An Introduction
to Separation Logic
©2009 John C. Reynolds
February 11, 2009

Chapter 4

Lists and List Segments

In this chapter, we begin to explore data structures that represent abstract types of data. Our starting point will be various kinds of lists, which represent sequences.

Sequences and their primitive operations are a sufficiently standard — and straightforward — mathematical concept that we omit their definition. We will use the following notations, where α and β are sequences:

- ϵ for the empty sequence.
- $[a]$ for the single-element sequence containing a. (We will omit the brackets when a is not a sequence.)
- $\alpha{\cdot}\beta$ for the composition of α followed by β.
- $\alpha^\dagger$ for the reflection of α.
- $\#\alpha$ for the length of α.
- α_i for the ith component of α (where $1 \leq i \leq \#\alpha$).

These operations obey a variety of laws, including:

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

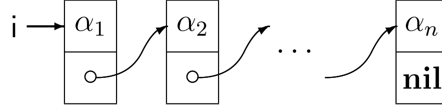
$$\epsilon^\dagger = \epsilon \qquad [a]^\dagger = [a] \qquad (\alpha{\cdot}\beta)^\dagger = \beta^\dagger{\cdot}\alpha^\dagger$$

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

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

4.1 Singly-Linked List Segments

In Section 1.6, we defined the predicate $\mathsf{list}\ \alpha\ \mathsf{i}$, indicating that i is a list representing the sequence α,



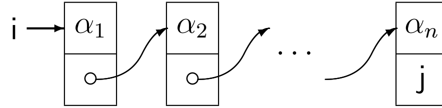
by structural induction on α:

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

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

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

We write $\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})$ to indicate that i to j is a *list segment* representing the sequence α:



As with list, this predicate is defined by structural induction on α:

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

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

It is easily shown to satisfy the following properties (where a and b denote values that are components of sequences):

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

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

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

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

It is illuminating to prove (by structural induction on α) the second of the above properties, which is a composition law for list segments. In the

base case, where α is empty, we use the definition of $\mathsf{lseg}\ \epsilon\,(\mathsf{i},\mathsf{j})$, the purity of $\mathsf{i} = \mathsf{j}$, the fact that **emp** is a neutral element, and the fact that ϵ is an identity for the composition of sequences, to obtain

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

For the induction step, when α has the form $\mathsf{a}{\cdot}\alpha'$, we use the definition of $\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha'\,(\mathsf{i},\mathsf{j})$, the induction hypothesis, the definition of $\mathsf{lseg}\ \mathsf{a}{\cdot}(\alpha'{\cdot}\beta)\,(\mathsf{i},\mathsf{j})$, and the associativity of the composition of sequences:

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

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

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

Moreover, as we will see in Section 4.3, one can show that $\mathsf{list}\ \alpha\ \mathsf{i}$ and $\exists \alpha.\ \mathsf{list}\ \alpha\ \mathsf{i}$ are precise predicates. For list segments, however, the situation is more complex.

One can derive the following valid assertions, which, when $\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})$ holds, give conditions for determining whether a list segment is empty (i.e., denotes the empty sequence):

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

But these formulas do not say whether α is empty when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$. In this case, for example, if the heap satisfies $\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}$ then $\mathsf{lseg}\ \mathsf{a}\,(\mathsf{i}, \mathsf{j})$ holds, but also $\mathsf{lseg}\ \epsilon\,(\mathsf{i}, \mathsf{j})$ holds for a subheap (namely the empty heap). Thus $\exists\alpha.\ \mathsf{lseg}\ \alpha\,(\mathsf{i}, \mathsf{j})$ is not precise. Indeed, when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$, there may be no way to compute whether the list segment is empty.

In general, when

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

we have

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

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

We can define *nontouching list segments* inductively by:

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

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

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

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

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

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

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

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

Nevertheless, there are cases where a list segment may be touching — an example is the cyclic buffer described in the next section — and one must face the fact that extra information is needed to determine whether the segment is empty.

It should be noted that $\mathsf{list}\,\alpha\,\mathsf{i}$, $\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})$, and $\mathsf{ntlseq}\,\alpha\,(\mathsf{i},\mathsf{j})$ are all precise assertions. On the other hand, although (as we will show in Section 4.3) $\exists\alpha.\ \mathsf{list}\,\alpha\,\mathsf{i}$ and $\exists\alpha.\ \mathsf{ntlseq}\,\alpha\,(\mathsf{i},\mathsf{j})$ are precise, $\exists\alpha.\ \mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})$ is not precise.

As simple illustrations of reasoning about list-processing, which illustrate use of the inference rules for heap-manipulating commands, we give detailed annotated specifications of programs for inserting and deleting list elements. To insert an element a at the beginning of a list segment:

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

or, more concisely:

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

To insert a at the end of a list segment, assuming j points to the last record in the segment:

$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\}$$
$$\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})\ ; \qquad \text{(CONSNOG)}$$
$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto -\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$[\mathsf{j}+1] := \mathsf{l} \qquad \text{(MUG)}$$
$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$\{\mathsf{lseg}\,\alpha\cdot\mathsf{a}\,(\mathsf{i},\mathsf{l})\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$$
$$\{\mathsf{lseg}\,\alpha\cdot\mathsf{a}\cdot\mathsf{b}\,(\mathsf{i},\mathsf{k})\}.$$

(Here we have included more annotations then we usually will, to make the use of the global mutation rule (MUG) explicit.)

