qquad\quad\mathsf{d} := \mathsf{d}-1$

$\qquad\mathbf{else}$

$\qquad\quad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{c}+1 \mapsto \mathsf{x}$

$\qquad\quad\quad *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+2,\mathsf{d}-2)\ *\ \mathsf{d}-1 \mapsto \mathsf{y}\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b}))$

$\qquad\quad\quad\wedge\ \alpha_1{\cdot}\mathsf{x}{\cdot}\alpha_2{\cdot}\mathsf{y}{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r} \wedge \mathsf{x} > \mathsf{r} \wedge \mathsf{y} \leq \mathsf{r}\}$

$\qquad\quad([\mathsf{c}+1] := \mathsf{y}\ ;\ [\mathsf{d}-1] := \mathsf{x}\ ;\ \mathsf{c} := \mathsf{c}+1\ ;\ \mathsf{d} := \mathsf{d}-1)\big)\big)\big)$

$\{\exists\alpha_1,\alpha_2.\ (\mathsf{array}\ \alpha_1(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2(\mathsf{c}+1,\mathsf{b}))$

$\quad\wedge\ \alpha_1{\cdot}\alpha_2 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}.$

For the most part, the reasoning here is straightforward. It should be noticed, however, that the assertion following the second **else** requires $\mathsf{c}+1 < \mathsf{d}-1$, which depends upon the **while**-test plus the validity of the implication

$$\mathsf{c}+1 \hookrightarrow \mathsf{x} \wedge \mathsf{d}-1 \hookrightarrow \mathsf{y} \wedge \mathsf{x} > \mathsf{r} \wedge \mathsf{y} \leq \mathsf{r} \Rightarrow \mathsf{c}+1 \neq \mathsf{d}-1$$

(which holds since the ordering relations imply that x and y must be distinct, which in turn implies that $c+1$ and $d-1$ must be distinct).

It is also straightforward to encapsulate the above program as a nonrecursive procedure. If we define

$$\begin{array}{l}\mathsf{partition}(\mathsf{c};\mathsf{a},\mathsf{b},\mathsf{r}) = \\ \quad \textbf{newvar}\ \mathsf{d},\mathsf{x},\mathsf{y}\ \textbf{in}\ \big(\mathsf{c} := \mathsf{a}-1\ ;\ \mathsf{d} := \mathsf{b}+1\ ; \\ \quad \textbf{while}\ \mathsf{d} > \mathsf{c}+1\ \textbf{do} \\ \quad\quad \big(\mathsf{x} := [\mathsf{c}+1]\ ;\ \textbf{if}\ \mathsf{x} \le \mathsf{r}\ \textbf{then}\ \mathsf{c} := \mathsf{c}+1\ \textbf{else} \\ \quad\quad\quad \big(\mathsf{y} := [\mathsf{d}-1]\ ;\ \textbf{if}\ \mathsf{y} > \mathsf{r}\ \textbf{then}\ \mathsf{d} := \mathsf{d}-1\ \textbf{else} \\ \quad\quad\quad\quad ([\mathsf{c}+1] := \mathsf{y}\ ;\ [\mathsf{d}-1] := \mathsf{x}\ ;\ \mathsf{c} := \mathsf{c}+1\ ;\ \mathsf{d} := \mathsf{d}-1)\big)\big)\big),\end{array}$$

then partition satisfies

$$\begin{array}{rl} H_{\mathsf{partition}} \stackrel{\text{def}}{=} & \{\mathsf{array}\ \alpha(\mathsf{a},\mathsf{b})\} \\ & \mathsf{partition}(\mathsf{c};\mathsf{a},\mathsf{b},\mathsf{r})\{\alpha\} \\ & \{\exists\alpha_1,\alpha_2.\ (\mathsf{array}\ \alpha_1(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2(\mathsf{c}+1,\mathsf{b})) \\ & \quad \wedge\ \alpha_1{\cdot}\alpha_2 \sim \alpha \wedge \{\alpha_1\} \le^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}. \end{array}$$

6.4 From Partition to Quicksort

We can use the procedure partition to define a version of the recursive sorting procedure called quicksort (which is again a variant of a well-known algorithm by Hoare [109]).

Since quicksort is recursive, we must state the specification of the procedure before we prove the correctness of its body, in order to reason about the recursive calls within the body. Fortunately, the specification is an obvious formalization of the requirements for sorting: We assume the specification

$$\begin{array}{rl} H_{\mathsf{quicksort}} \stackrel{\text{def}}{=} & \{\mathsf{array}\ \alpha\ (\mathsf{a},\mathsf{b})\} \\ & \mathsf{quicksort}(;\mathsf{a},\mathsf{b})\{\alpha\} \\ & \{\exists\beta.\ \mathsf{array}\ \beta\ (\mathsf{a},\mathsf{b}) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta\}. \end{array} \tag{6.8}$$

(Notice that quicksort does not modify any variables.)

The basic idea behind quicksort is straightforward: One chooses a pivot value, partitions the array to be sorted into segments that are smaller or equal to the pivot and larger or equal to the pivot. Then one uses recursive calls to sort the two segments.

In our version, there is a complication because it is possible that one of the segments produced by the procedure partition will be empty while the other is the entire array to be sorted, in which case a naive version of quicksort will never terminate. (Consider, for example, the case where all elements of the array have the same value.) To circumvent this problem, we sort the end elements of the array separately, use their mean as the pivot value, and apply partition only to the interior of the array, so that the division of the entire array always has at least one element in each segment.