Next is a program for deleting an element at the beginning of a nonempty list segment:

$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}$
$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$
$\{\exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$
$\mathsf{j} := [\mathsf{i}+1]\,;$ (LKNOG)
$\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ (\mathsf{i} \mapsto \mathsf{a}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$
$\{\mathsf{i} \mapsto \mathsf{a}\ *\ (\mathsf{i}+1 \mapsto \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$
dispose $\mathsf{i}\,;$ (DISG)
$\{\mathsf{i}+1 \mapsto \mathsf{j}\ *\ \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$
dispose $\mathsf{i}+1\,;$ (DISG)
$\{\mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$
$\mathsf{i} := \mathsf{j}$ (AS)
$\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{k})\}.$

Notice that the effect of the lookup command is to erase the existential quantifier of j.

Finally, to delete an element at the end of segment in constant time, we must have pointers j and k to the last two records:

$\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}$
$[\mathsf{j}+1] := \mathsf{l}\,;$ (MUG)
$\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}$
dispose $\mathsf{k}\,;$ (DISG)
dispose $\mathsf{k}+1$ (DISG)
$\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\}$
$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\}.$

4.2 A Cyclic Buffer

As a more elaborate example, we consider a cyclic buffer, consisting of an active list segment satisfying $\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})$ (where the sequence α is the contents of the buffer) and an inactive segment satisfying $\mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})$ (where the sequence β is arbitrary). We will use an unchanging variable n to record the combined length of the two buffers.

When $\mathsf{i} = \mathsf{j}$, the buffer is either empty ($\#\alpha = 0$) or full ($\#\beta = 0$). To distinguish these cases, one must keep track of additional information; we will do this by recording the length of the active segment in a variable m. Thus we have the following invariant, which must be preserved by programs for inserting or deleting elements:

$$\exists\beta.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} = \#\alpha + \#\beta$$

The following program will insert the element x into the buffer:

$$\begin{array}{ll}
\{\exists\beta.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} = \#\alpha + \#\beta \wedge \mathsf{n} - \mathsf{m} > 0\} & \\
\{\exists\mathsf{b},\beta.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{lseg}\,\mathsf{b}{\cdot}\beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} = \#\alpha + \#\mathsf{b}{\cdot}\beta\} & \\
\{\exists\beta,\mathsf{j}''.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto -,\mathsf{j}'' \ast \mathsf{lseg}\,\beta\,(\mathsf{j}'',\mathsf{i})) \wedge & \\
\quad \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta\} & \\
[\mathsf{j}] := \mathsf{x}\,; & \text{(MUG)} \\
\{\exists\beta,\mathsf{j}''.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{x},\mathsf{j}'' \ast \mathsf{lseg}\,\beta\,(\mathsf{j}'',\mathsf{i})) \wedge & \\
\quad \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta\} & \\
\{\exists\beta,\mathsf{j}''.\ \mathsf{j} + 1 \mapsto \mathsf{j}'' \ast ((\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{x} \ast \mathsf{lseg}\,\beta\,(\mathsf{j}'',\mathsf{i})) \wedge & \\
\quad \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta)\} & \\
\mathsf{j} := [\mathsf{j} + 1]\,; & \text{(LKG)} \\
\{\exists\beta,\mathsf{j}'.\ \mathsf{j}' + 1 \mapsto \mathsf{j} \ast ((\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}') \ast \mathsf{j}' \mapsto \mathsf{x} \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge & \\
\quad \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta)\} & \\
\{\exists\beta,\mathsf{j}'.\ (\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}') \ast \mathsf{j}' \mapsto \mathsf{x},\mathsf{j} \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge & \\
\quad \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta\} & \\
\{\exists\beta.\ (\mathsf{lseg}\,\alpha{\cdot}\mathsf{x}\,(\mathsf{i},\mathsf{j}) \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} + 1 = \#\alpha{\cdot}\mathsf{x} \wedge \mathsf{n} = \#\alpha{\cdot}\mathsf{x} + \#\beta\} & \\
\mathsf{m} := \mathsf{m} + 1 & \text{(AS)} \\
\{\exists\beta.\ (\mathsf{lseg}\,\alpha{\cdot}\mathsf{x}\,(\mathsf{i},\mathsf{j}) \ast \mathsf{lseg}\,\beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\alpha{\cdot}\mathsf{x} \wedge \mathsf{n} = \#\alpha{\cdot}\mathsf{x} + \#\beta\} &
\end{array}$$

Note the use of (LKG) for $j := [j+1]$, with v, v', and v'' replaced by j, j', and j''; e replaced by $\mathsf{j}+1$; and r replaced by

$$((\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}') \ * \ \mathsf{j}' \mapsto \mathsf{x} \ * \ \mathsf{lseg}\ \beta\ (\mathsf{j}'', \mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} - 1 = \#\alpha + \#\beta).$$

4.3 Preciseness Revisited

Before turning to other kinds of lists and list segments, we establish the preciseness of various assertions about simple lists. The basic idea is straighforward, but the details become somewhat complicated since we wish to make a careful distinction between language and metalanguage. In particular, we will have both metavariables and object variables that range over sequences (as well as both metavariables and object variables that range over integers). We will also extend the concept of the store so that it maps object sequence variables into sequences (as well as integer variables into integers). (Note, however, that we will use α with various decorations for both object and metavariables ranging over sequences.)