Then the following is an annotated specification of the body of quicksort:

$$
\begin{array}{l}
H_{\mathsf{partition}}, H_{\mathsf{quicksort}} \vdash \\
\quad \{\mathsf{array}\ \alpha\ (\mathsf{a}, \mathsf{b})\} \\
\quad \textbf{if } \mathsf{a} < \mathsf{b} \textbf{ then newvar } \mathsf{c} \textbf{ in} \\
\quad \Big(\{\exists \mathsf{x}_1, \alpha_0, \mathsf{x}_2.\ (\mathsf{a} \mapsto \mathsf{x}_1\ *\ \mathsf{array}\ \alpha_0(\mathsf{a}+1, \mathsf{b}-1)\ *\ \mathsf{b} \mapsto \mathsf{x}_2) \\
\qquad \wedge\ \mathsf{x}_1 \cdot \alpha_0 \cdot \mathsf{x}_2 \sim \alpha\} \\
\quad \textbf{newvar } \mathsf{x1}, \mathsf{x2}, \mathsf{r} \textbf{ in} \\
\quad\quad \Big(\mathsf{x1} := [\mathsf{a}]\ ;\ \mathsf{x2} := [\mathsf{b}]\ ; \\
\quad\quad \textbf{if } \mathsf{x1} > \mathsf{x2} \textbf{ then } ([\mathsf{a}] := \mathsf{x2}\ ;\ [\mathsf{b}] := \mathsf{x1}) \textbf{ else skip}\ ; \\
\quad\quad \mathsf{r} := (\mathsf{x1} + \mathsf{x2}) \div 2\ ; \\
\quad\quad \{\exists \mathsf{x}_1, \alpha_0, \mathsf{x}_2.\ (\mathsf{a} \mapsto \mathsf{x}_1\ *\ \mathsf{array}\ \alpha_0(\mathsf{a}+1, \mathsf{b}-1)\ *\ \mathsf{b} \mapsto \mathsf{x}_2) \\
\qquad\quad \wedge\ \mathsf{x}_1 \cdot \alpha_0 \cdot \mathsf{x}_2 \sim \alpha \wedge \mathsf{x}_1 \le \mathsf{r} \le \mathsf{x}_2\} \\
\quad\quad \left.\begin{array}{l}
\{\mathsf{array}\ \alpha_0(\mathsf{a}+1, \mathsf{b}-1)\} \\
\mathsf{partition}(\mathsf{c}; \mathsf{a}+1, \mathsf{b}-1, \mathsf{r})\{\alpha_0\} \\
\{\exists \alpha_1, \alpha_2.\ (\mathsf{array}\ \alpha_1(\mathsf{a}+1, \mathsf{c}) \\
\qquad\quad *\ \mathsf{array}\ \alpha_2(\mathsf{c}+1, \mathsf{b}-1)) \\
\qquad \wedge\ \alpha_1 \cdot \alpha_2 \sim \alpha_0 \\
\qquad \wedge\ \{\alpha_1\} \le^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}
\end{array}\right\} * \left\{\begin{array}{l}
(\mathsf{a} \mapsto \mathsf{x}_1 * \mathsf{b} \mapsto \mathsf{x}_2) \\
\wedge\ \mathsf{x}_1 \cdot \alpha_0 \cdot \mathsf{x}_2 \sim \alpha \\
\wedge\ \mathsf{x}_1 \le \mathsf{r} \le \mathsf{x}_2
\end{array}\right\} \exists \mathsf{x}_1, \alpha_0, \mathsf{x}_2 \\
\quad\quad \{\exists \mathsf{x}_1, \alpha_1, \alpha_2, \mathsf{x}_2. \\
\qquad (\mathsf{a} \mapsto \mathsf{x}_1 * \mathsf{array}\ \alpha_1(\mathsf{a}+1, \mathsf{c}) * \mathsf{array}\ \alpha_2(\mathsf{c}+1, \mathsf{b}-1) * \mathsf{b} \mapsto \mathsf{x}_2) \\
\qquad \wedge\ \mathsf{x}_1 \cdot \alpha_1 \cdot \alpha_2 \cdot \mathsf{x}_2 \sim \alpha \wedge \mathsf{x}_1 \le \mathsf{r} \le \mathsf{x}_2 \wedge \{\alpha_1\} \le^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}\Big)\ ; \\
\{\exists \alpha_1, \alpha_2.\ (\mathsf{array}\ \alpha_1\ (\mathsf{a}, \mathsf{c})\ *\ \mathsf{array}\ \alpha_2\ (\mathsf{c}+1, \mathsf{b})) \wedge \alpha_1 \cdot \alpha_2 \sim \alpha \wedge \{\alpha_1\} \le^* \{\alpha_2\}\} \\
\quad\quad \vdots
\end{array}
$$

$$
\begin{array}{l}
\quad\vdots \\
\{\exists \alpha_1, \alpha_2.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{b})) \\
\qquad \wedge\, \alpha_1{\cdot}\alpha_2 \sim \alpha \wedge \{\alpha_1\} \leq^* \{\alpha_2\}\} \\
\left.\begin{array}{l}
\{\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\} \\
\mathsf{quicksort}(;\mathsf{a},\mathsf{c})\{\alpha_1\} \\
\{\exists \beta.\ \mathsf{array}\ \beta\,(\mathsf{a},\mathsf{c}) \\
\qquad \wedge\, \beta \sim \alpha_1 \wedge \mathbf{ord}\ \beta\}
\end{array}\right\} * \left\{\begin{array}{l}
\mathsf{array}\ \alpha_2(\mathsf{c}+1,\mathsf{b}) \\
\quad \wedge\, \alpha_1{\cdot}\alpha_2 \sim \alpha \\
\quad \wedge\, \{\alpha_1\} \leq^* \{\alpha_2\}
\end{array}\right\} \exists \alpha_1, \exists \alpha_2 \\
\{\exists \beta_1, \alpha_2.\ (\mathsf{array}\ \beta_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{b})) \\
\qquad \wedge\, \beta_1{\cdot}\alpha_2 \sim \alpha \wedge \{\beta_1\} \leq^* \{\alpha_2\} \wedge \mathbf{ord}\ \beta_1\} \\
\left.\begin{array}{l}
\{\mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{b})\} \\
\mathsf{quicksort}(;\mathsf{c}+1,\mathsf{b})\{\alpha_2\} \\
\{\exists \beta.\ \mathsf{array}\ \beta\,(\mathsf{c}+1,\mathsf{b}) \\
\qquad \wedge\, \beta \sim \alpha_2 \wedge \mathbf{ord}\ \beta\}
\end{array}\right\} * \left\{\begin{array}{l}
\mathsf{array}\ \beta_1(\mathsf{a},\mathsf{c}) \\
\quad \wedge\, \beta_1{\cdot}\alpha_2 \sim \alpha \\
\quad \wedge\, \{\beta_1\} \leq^* \{\alpha_2\} \\
\quad \wedge\, \mathbf{ord}\ \beta_1
\end{array}\right\} \exists \beta_1, \exists \alpha_2 \\
\{\exists \beta_1, \beta_2.\ (\mathsf{array}\ \beta_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \beta_2\,(\mathsf{c}+1,\mathsf{b})) \\
\qquad \wedge\, \beta_1{\cdot}\beta_2 \sim \alpha \wedge \{\beta_1\} \leq^* \{\beta_2\} \wedge \mathbf{ord}\ \beta_1 \wedge \mathbf{ord}\ \beta_2\}\big) \\
\mathbf{else\ skip} \\
\{\exists \beta.\ \mathsf{array}\ \beta\,(\mathsf{a},\mathsf{b}) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta\}.
\end{array}
$$

The pre- and postconditions of the above specification match those of the assumed specification 6.8. Moreover, the only free variables of the specified command are a and b, neither of which is modified. Thus we may satisfy 6.8

by using the command as the body of the procedure declaration:

$$
\begin{array}{l}
\mathsf{quicksort}(\mathsf{a},\mathsf{b}) = \\
\quad \textbf{if } \mathsf{a} < \mathsf{b} \textbf{ then newvar } \mathsf{c} \textbf{ in} \\
\quad \big(\textbf{newvar } \mathsf{x1},\mathsf{x2},\mathsf{r} \textbf{ in} \\
\quad\quad \big(\mathsf{x1} := [\mathsf{a}]\,;\, \mathsf{x2} := [\mathsf{b}]\,; \\
\quad\quad \textbf{if } \mathsf{x1} > \mathsf{x2} \textbf{ then } ([\mathsf{a}] := \mathsf{x2}\,;\, [\mathsf{b}] := \mathsf{x1}) \textbf{ else skip}\,; \\
\quad\quad \mathsf{r} := (\mathsf{x1} + \mathsf{x2}) \div 2\,;\, \mathsf{partition}(\mathsf{a}+1, \mathsf{b}-1, \mathsf{r}; \mathsf{c})\big)\,; \\
\quad\quad \mathsf{quicksort}(\mathsf{a},\mathsf{c})\,;\, \mathsf{quicksort}(\mathsf{c}+1,\mathsf{b})\big) \\
\quad \textbf{else skip}.
\end{array}
$$

6.5 Another Cyclic Buffer

When an array is used as a cyclic buffer, it represents a sequence in a more complex way than is described by the predicate array: The array location holding a sequence element is determined by modular arithmetic.

To illustrate, we assume that an n-element array has been allocated at location l, and we write $x \oplus y$ for the integer such that

$$x \oplus y = x + y \text{ modulo } \mathsf{n} \quad \text{and} \quad \mathsf{l} \le \mathsf{j} < \mathsf{l} + \mathsf{n}.$$

We will also use the following variables:

$$
\begin{array}{rl}
\mathsf{m}: & \text{number of active elements} \\
\mathsf{i}: & \text{pointer to first active element} \\
\mathsf{j}: & \text{pointer to first inactive element.}
\end{array}
$$

Let R abbreviate the assertion

$$R \stackrel{\text{def}}{=} 0 \le \mathsf{m} \le \mathsf{n} \wedge \mathsf{l} \le \mathsf{i} < \mathsf{l} + \mathsf{n} \wedge \mathsf{l} \le \mathsf{j} < \mathsf{l} + \mathsf{n} \wedge \mathsf{j} = \mathsf{i} \oplus \mathsf{m}.$$

It is easy to show (using ordinary Hoare logic) that

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

and

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

Then the following invariant describes the situation where the cyclic buffer contains the sequence α:

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

and the following is an annotated specification of a command that inserts the value x at the end of the sequence α. (The indications on the right refer to axiom schema in Section 6.1.)

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

$\{((\bigodot_{k=0}^{m-1} i \oplus k \mapsto \alpha_{k+1}) \ * \ (\bigodot_{k=0}^{0} j \oplus k \mapsto -) \ *$ (6.3)
$\quad (\bigodot_{k=1}^{n-m-1} j \oplus k \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$

$\{((\bigodot_{k=0}^{m-1} i \oplus k \mapsto \alpha_{k+1}) \ * \ j \oplus 0 \mapsto - \ *$ (6.2)
$\quad (\bigodot_{k=1}^{n-m-1} j \oplus k \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$

$[j] := x\,;$

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

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

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

$\{((\bigodot_{k=0}^{m-1} i \oplus k \mapsto (\alpha{\cdot}x)_{k+1}) \ * \ (\bigodot_{k=m}^{m} i \oplus k \mapsto (\alpha{\cdot}x)_{k+1}) \ *$ (6.2)
$\quad (\bigodot_{k=1}^{n-m-1} j \oplus k \mapsto -)) \wedge m = \#\alpha \wedge R \wedge m < n\}$

$\{((\bigodot_{k=0}^{m} i \oplus k \mapsto (\alpha{\cdot}x)_{k+1}) \ * \ (\bigodot_{k=1}^{n-m-1} j \oplus k \mapsto -))$ (6.3)
$\quad \wedge m + 1 = \#(\alpha{\cdot}x) \wedge R \wedge m < n\}$

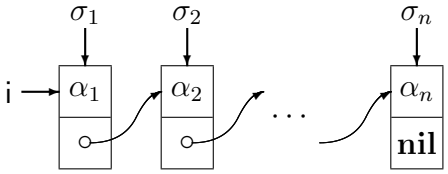
$\{((\bigodot_{k=0}^{m} i \oplus k \mapsto (\alpha{\cdot}x)_{k+1}) \ * \ (\bigodot_{k=0}^{n-m-2} j \oplus k \oplus 1 \mapsto -))$ (6.4)
$\quad \wedge m + 1 = \#(\alpha{\cdot}x) \wedge R \wedge m < n\}$

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

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

6.6 Connecting Two Views of Simple Lists

Somewhat surprisingly, the iterated separating conjunction can be used profitably to describe lists as well as arrays. A simple example is the connection between ordinary simple lists and Bornat lists:



From the definitions

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

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

and

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

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

one can show that list can be described in terms of listN and the separating conjunction by

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

The proof is by structural induction on α.