Proposition 15 (1) $\exists\alpha.\ \mathsf{list}\ \alpha\ \mathsf{i}$ *is a precise assertion.* (2) $\mathsf{list}\ \alpha\ \mathsf{i}$ *is a precise assertion.*

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

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

$$[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \models \mathsf{list}\ \alpha\ \mathsf{i} \wedge \alpha = \epsilon$$

$$[\,\mathsf{i}\colon i \mid \alpha\colon \epsilon\,], h \models \mathsf{list}\ \epsilon\ \mathsf{i}$$

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

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

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

$$[\,\mathsf{i}\colon i \mid \alpha\colon a{\cdot}\alpha' \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha'\,], h \models \mathsf{list}\ \alpha\ \mathsf{i} \wedge \alpha = \mathsf{a}{\cdot}\alpha'$$

$$[\,\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha'\,], h \models \mathsf{list}\ (\mathsf{a}{\cdot}\alpha')\ \mathsf{i}$$

$$[\,\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \alpha'\colon \alpha'\,], h \models \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ * \ \mathsf{list}\ \alpha'\ \mathsf{j}$$

$$\exists j.\ [\,\mathsf{i}\colon i \mid \mathsf{a}\colon a \mid \mathsf{j}\colon j \mid \alpha'\colon \alpha'\,], h \models \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ * \ \mathsf{list}\ \alpha'\ \mathsf{j},$$

so that there are j and h' such that $i \neq \mathbf{nil}$ (since $\mathbf{nil}$ is not a location), $h = [\,i\colon a \mid i{+}1\colon j\,]{\cdot}h'$, and $[\,\mathsf{j}\colon j \mid \alpha'\colon \alpha'\,], h' \models \mathsf{list}\ \alpha'\ \mathsf{j}$ — and by the substitution theorem (Proposition 3 in Section 2.1) $[\,\mathsf{i}\colon j \mid \alpha\colon \alpha'\,], h' \models \mathsf{list}\ \alpha\ \mathsf{i}$.

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

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

We must show that $h_0 = h_1$.

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

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

We will complete our proof by showing, by structural induction on α_0, that, for all α_0, α_1, i, h, h_0, and h_1, if h_0, $h_1 \subseteq h$ and the statements displayed above hold, then $h_0 = h_1$.

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

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

For the induction step suppose $\alpha_0 = a_0{\cdot}\alpha_0'$. Then by (b), there are j_0 and h_0' such that $i \neq \mathbf{nil}$, $h_0 = [\,i\colon a_0 \mid i+1\colon j_0\,]\cdot h_0'$. and $[\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha_0'\,], h_0' \models \mathsf{list}\ \alpha\ \mathsf{i}$.

Moreover, if α_1 were empty, then by (a) we would have the contradiction $i = \mathbf{nil}$. Thus α_1 must be $a_1{\cdot}\alpha_1'$ for some a_1 and α_1'. Then by (b), there are j_1 and h_1' such that $i \neq \mathbf{nil}$, $h_1 = [\,i\colon a_1 \mid i+1\colon j_1\,]\cdot h_1'$. and $[\,\mathsf{i}\colon j_1 \mid \alpha\colon \alpha_1'\,], h_1' \models \mathsf{list}\ \alpha\ \mathsf{i}$.

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

$$[\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha_0'\,], h_0' \models \mathsf{list}\ \alpha\ \mathsf{i} \text{ and } [\,\mathsf{i}\colon j_0 \mid \alpha\colon \alpha_1'\,], h_1' \models \mathsf{list}\ \alpha\ \mathsf{i},$$

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

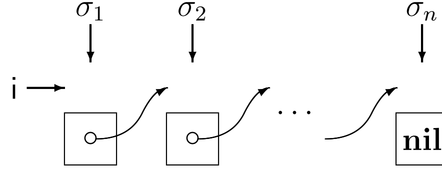
(2) As described in Section 2.3.3, p is precise whenever $p \Rightarrow q$ is valid and q is precise. Thus, since $\mathbf{list}\ \alpha\ \mathsf{i} \Rightarrow \exists\alpha.\ \mathbf{list}\ \alpha\ \mathsf{i}$ is valid, $\mathbf{list}\ \alpha\ \mathsf{i}$ is precise.

END OF PROOF

4.4 Bornat Lists

An alternative approach to lists has been advocated by Richard Bornat. His view is that a list represents, not a sequence of values, but a sequence of

addresses where values may be placed. To capture this concept we write $\mathsf{listN}\ \sigma\ \mathsf{i}$ to indicate that i is a Bornat list representing the sequence σ of addresses:



The definition by structural induction on σ is:

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

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

Notice that the heap described by $\mathsf{listN}\ \sigma\ \mathsf{i}$ consists only of the link fields of the list, not the data fields.

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

The following is an annotated specification of a program for reversing a Bornat list — which is the same program as in Section 1.6, but with a very different specification:

$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i}\}$$
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i}\ *\ (\mathbf{emp} \land \mathbf{nil} = \mathbf{nil})\}$$
$$\mathsf{j} := \mathbf{nil}\ ; \qquad \text{(AS)}$$
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i}\ *\ (\mathbf{emp} \land \mathsf{j} = \mathbf{nil})\}$$
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i}\ *\ \mathsf{listN}\ \epsilon\ \mathsf{j}\}$$
$$\{\exists \sigma, \tau.\ (\mathsf{listN}\ \sigma\ \mathsf{i}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau\}$$
$$\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$\big(\{\exists \sigma, \tau.\ (\mathsf{listN}\ (\mathsf{i}{\cdot}\sigma)\ \mathsf{i}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = (\mathsf{i}{\cdot}\sigma)^\dagger{\cdot}\tau\}$$
$$\{\exists \sigma, \tau, \mathsf{k}.\ (\mathsf{i} + 1 \mapsto \mathsf{k}\ *\ \mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = (\mathsf{i}{\cdot}\sigma)^\dagger{\cdot}\tau\}$$
$$\mathsf{k} := [\mathsf{i} + 1]\ ; \qquad \text{(LKNOG)}$$
$$\{\exists \sigma, \tau.\ (\mathsf{i} + 1 \mapsto \mathsf{k}\ *\ \mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = (\mathsf{i}{\cdot}\sigma)^\dagger{\cdot}\tau\}$$
$$[\mathsf{i} + 1] := \mathsf{j}\ ; \qquad \text{(MUG)}$$
$$\{\exists \sigma, \tau.\ (\mathsf{i} + 1 \mapsto \mathsf{j}\ *\ \mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = (\mathsf{i}{\cdot}\sigma)^\dagger{\cdot}\tau\}$$
$$\{\exists \sigma, \tau.\ (\mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ (\mathsf{i}{\cdot}\tau)\ \mathsf{i}) \land \sigma_0^\dagger = \sigma^\dagger{\cdot}\mathsf{i}{\cdot}\tau\}$$

$$\{\exists \sigma, \tau.\ (\mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ \tau\ \mathsf{i}) \wedge \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau\}$$

$$\mathsf{j} := \mathsf{i}\,; \qquad \text{(AS)}$$

$$\mathsf{i} := \mathsf{k} \qquad \text{(ASan)}$$

$$\{\exists \sigma, \tau.\ (\mathsf{listN}\ \sigma\ \mathsf{i}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \wedge \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau\}\Big)$$

$$\{\exists \sigma, \tau.\ \mathsf{listN}\ \tau\ \mathsf{j} \wedge \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau \wedge \sigma = \epsilon\}$$

$$\{\mathsf{listN}\ \sigma_0^\dagger\ \mathsf{j}\}$$

In fact, this is a stronger specification then that given earlier, since it shows that the program does not alter the addresses where the list data is stored.

4.5 Simple Procedures

To program more complex examples of list processing, we will need procedures — especially recursive procedures. So we will digress from our exposition to add a simple procedure mechanism to our programming language, and to formulate additional inference rules for the verification of such procedures.

By "simple", we mean that the following restrictions are imposed:

- Parameters are variables and expressions, not commands or procedure names.
- There are no "global" variables: All free variables of the procedure body must be formal parameters of the procedure.
- Procedures are proper, i.e., their calls are commands.
- Calls are restricted to prevent aliasing.

(More general kinds of procedures will be considered later.) An additional peculiarity, which substantially simplifies reasoning about simple procedures, is that we syntactically distinguish parameters that may be modified from those that may not be.

A *simple nonrecursive procedure definition* is a command of the form

$$\textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c',$$

while a *simple recursive procedure definition* is a command of the form

$$\textbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c',$$

where

- h is a binding occurrence of a procedure name, whose scope is c' in the nonrecursive case, and c and c' in the recursive case. (Procedure names are nonassignable variables that will have a different behavior than the variables introduced earlier.)
- c and c' are commands.
- $v_1, \ldots, v_m; v'_1, \ldots, v'_n$ is a list of distinct variables, called *formal parameters*, that includes all of the free variables of c. The formal parameters are binding occurrences whose scope is c.
- $v_1, \ldots, v_m$ includes all of the variables modified by c.

Then a *procedure call* is a command of the form

$$h(w_1, \ldots, w_m; e'_1, \ldots, e'_n),$$

where

- h is a procedure name.
- $w_1, \ldots, w_m$ and $e'_1, \ldots, e'_n$ are called *actual parameters*.
- $w_1, \ldots, w_m$ are distinct variables.
- $e'_1, \ldots, e'_n$ are expressions that do not contain occurrences of the variables $w_1, \ldots, w_m$.
- The free variables of the procedure call are

  $$\mathrm{FV}(h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)) = \{w_1, \ldots, w_m\} \cup \mathrm{FV}(e'_1) \cup \cdots \cup \mathrm{FV}(e'_n)$$

  and the variables modified by the call are $w_1, \ldots, w_m$.