6.7 Specifying a Procedure for Subset Lists

A more spectacular example of the use of separated iterative conjunction with lists is provided by an early LISP program that, given a list representing a finite set, computes a list of lists representing all subsets of the input. (More generally, the program maps a list representing a finite multiset into a list of lists representing all sub-multisets; sets are the special case where the lists contain no duplicate elements.)

This algorithm was historically important since it created sublists that shared storage extensively, to an extent that reduced the use of storage to a lower order of complexity compared with unshared sublists.

Indeed, the resulting sharing structure is more complex than anything produced by the other algorithms in these notes. Thus, it is significant that the program can be verified in separation logic, and even more so that, with the use of the iterated separating conjunction, one can prove enough about the result to determine its size precisely.

We use the following variables to denote various kinds of sequences:

$$\begin{aligned} \alpha &: \text{ sequences of integers} \\ \beta, \gamma &: \text{ nonempty sequences of addresses} \\ \sigma &: \text{ nonempty sequences of sequences of integers.} \end{aligned}$$

Our goal is to write a procedure $\mathsf{subsets}$ satisfying

$$\begin{array}{l} H_{\mathsf{subsets}} \stackrel{\text{def}}{=} \\ \quad \{\mathsf{list}\,\alpha\,\mathsf{i}\} \\ \quad \mathsf{subsets}(\mathsf{j};\mathsf{i})\{\alpha\} \\ \quad \{\exists \sigma, \beta.\ \ \mathsf{ss}(\alpha,\sigma) \wedge \big(\mathsf{list}\,\alpha\,\mathsf{i} \ * \ \mathsf{list}\,\beta\,\mathsf{j} \ * \ (Q(\sigma,\beta) \wedge R(\beta))\big)\}. \end{array} \tag{6.9}$$

Here $\mathsf{ss}(\alpha,\sigma)$ asserts that σ is a sequence of the sequences that represent the subsets of (the set represented by) α. The inductive definition also specifies the order of elements in σ and its elements:

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

where $\mathsf{ext}_{\mathsf{a}}$ is a function that prefixes a to every element of its argument:

$$\begin{aligned} \#\mathsf{ext}_{\mathsf{a}}\sigma &\stackrel{\text{def}}{=} \#\sigma \\ \forall_{\mathsf{i}=1}^{\#\sigma}\,(\mathsf{ext}_{\mathsf{a}}\sigma)_i &\stackrel{\text{def}}{=} \mathsf{a}{\cdot}\sigma_i. \end{aligned}$$

(Here $\forall_{\mathsf{i}=1}^{\#\sigma}\,p$ abbreviates $\forall \mathsf{i}.\ (1 \le \mathsf{i} \le \#\sigma) \Rightarrow p$.)

The formula $Q(\sigma,\beta)$ asserts that the elements of β are lists representing the elements of σ:

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

The formula $R(\beta)$ uses the iterated separating conjunction to assert that the final element of β is the empty list, and that every previous element is a list

consisting of a single record followed by a list occurring later in β:

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

At this stage, we can determine the storage used by subsets. By induction on the definition of ss, using $\#\mathsf{ext}_{\mathsf{a}}\sigma = \#\sigma$, one can show that $\mathsf{ss}(\alpha, \sigma)$ implies $\#\sigma = 2^{\#\alpha}$. Then the definition of Q shows directly that $\#\beta = \#\sigma$.

Next, by induction on the definition of list, it is clear that list α describes a heap containing $\#\alpha$ two-cells, and similarly list β describes a heap containing $\#\beta$ two-cells. Also, from the definition of $R(\beta)$ and the iterated separating conjunction, $R(\beta)$ describes a heap containing $\#\beta - 1$ two-cells.

Now consider the postcondition of the specification (6.9) of subsets:

$$\{\exists \sigma, \beta.\ \mathsf{ss}(\alpha, \sigma) \wedge \Big(\mathsf{list}\,\alpha\,\mathsf{i} \ * \ \mathsf{list}\,\beta\,\mathsf{j} \ * \ (Q(\sigma, \beta) \wedge R(\beta))\Big)\}.$$

This assertion describes a heap containing three disjoint parts: a list containing $\#\alpha$ two-cells (the input list), a list containing $2^{\#\alpha}$ two-cells, and a list containing $2^{\#\alpha} - 1$ two-cells.

To verify the specification of subsets, we will also need to define the formula

$$W(\beta, \gamma, \mathsf{a}) \stackrel{\text{def}}{=} \#\gamma = \#\beta \wedge \bigodot_{\mathsf{i}=1}^{\#\gamma} \gamma_{\mathsf{i}} \mapsto \mathsf{a}, \beta_{\mathsf{i}},$$

which asserts that γ is a sequence of addresses such that γ_{i} is a list consisting of a followed by the ith element of β.