Whenever a procedure name in a procedure call is bound by the same name in a procedure definition, the number of actual parameters in the call (on each side of the semicolon) must equal the number of formal parameters in the definition (on each side of the semicolon).

In the inference rules in the rest of this section, it is assumed that all formal and actual parameter lists meet the restrictions described above.

If a command c contains procedure calls, then the truth of a specification $\{p\}\ c\ \{q\}$ will depend upon the meanings of the procedure names occurring

free in c, or, more abstractly, on a mapping of these names into procedure meanings, which we will call an *environment*. (It is beyond the scope of these notes to give a precise semantic definition of procedure meanings, but the intuitive idea is clear.) Normally, we are not interested in whether a specification holds in all environments, but only in the environments that satisfy certain hypotheses, which may also be described by specifications.

We define a hypothetical specification to have the form

$$\Gamma \vdash \{p\}\ c\ \{q\},$$

where the *context* Γ is a sequence of specifications of the form $\{p_0\}\ c_0\ \{q_0\}, \ldots,$ $\{p_{n-1}\}\ c_{n-1}\ \{q_{n-1}\}$. We say that such a hypothetical specification is true iff $\{p\}\ c\ \{q\}$ holds for every environment in which all of the specifications in Γ hold.

Thus procedure names are distinguished from variables by being implicitly quantified over hypothetical specifications rather than over Hoare triples or assertions.

Before proceding further, we must transform the inference rules we have already developed, so that they deal with hypothetical specifications. Fortunately, this is trivial — one simply adds $\Gamma \vdash$ to all premisses and conclusions that are Hoare triples. For example, the rules (SP) and (SUB) become

- Strengthening Precedent (SP)

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

- Substitution (SUB)

$$\frac{\Gamma \vdash \{p\}\ c\ \{q\}}{\Gamma \vdash \{p/\delta\}\ (c/\delta)\ \{q/\delta\},}$$

  where δ is the substitution $v_1 \to e_1, \ldots, v_n \to e_n$, $v_1, \ldots, v_n$ 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.

Note that substitutions do not affect procedure names.

Next, there is a rule for using hypotheses, which allows one to infer any hypothesis that occurs in the context:

- Hypothesis (HYPO)

$$\frac{}{\Gamma, \{p\}\ c\ \{q\}, \Gamma' \vdash \{p\}\ c\ \{q\}.}$$

Then there are rules that show how procedure declarations give rise to hypotheses about procedure calls. In the nonrecursive case,

- Simple Procedures (SPROC)

$$\frac{\Gamma \vdash \{p\}\ c\ \{q\} \qquad \Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}}{\Gamma \vdash \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},}$$

where h does not occur free in any triple of Γ.

In other words, if one can prove $\{p\}\ c\ \{q\}$ about the body of the definition $h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c$, one can use $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$ as an hypothesis in reasoning about the scope of the definition.

In the recursive case (though only for partial correctness), one can also use the hypothesis in proving $\{p\}\ c\ \{q\}$:

- Simple Recursive Procedures (SRPROC)

$$\frac{\begin{array}{l}\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p\}\ c\ \{q\} \\ \qquad \Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}\end{array}}{\Gamma \vdash \{p'\}\ \mathbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},}$$

where h does not occur free in any triple of Γ.

In essence, one must guess a specification $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$ of the leftside of the definition, and then use this guess as a hypothesis to infer that the rightside of the definition satisfies a similar specification $\{p\}\ c\ \{q\}$.

To keep our exposition straightforward, we have ignored some more general possibilities for defining and reasoning about procedures. Most obviously, we have neglected simultaneous recursion. In addition, we have not dealt with the possibility that a single procedure definition may give rise to more than one useful hypothesis about its calls.

Turning to calls of procedures, one can derive a specialization of (HYPO) in which the command c is a call:

- Call (CALL)

$$\frac{}{\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}.}$$

When used by itself, this rule specifies only procedure calls whose actual parameters match the formal parameters of the corresponding definition. However, one can follow (CALL) with an application of the rule (SUB) of substitution to obtain a specification of any legal (i.e., nonaliasing) call. This allow one to derive the following rule:

- General Call (GCALL)

$$\frac{}{\begin{array}{l}\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \\ \qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

  where δ is a substitution

$$\delta = v_1 \to w_1, \ldots, v_m \to w_m, v'_1 \to e'_1, \ldots, v'_n \to e'_n, v''_1 \to e''_1 \ldots, v''_k \to e''_k,$$

  which acts on all the free variables in $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$, and $w_1, \ldots, w_m$ are distinct variables that do not occur free in the expressions $e'_1 \ldots, e'_n$ or $e''_1 \ldots, e''_k$.

The reader may verify that the restrictions on δ, along with the syntactic restrictions on actual parameters, imply the conditions on the substitution in the rule (SUB).

It is important to note that there may be *ghost variables* in p and q, i.e., variables that are not formal parameters of the procedure, but appear as $v''_1, \ldots, v''_k$ in the substitution δ. One can think of these variables as formal ghost parameters (of the procedure specification, rather than the procedure itself), and the corresponding $e''_1, \ldots, e''_k$ as actual ghost parameters.

Now consider annotated specifications. Each annotated instance of the rule (GCALL) must specify the entire substitution δ explicitly — including the ghost parameters (if any). For this purpose, we introduce *annotated contexts*, which are sequences of *annotated hypotheses*, which have the form

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\},$$

where $v''_1, \ldots, v''_k$ is a list of formal ghost parameters (and all of the formal parameters, including the ghosts, are distinct). (We ignore the possibility of hypotheses about other commands than procedure calls, and assume that each hypothesis in a context describes a different procedure name.)

We write $\hat{\Gamma}$ to denote an annotated context, and Γ to denote the corresponding ordinary context that is obtained by erasing the lists of ghost formal

parameters in each hypothesis. Then we generalize annotation descriptions to have the form

$$\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\},$$

meaning that $\hat{\Gamma} \vdash \mathcal{A}$ is an annotated hypothetical specification proving the hypothetical specification $\Gamma \vdash \{p\}\ c\ \{q\}$.

Then (GCALL) becomes

- General Call (GCALLan)

$$\frac{}{\begin{array}{l}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\}, \hat{\Gamma}' \vdash \\ \qquad h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\{e''_1, \ldots, e''_k\} \gg \\ \qquad\qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

where δ is the substitution

$$\delta = v_1 \to w_1, \ldots, v_m \to w_m, v'_1 \to e'_1, \ldots, v'_n \to e'_n, v''_1 \to e''_1 \ldots, v''_k \to e''_k,$$

which acts on all the free variables in $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$, and $w_1, \ldots, w_m$ are distinct variables that do not occur free in the expressions $e'_1 \ldots, e'_n$ or $e''_1 \ldots, e''_k$.

It is straightforward to annotate procedure definitions — it is enough to provide a list of the formal ghost parameters, along with annotated specifications of the procedure body and of the command in which the procedure is used:

- Simple Procedures (SPROCan)

$$\frac{\begin{array}{l}\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\} \\ \hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\}\end{array}}{\begin{array}{l}\hat{\Gamma} \vdash \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \mathcal{A}\ \mathbf{in}\ \mathcal{A}' \gg \\ \qquad \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},\end{array}}$$

where h does not occur free in any triple of $\hat{\Gamma}$.

In the recursive case, however, the annotation of the procedure body must be complete, since explicit occurrences of p and q are needed to determine the hypothesis describing recursive calls:

- Simple Recursive Procedures (SRPROCan)

$$\frac{\begin{array}{l}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\} \\ \hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\}\end{array}}{\begin{array}{l}\hat{\Gamma} \vdash \textbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \textbf{in}\ \mathcal{A}' \gg \\ \qquad\qquad \{p'\}\ \textbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c'\ \{q'\},\end{array}}$$

  where h does not occur free in any triple of $\hat{\Gamma}$.

We conclude with a simple example involving modified, unmodified, and ghost parameters: a recursive procedure $\mathsf{multfact(r;n)}$ that multiplies r by the factorial of n. The procedure is used to multiply 10 by the factorial of 5:

$$\begin{array}{ll}
\{\mathsf{z} = 10\} & \\
\textbf{letrec}\ \mathsf{multfact(r;n)}\{\mathsf{r}_0\} = & \\
\quad \{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\} & \\
\quad \textbf{if}\ \mathsf{n} = 0\ \textbf{then} & \\
\qquad \{\mathsf{n} = 0 \wedge \mathsf{r} = \mathsf{r}_0\}\ \textbf{skip}\ \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\} & \\
\quad \textbf{else} & \\
\qquad \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} & \\
\qquad \mathsf{r} := \mathsf{n} \times \mathsf{r}\,; & \\
\qquad \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} & \\
\qquad\quad \left.\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\ \mathsf{multfact(r;n-1)}\{\mathsf{n} \times \mathsf{r}_0\} \\ \{\mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\}\end{array}\right\} * \ \mathsf{n} - 1 \geq 0 & \begin{array}{l}(*) \\ \\ (*)\end{array} \\
\qquad \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\} & \\
\quad \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\} & \\
\textbf{in} & \\
\{5 \geq 0 \wedge \mathsf{z} = 10\} & (*) \\
\mathsf{multfact(z;5)}\{10\} & \\
\{\mathsf{z} = 5! \times 10\} & (*)
\end{array}$$

(In fact, our rules for annotation descriptions permit the lines marked (*) to be omitted, but this would make the annotations harder to understand.)