It is immediately evident that:

$$Q([\epsilon], [\mathbf{nil}]) \Leftrightarrow \mathbf{true}$$

$$R([\mathbf{nil}]) \Leftrightarrow \mathbf{emp}.$$

Less trivial are the following:

Proposition 19

$$W(\beta, \gamma, \mathsf{a}) \ * \ \mathsf{g} \mapsto \mathsf{a}, \mathsf{b} \Leftrightarrow W(\mathsf{b}{\cdot}\beta, \mathsf{g}{\cdot}\gamma, \mathsf{a}).$$

PROOF

$$
\begin{aligned}
& W(\beta,\gamma,\mathsf{a}) \;*\; \mathsf{g} \mapsto \mathsf{a},\mathsf{b} \\
& \quad \Leftrightarrow \#\gamma = \#\beta \wedge \left(\mathsf{g} \mapsto \mathsf{a},\mathsf{b} \;*\; \textstyle\bigodot_{\mathsf{i}=1}^{\#\gamma} \gamma_{\mathsf{i}} \mapsto \mathsf{a},\beta_{\mathsf{i}}\right) \\
& \quad \Leftrightarrow \#\mathsf{g}{\cdot}\gamma = \#\mathsf{b}{\cdot}\beta \wedge \\
& \qquad \left((\textstyle\bigodot_{\mathsf{i}=1}^{1}(\mathsf{g}{\cdot}\gamma)_{\mathsf{i}} \mapsto \mathsf{a},(\mathsf{b}{\cdot}\beta)_{\mathsf{i}}) \;*\; (\textstyle\bigodot_{\mathsf{i}=1}^{\#\mathsf{g}\cdot\gamma-1}(\mathsf{g}{\cdot}\gamma)_{\mathsf{i}+1} \mapsto \mathsf{a},(\mathsf{b}{\cdot}\beta)_{\mathsf{i}+1})\right) \\
& \quad \Leftrightarrow \#\mathsf{g}{\cdot}\gamma = \#\mathsf{b}{\cdot}\beta \wedge \textstyle\bigodot_{\mathsf{i}=1}^{\#\mathsf{g}\cdot\gamma}(\mathsf{g}{\cdot}\gamma)_{\mathsf{i}} \mapsto \mathsf{a},(\mathsf{b}{\cdot}\beta)_{\mathsf{i}} \\
& \quad \Leftrightarrow W(\mathsf{b}{\cdot}\beta,\mathsf{g}{\cdot}\gamma,\mathsf{a}).
\end{aligned}
$$

END OF PROOF

Proposition 20

$$Q(\sigma,\beta) \;*\; W(\beta,\gamma,\mathsf{a}) \Rightarrow Q((\mathsf{ext}_{\mathsf{a}}\sigma){\cdot}\sigma,\gamma{\cdot}\beta).$$

PROOF Let

$$
\begin{aligned}
p(\mathsf{i}) &\stackrel{\text{def}}{=} \mathsf{list}\;\sigma_{\mathsf{i}}\;\beta_{\mathsf{i}} \\
q(\mathsf{i}) &\stackrel{\text{def}}{=} \gamma_{\mathsf{i}} \mapsto \mathsf{a},\beta_{\mathsf{i}} \\
n &\stackrel{\text{def}}{=} \#\sigma.
\end{aligned}
$$

Then

$$
\begin{aligned}
&Q(\sigma,\beta) \ * \ W(\beta,\gamma,\mathsf{a}) \\
&\quad \Rightarrow \big(\#\beta = n \wedge \forall_{\mathsf{i}=1}^{\#\beta} p(\mathsf{i}) \ * \ \mathbf{true}\big) \ * \ \big(\#\gamma = \#\beta \wedge \bigodot_{\mathsf{i}=1}^{\#\gamma} q(\mathsf{i})\big) \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \Big(\big(\forall_{\mathsf{i}=1}^{n} p(\mathsf{i}) \ * \ \mathbf{true}\big) \ * \ \bigodot_{\mathsf{i}=1}^{n} q(\mathsf{i})\Big) \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \Big(\big(\forall \mathsf{i}.\ 1 \le \mathsf{i} \le n \Rightarrow p(\mathsf{i}) \ * \ \mathbf{true}\big) \ * \ \bigodot_{\mathsf{i}=1}^{n} q(\mathsf{i})\Big) \qquad \text{(a)} \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \Big(\forall \mathsf{i}.\ \big((1 \le \mathsf{i} \le n \Rightarrow p(\mathsf{i}) \ * \ \mathbf{true}) \ * \ \bigodot_{\mathsf{j}=1}^{n} q(\mathsf{j})\big)\Big) \qquad \text{(b)} \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \Big(\forall \mathsf{i}.\ \big(1 \le \mathsf{i} \le n \Rightarrow (p(\mathsf{i}) \ * \ \mathbf{true} \ * \ \bigodot_{\mathsf{j}=1}^{n} q(\mathsf{j}))\big)\Big) \qquad \text{(c)} \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true} \ * \ \bigodot_{\mathsf{j}=1}^{n} q(\mathsf{j})\big) \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true}\big) \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true} \ * \ \bigodot_{\mathsf{j}=1}^{n} q(\mathsf{j})\big) \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true}\big) \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true} \ * \ q(\mathsf{i})\big) \\
&\quad \Rightarrow \#\beta = n \wedge \#\gamma = n \wedge \forall_{\mathsf{i}=1}^{n}\big(p(\mathsf{i}) \ * \ \mathbf{true}\big) \wedge \\
&\qquad \forall_{\mathsf{i}=1}^{n}\big(\mathsf{list}\ \sigma_{\mathsf{i}}\ \beta_{\mathsf{i}} \ * \ \mathbf{true} \ * \ \gamma_{\mathsf{i}} \mapsto \mathsf{a}, \beta_{\mathsf{i}}\big) \\
&\quad \Rightarrow \#\gamma = n \wedge \#\beta = n \wedge \forall_{\mathsf{i}=1}^{n}\big(\mathsf{list}\ \sigma_{\mathsf{i}}\ \beta_{\mathsf{i}} \ * \ \mathbf{true}\big) \wedge \\
&\qquad \forall_{\mathsf{i}=1}^{n}\big(\mathsf{list}\ (\mathsf{ext}_{\mathsf{a}}\sigma)_{\mathsf{i}}\ \gamma_{\mathsf{i}} \ * \ \mathbf{true}\big) \\
&\quad \Rightarrow \#\gamma{\cdot}\beta = \#(\mathsf{ext}_{\mathsf{a}}\sigma){\cdot}\sigma \wedge \forall_{\mathsf{i}=1}^{\#\gamma\cdot\beta}\big(\mathsf{list}\ ((\mathsf{ext}_{\mathsf{a}}\sigma){\cdot}\sigma)_{\mathsf{i}}\ (\gamma{\cdot}\beta)_{\mathsf{i}} \ * \ \mathbf{true}\big) \\
&\quad \Rightarrow Q((\mathsf{ext}_{\mathsf{a}}\sigma){\cdot}\sigma, \gamma{\cdot}\beta).
\end{aligned}
$$