To see how the annotations here determine an actual formal proof, we first note that the application of (SRPROCan) to the **letrec** definition gives rise to the hypothesis

$$\{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\}$$
$$\mathsf{multfact}(\mathsf{r};\mathsf{n})\{\mathsf{r}_0\}$$
$$\{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\}.$$

Now consider the recursive call of multfact. By (GCALLan), the hypothesis entails

$$\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\}$$
$$\mathsf{multfact}(\mathsf{r};\mathsf{n}-1)\{\mathsf{n} \times \mathsf{r}_0\}$$
$$\{\mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\}.$$

Next, since n is not modified by the call $\mathsf{multfact}(\mathsf{r};\mathsf{n}-1)$, the frame rule gives

$$\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = \mathsf{n} \times \mathsf{r}_0 \;*\; \mathsf{n} - 1 \geq 0\}$$
$$\mathsf{multfact}(\mathsf{r};\mathsf{n}-1)\{\mathsf{n} \times \mathsf{r}_0\}$$
$$\{\mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0 \;*\; \mathsf{n} - 1 \geq 0\}.$$

But the assertions here are all pure, so that the separating conjunctions can be replaced by ordinary conjunctions. (In effect, we have used the frame rule as a stand-in for the rule of constancy in Hoare logic.) Then, some obvious properties of ordinary conjunction allow us to strengthen the precondition and weaken the postcondition, to obtain

$$\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\}$$
$$\mathsf{multfact}(\mathsf{r};\mathsf{n}-1)\{\mathsf{n} \times \mathsf{r}_0\}$$
$$\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\}.$$

As for the specification of the main-level call

$$\{5 \geq 0 \wedge \mathsf{z} = 10\}$$
$$\mathsf{multfact}(\mathsf{z};5)\{10\}$$
$$\{\mathsf{z} = 5! \times 10\},$$

by (GCALLan) it is directly entailed by the hypothesis.

In the examples that follow, we will usually be interested in the specification of a procedure itself, rather than of some program that uses the

procedure. In this situation, it is enough to establish the first premiss of (SPROC) or (SRPROC).

4.6 Still More about Annotated Specifications

In this section, we will extend the developments in Section 3.12 to deal with our more complex notion of specifications. The most pervasive changes deal with the introduction of contexts.

We begin by redefining the function erase-annspec and erase-spec to accept hypothetical annotation descriptions and produce hypothetical specifications and annotated specifications:

$$\text{erase-annspec}(\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}) \stackrel{\text{def}}{=} \Gamma \vdash \{p\}\ c\ \{q\}$$

$$\text{erase-spec}(\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}) \stackrel{\text{def}}{=} \hat{\Gamma} \vdash \mathcal{A},$$

and extending the definition of cd to our new annotations:

$$\text{cd}(h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\{e''_1, \ldots, e''_k\}) = h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)$$

$$\begin{aligned}&\text{cd}(\textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \mathcal{A}\ \textbf{in}\ \mathcal{A}')\\ &\qquad = \textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = \text{cd}(\mathcal{A})\ \textbf{in}\ \text{cd}(\mathcal{A}'),\end{aligned}$$

and similarly for **letrec**.

Then Proposition 12 remains true, except that the supposition $\mathcal{A} \gg \{p\}\ c\ \{q\}$ becomes $\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}$.

We also generalize the definitions of left-compl, right-compl, and compl by prefixing $\hat{\Gamma} \vdash$ to all annotation descriptions.

Our next task is to redefine the function Φ. For this purpose, we assume a standard ordering for variables (say, alphabetic ordering), and we (temporarily) redefine the annotated context $\hat{\Gamma}$ to be in a standard form that is uniquely determined by the unannotated Γ. Specifically, $\hat{\Gamma}$ is obtained from Γ by replacing each hypothesis

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$$

by

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\},$$

where $v''_1, \ldots, v''_k$ lists, in the standard ordering, the variables occurring free in p or q that are not among the formal parameters $v_1, \ldots, v_m, v'_1, \ldots, v'_n$. This establishes a one-to-one correspondence between Γ and $\hat{\Gamma}$.

Then we modify the clauses defining Φ in Section 12 by prefixing $\Gamma \vdash$ to all specifications and $\hat{\Gamma} \vdash$ to all annotation descriptions.

Next, we add clauses to describe with our new inference rules. (We ignore (HYPO) and (CALL), but deal with the derived rule (GCALL).)

(GCALL) Suppose $\mathcal{P}$ is

$$\frac{}{\begin{array}{l}\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \\ \qquad\qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

where δ is the substitution

$$\delta = v_1 \to w_1, \ldots, v_m \to w_m, v'_1 \to e'_1, \ldots, v'_n \to e'_n, v''_1 \to e''_1, \ldots, v''_k \to e''_k,$$

which acts on all the free variables in $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$, and $w_1, \ldots, w_m$ are distinct variables that do not occur free in the expressions $e'_1 \ldots, e'_n$ or $e''_1 \ldots, e''_k$. Without loss of generality, we can assume that the domain of δ is exactly the set of variables occurring free in $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}$. Then $\Phi(\mathcal{P})$ is

$$\frac{}{\begin{array}{l}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\}, \hat{\Gamma}' \vdash \\ \qquad\qquad h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\{e''_1, \ldots, e''_k\} \gg \\ \qquad\qquad\qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

where $v''_1, \ldots, v''_k$ are listed in the standard ordering.