Here (a) implies (b) by the semidistributive law for $*$ and $\forall$, while (b) implies (c) since, as the reader may verify, $((p \Rightarrow q) \ * \ r) \Rightarrow (p \Rightarrow (q \ * \ r))$ is valid when p is pure. END OF PROOF

Proposition 21

$$R(\beta) \ * \ W(\beta,\gamma,\mathsf{a}) \Rightarrow R(\gamma{\cdot}\beta).$$

PROOF

$$
\begin{aligned}
& R(\beta) \ * \ W(\beta, \gamma, \mathsf{a}) \\
& \quad \Rightarrow (\beta_{\#\beta} = \mathbf{nil} \wedge \mathbf{emp}) \ * \\
& \qquad \textstyle\bigodot_{\mathsf{i}=1}^{\#\gamma} \gamma_{\mathsf{i}} \mapsto \mathsf{a}, \beta_{\mathsf{i}} \ * \\
& \qquad \textstyle\bigodot_{\mathsf{i}=1}^{\#\beta-1} (\exists \mathsf{a}, \mathsf{k}.\ \ \mathsf{i} < \mathsf{k} \leq \#\beta \wedge \beta_{\mathsf{i}} \mapsto \mathsf{a}, \beta_{\mathsf{k}}) \\
& \quad \Rightarrow ((\gamma{\cdot}\beta)_{\#\gamma\cdot\beta} = \mathbf{nil} \wedge \mathbf{emp}) \ * \\
& \qquad \textstyle\bigodot_{\mathsf{i}=1}^{\#\gamma} (\exists \mathsf{a}, \mathsf{k}.\ \ \mathsf{i} \leq \#\gamma < \mathsf{k} \leq \#\gamma{\cdot}\beta \wedge (\gamma{\cdot}\beta)_{\mathsf{i}} \mapsto \mathsf{a}, (\gamma{\cdot}\beta)_{\mathsf{k}}) \ * \\
& \qquad \textstyle\bigodot_{\mathsf{i}=\#\gamma+1}^{\#\gamma\cdot\beta-1} (\exists \mathsf{a}, \mathsf{k}.\ \ \mathsf{i} < \mathsf{k} \leq \#\gamma{\cdot}\beta \wedge (\gamma{\cdot}\beta)_{\mathsf{i}} \mapsto \mathsf{a}, (\gamma{\cdot}\beta)_{\mathsf{k}}) \\
& \quad \Rightarrow ((\gamma{\cdot}\beta)_{\#\gamma\cdot\beta} = \mathbf{nil} \wedge \mathbf{emp}) \ * \\
& \qquad \textstyle\bigodot_{\mathsf{i}=1}^{\#\gamma\cdot\beta-1} (\exists \mathsf{a}, \mathsf{k}.\ \ \mathsf{i} < \mathsf{k} \leq \#\gamma{\cdot}\beta \wedge (\gamma{\cdot}\beta)_{\mathsf{i}} \mapsto \mathsf{a}, (\gamma{\cdot}\beta)_{\mathsf{k}}) \\
& \quad \Rightarrow R(\gamma{\cdot}\beta).
\end{aligned}
$$

END OF PROOF

From the last two propositions, we have

$$
\begin{aligned}
& (Q(\sigma, \beta) \wedge R(\beta)) \ * \ W(\beta, \gamma, \mathsf{a}) \\
& \quad \Rightarrow (Q(\sigma, \beta) \ * \ W(\beta, \gamma, \mathsf{a})) \wedge (R(\beta) \ * \ W(\beta, \gamma, \mathsf{a})) \\
& \quad \Rightarrow Q((\mathsf{ext}_{\mathsf{a}}\sigma){\cdot}\sigma, \gamma{\cdot}\beta) \wedge R(\gamma{\cdot}\beta).
\end{aligned}
$$

Using these results, we can construct and verify a recursive procedure satisfying (6.9). The first step is to construct a subsidiary recursive procedure extapp that, given a list representing a sequence β of lists, creates a list representing the sequence of those lists that are obtained by prefixing the integer a to each element of β:

$$
\begin{aligned}
H_{\mathsf{extapp}} \stackrel{\text{def}}{=} & \{\mathsf{list}\ \beta\, \mathsf{i}\} \\
& \mathsf{extapp}(\mathsf{k}; \mathsf{a}, \mathsf{i}, \mathsf{j})\{\beta\} \\
& \{\exists \gamma.\ \ \mathsf{list}\ \beta\, \mathsf{i} \ * \ \mathsf{lseg}\ \gamma\, (\mathsf{k}, \mathsf{j}) \ * \ W(\beta, \gamma, \mathsf{a})\}.
\end{aligned}
$$

The following annotated specification describes the body of extapp:

$$
\begin{array}{l}
\{\mathsf{list}\ \beta\,\mathsf{i}\}\ \mathsf{extapp}(\mathsf{k};\mathsf{a},\mathsf{i},\mathsf{j})\{\beta\}\ \{\exists\gamma.\ \ \mathsf{list}\ \beta\,\mathsf{i}\ *\ \mathsf{lseg}\ \gamma\,(\mathsf{k},\mathsf{j})\ *\ W(\beta,\gamma,\mathsf{a})\}\vdash \\
\quad \{\mathsf{list}\ \beta\,\mathsf{i}\} \\
\quad \textbf{if}\ \mathsf{i} = \textbf{nil}\ \textbf{then}\ \mathsf{k} := \mathsf{j}\ \textbf{else} \\
\qquad \{\exists \mathsf{b},\mathsf{i}',\beta'.\ \ \beta = \mathsf{b}{\cdot}\beta' \wedge (\mathsf{i} \mapsto \mathsf{b},\mathsf{i}'\ *\ \mathsf{list}\ \beta'\,\mathsf{i}')\} \\
\qquad \textbf{newvar}\ \mathsf{b},\mathsf{i}',\mathsf{g}\ \textbf{in} \\
\qquad\quad \Big(\mathsf{b} := [\mathsf{i}]\ ;\ \mathsf{i}' := [\mathsf{i}+1]\ ; \\
\qquad\quad \{\exists\beta'.\ \ \beta = \mathsf{b}{\cdot}\beta' \wedge (\mathsf{i} \mapsto \mathsf{b},\mathsf{i}'\ *\ \mathsf{list}\ \beta'\,\mathsf{i}')\} \\
\qquad\quad \left.\begin{array}{l}\{\mathsf{list}\ \beta'\,\mathsf{i}'\} \\ \mathsf{extapp}(\mathsf{k};\mathsf{a},\mathsf{i}',\mathsf{j})\{\beta'\} \\ \{\exists\gamma.\ \ \mathsf{list}\ \beta'\,\mathsf{i}'\ *\ \mathsf{lseg}\ \gamma\,(\mathsf{k},\mathsf{j})\ *\ W(\beta',\gamma,\mathsf{a})\} \\ \{\exists\gamma'.\ \ \mathsf{list}\ \beta'\,\mathsf{i}'\ *\ \mathsf{lseg}\ \gamma'\,(\mathsf{k},\mathsf{j})\ *\ W(\beta',\gamma',\mathsf{a})\}\end{array}\right\} * \left\{\begin{array}{c}\beta = \mathsf{b}{\cdot}\beta' \\ \wedge \\ \mathsf{i} \mapsto \mathsf{b},\mathsf{i}'\end{array}\right\} \exists\beta' \\
\qquad\quad\ \ \{\exists\beta',\gamma'.\ \ \beta = \mathsf{b}{\cdot}\beta' \wedge \\
\qquad\quad\ \ \ \ (\mathsf{list}\ (\mathsf{b}{\cdot}\beta')\,\mathsf{i}\ *\ \mathsf{lseg}\ \gamma'\,(\mathsf{k},\mathsf{j})\ *\ W(\beta',\gamma',\mathsf{a}))\} \\
\qquad\quad\ \ \mathsf{g} := \mathbf{cons}(\mathsf{a},\mathsf{b}); \\
\qquad\quad\ \ \{\exists\beta',\gamma'.\ \ \beta = \mathsf{b}{\cdot}\beta' \wedge \\
\qquad\quad\ \ \ \ (\mathsf{list}\ (\mathsf{b}{\cdot}\beta')\,\mathsf{i}\ *\ \mathsf{lseg}\ \gamma'\,(\mathsf{k},\mathsf{j})\ *\ W(\mathsf{b}{\cdot}\beta',\mathsf{g}{\cdot}\gamma',\mathsf{a}))\} \\
\qquad\quad\ \ \mathsf{k} := \mathbf{cons}(\mathsf{g},\mathsf{k}) \\
\qquad\quad\ \ \{\exists\beta',\gamma'.\ \ \beta = \mathsf{b}{\cdot}\beta' \wedge \\
\qquad\quad\ \ \ \ (\mathsf{list}\ (\mathsf{b}{\cdot}\beta')\,\mathsf{i}\ *\ \mathsf{lseg}\ \mathsf{g}{\cdot}\gamma'\,(\mathsf{k},\mathsf{j})\ *\ W(\mathsf{b}{\cdot}\beta',\mathsf{g}{\cdot}\gamma',\mathsf{a}))\}\Big) \\
\quad \{\exists\gamma.\ \ \mathsf{list}\ \beta\,\mathsf{i}\ *\ \mathsf{lseg}\ \gamma\,(\mathsf{k},\mathsf{j})\ *\ W(\beta,\gamma,\mathsf{a})\}.
\end{array}
$$

Finally, we may define and verify the main procedure subsets satisfying (6.9):

$$
\begin{array}{l}
H_{\mathsf{subsets}}, H_{\mathsf{extapp}} \vdash \\
\quad \{\mathsf{list}\ \alpha\ \mathsf{i}\} \\
\quad \textbf{if}\ \mathsf{i} = \textbf{nil}\ \textbf{then}\ \mathsf{j} := \textbf{cons}(\textbf{nil}, \textbf{nil})\ \textbf{else} \\
\qquad \{\exists \mathsf{a}, \mathsf{i}', \alpha'.\ \ \alpha = \mathsf{a}{\cdot}\alpha' \wedge (\mathsf{i} \mapsto \mathsf{a}, \mathsf{i}'\ *\ \mathsf{list}\ \alpha'\ \mathsf{i}')\} \\
\qquad \textbf{newvar}\ \mathsf{a}, \mathsf{i}', \mathsf{j}'\ \textbf{in}\ \big(\mathsf{a} := [\mathsf{i}]\ ;\ \mathsf{i}' := [\mathsf{i}+1]\ ; \\
\qquad\quad \{\exists \alpha'.\ \ \alpha = \mathsf{a}{\cdot}\alpha' \wedge (\mathsf{i} \mapsto \mathsf{a}, \mathsf{i}'\ *\ \mathsf{list}\ \alpha'\ \mathsf{i}')\} \\
\qquad\quad \left.\begin{array}{l} \{\mathsf{list}\ \alpha'\ \mathsf{i}'\} \\ \mathsf{subsets}(\mathsf{j}'; \mathsf{i}')\{\alpha'\} \\ \{\exists \sigma, \beta.\ \ \mathsf{ss}(\alpha', \sigma) \wedge \\ \quad (\mathsf{list}\ \alpha'\ \mathsf{i}'\ *\ \mathsf{list}\ \beta\ \mathsf{j}'\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\} \\ \{\exists \sigma', \beta'.\ \ \mathsf{ss}(\alpha', \sigma') \wedge \\ \quad (\mathsf{list}\ \alpha'\ \mathsf{i}'\ *\ \mathsf{list}\ \beta'\ \mathsf{j}'\ *\ (Q(\sigma', \beta') \wedge R(\beta')))\} \end{array}\right\} * \left\{\begin{array}{c} \alpha = \mathsf{a}{\cdot}\alpha' \\ \wedge \\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{i}' \end{array}\right\} \exists \alpha' \\
\qquad\quad \{\exists \alpha', \sigma', \beta'.\ \ \big(\alpha = \mathsf{a}{\cdot}\alpha' \wedge \mathsf{ss}(\alpha', \sigma') \wedge (\mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i}\ *\ (Q(\sigma', \beta') \wedge R(\beta')))\big)\ *\ \mathsf{list}\ \beta'\ \mathsf{j}'\} \\
\qquad\quad \left.\begin{array}{l} \{\mathsf{list}\ \beta'\ \mathsf{j}'\} \\ \mathsf{extapp}(\mathsf{j}; \mathsf{a}, \mathsf{j}', \mathsf{j}')\{\beta'\} \\ \{\exists \gamma.\ \ \mathsf{list}\ \beta'\ \mathsf{j}'\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{j}, \mathsf{j}')\ *\ W(\beta', \gamma, \mathsf{a})\} \end{array}\right\} * \left\{\begin{array}{c} \alpha = \mathsf{a}{\cdot}\alpha' \wedge \mathsf{ss}(\alpha', \sigma') \wedge \\ (\mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i}\ * \\ (Q(\sigma', \beta') \wedge R(\beta'))) \end{array}\right\} \exists \alpha', \sigma', \beta' \\
\qquad\quad \{\exists \alpha', \sigma', \beta'.\ \ \big(\alpha = \mathsf{a}{\cdot}\alpha' \wedge \mathsf{ss}(\alpha', \sigma') \wedge (\mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i}\ *\ (Q(\sigma', \beta') \wedge R(\beta')))\big)\ * \\
\qquad\quad\quad (\exists \gamma.\ \ \mathsf{list}\ \beta'\ \mathsf{j}'\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{j}, \mathsf{j}')\ *\ W(\beta', \gamma, \mathsf{a}))\} \\
\qquad\quad \{\exists \alpha', \sigma', \beta', \gamma.\ \ \alpha = \mathsf{a}{\cdot}\alpha' \wedge \mathsf{ss}(\mathsf{a}{\cdot}\alpha', (\mathsf{ext}_{\mathsf{a}}\sigma'){\cdot}\sigma') \wedge \\
\qquad\quad\quad \big(\mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i}\ *\ \mathsf{list}\ (\gamma{\cdot}\beta')\ \mathsf{j}\ *\ (Q((\mathsf{ext}_{\mathsf{a}}\sigma'){\cdot}\sigma', \gamma{\cdot}\beta') \wedge R(\gamma{\cdot}\beta'))\big)\}\big) \\
\{\exists \sigma, \beta.\ \ \mathsf{ss}(\alpha, \sigma) \wedge \big(\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta))\big)\}.
\end{array}
$$

Exercise 1

Derive the axiom scheme

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

from the other axiom schemata for iterating separating conjunction given in Section 6.1.

Exercise 2

The following is an alternative global rule for allocation that uses a ghost variable (v'):

- The ghost-variable global form (ALLOCGG)

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

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

Derive (ALLOCGG) from (ALLOCG) and (ALLOCL) from (ALLOCGG).

Exercise 3

Write an iterative version (in which recursion or, for that matter, procedures are not used) of the program for subset lists in Section 6.7. Since it is natural for efficent iterative programs to reverse lists, your program will not give exactly the same results as the one in Section 6.7.

Specifically, you will need to replace the predicates ss and W by

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

and

$$W'(\beta, \gamma, \mathsf{a}) \stackrel{\text{def}}{=} \#\gamma = \#\beta \wedge \bigodot_{\mathsf{i}=1}^{\#\gamma} \gamma_{\mathsf{i}} \mapsto \mathsf{a}, (\beta^{\dagger})_{\mathsf{i}}.$$

Then your program should contain a nest of two **while** commands. It should satisfy

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

The invariant of the outer **while** should be

$$\exists\alpha',\alpha'',\sigma,\beta.\ \ \alpha'^\dagger\cdot\alpha''=\alpha\wedge\mathsf{ss}'(\alpha',\sigma)\wedge$$
$$(\mathsf{list}\,\alpha''\,\mathsf{i}\ *\ \mathsf{list}\,\beta\,\mathsf{j}\ *\ (Q(\sigma,\beta)\wedge R(\beta))),$$

and the invariant of the inner **while** should be

$$\exists\alpha',\alpha'',\sigma,\beta',\beta'',\gamma.\ \ \alpha'^\dagger\cdot\mathsf{a}\cdot\alpha''=\alpha\wedge\mathsf{ss}'(\alpha',\sigma)\wedge$$
$$(\mathsf{list}\,\alpha''\,\mathsf{i}\ *\ \mathsf{lseg}\,\gamma\,(\mathsf{l},\mathsf{j})\ *\ \mathsf{lseg}\,\beta'\,(\mathsf{j},\mathsf{m})\ *\ \mathsf{list}\,\beta''\,\mathsf{m}\ *$$
$$(Q(\sigma,\beta'\cdot\beta'')\wedge R(\beta'\cdot\beta''))\ *\ W'(\beta',\gamma,\mathsf{a})).$$

At the completion of the inner **while**, the assertion

$$\exists\alpha',\alpha'',\sigma,\beta',\gamma.\ \ \alpha'^\dagger\cdot\mathsf{a}\cdot\alpha''=\alpha\wedge\mathsf{ss}'(\alpha',\sigma)\wedge$$
$$(\mathsf{list}\,\alpha''\,\mathsf{i}\ *\ \mathsf{lseg}\,\gamma\,(\mathsf{l},\mathsf{j})\ *\ \mathsf{list}\,\beta'\,\mathsf{j}\ *$$
$$(Q(\sigma,\beta')\wedge R(\beta'))\ *\ W'(\beta',\gamma,\mathsf{a}))$$

should hold.