(SPROC) Suppose $\mathcal{P}$ is

$$\frac{\mathcal{P}_1[\Gamma \vdash \{p\}\ c\ \{q\}] \qquad \mathcal{P}_2[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}]}{\Gamma \vdash \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},}$$

where h does not occur free in any triple of Γ, and

$$\Phi(\mathcal{P}_1[\Gamma \vdash \{p\}\ c\ \{q\}]) = \mathcal{Q}_1(\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\})$$

$$\begin{array}{l}\Phi(\mathcal{P}_2[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}]) = \\ \quad \mathcal{Q}_2(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\}).\end{array}$$

Then $\Phi(\mathcal{P})$ is

$$\frac{\begin{array}{l}\mathcal{Q}_1(\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}) \\ \mathcal{Q}_2(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\})\end{array}}{\begin{array}{r}\hat{\Gamma} \vdash \textbf{let } h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\} = \mathcal{A} \textbf{ in } \mathcal{A}' \gg \\ \{p'\} \textbf{ let } h(v_1, \ldots, v_m; v_1', \ldots, v_n') = c \textbf{ in } c'\ \{q'\}.\end{array}}$$

(SRPROC) Suppose $\mathcal{P}$ is

$$\frac{\begin{array}{l}\mathcal{P}_1[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\} \vdash \{p\}\ c\ \{q\}] \\ \mathcal{P}_2[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\} \vdash \{p'\}\ c'\ \{q'\}]\end{array}}{\Gamma \vdash \{p'\} \textbf{ let } h(v_1, \ldots, v_m; v_1', \ldots, v_n') = c \textbf{ in } c'\ \{q'\},}$$

where h does not occur free in any triple of Γ, and

$$\begin{array}{l}\mathrm{compl}(\Phi(\mathcal{P}_1[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\} \vdash \{p\}\ c\ \{q\}])) = \\ \quad \mathcal{Q}_1(\hat{\Gamma} \vdash \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\})\end{array}$$

$$\begin{array}{l}\Phi(\mathcal{P}_2[\Gamma, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\} \vdash \{p'\}\ c'\ \{q'\}]) = \\ \quad \mathcal{Q}_2(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\}).\end{array}$$

Then $\Phi(\mathcal{P})$ is

$$\frac{\begin{array}{l}\mathcal{Q}_1(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\}\ \{q\} \vdash \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\}) \\ \mathcal{Q}_2(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\})\end{array}}{\begin{array}{r}\hat{\Gamma} \vdash \textbf{let } h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\} = \{p\}\ \mathcal{A}\ \{q\} \textbf{ in } \mathcal{A}' \gg \\ \{p'\} \textbf{ let } h(v_1, \ldots, v_m; v_1', \ldots, v_n') = c \textbf{ in } c'\ \{q'\}.\end{array}}$$

With this extension, the function Φ satisfies Proposition 13, except that Part 1 of that proposition becomes:

1. *If $\mathcal{P}$ proves $\Gamma \vdash \{p\}\ c\ \{q\}$, then $\Phi(\mathcal{P})$ proves $\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}$ for some annotated specification $\mathcal{A}$.*

Finally, we must extend the function Ψ appropriately. First, we redefine a *premiss from p to q* to be either a hypothetical specification $\Gamma \vdash \{p\}\ c\ \{q\}$

or a verification condition $p \Rightarrow q$, and we require all the hypothetical specifications in a coherent premiss sequence (or the conclusions of a coherent proof sequence) to contain the same context.

The function "code" ignores the contexts in its argument.

Now, for any proper coherent premiss sequence $\mathcal{S}$ from p to q, one can derive the inference rule

$$\frac{\mathcal{S}}{\Gamma \vdash \{p\}\ \mathrm{code}(\mathcal{S})\ \{q\},}$$

where Γ is the common context of the one or more hypothetical specifications in $\mathcal{S}$. Thus, if $\Psi_0(\mathcal{A})$ is a proper coherent proof sequence from p to q, and Γ is the common context of the hypothetical specifications in $\mathrm{concl}(\Psi_0(\mathcal{A}))$, then we may define

$$\Psi(\mathcal{A}) = \frac{\Psi_0(\mathcal{A})}{\Gamma \vdash \{p\}\ \mathrm{code}(\mathrm{concl}(\Psi_0(\mathcal{A})))\ \{q\}.}$$

The function Ψ becomes a map from hypothetical annotated specifications to proofs of hypothetical specifications. In the cases given in Section 3.12, one prefixes $\hat{\Gamma}$ to every annotated specification and Γ to every specification. (We no longer require $\hat{\Gamma}$ to be in standard form, but Γ is still obtained from $\hat{\Gamma}$ by erasing lists of ghost parameters.) For example:

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, v := e\, \{q\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{q/v \to e\}), \frac{}{\Gamma \vdash \{q/v \to e\}\ v := e\ \{q\}}$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ \mathcal{A}_1\ \mathbf{else}\ (\mathcal{A}_2)\ \{q\}) =$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p\}), \frac{\Psi(\hat{\Gamma} \vdash \{p \wedge b\}\ \mathcal{A}_1\ \{q\}) \quad \Psi(\hat{\Gamma} \vdash \{p \wedge \neg b\}\ \mathcal{A}_2\ \{q\})}{\Gamma \vdash \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ \mathrm{cd}(\mathcal{A}_1)\ \mathbf{else}\ \mathrm{cd}(\mathcal{A}_2)\ \{q\}}$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \big\{\mathcal{A}\big\}\, \exists v) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{\exists v.\ p\}), \frac{\Psi(\hat{\Gamma} \vdash \mathcal{A})}{\Gamma \vdash \{\exists v.\ p\}\ c\ \{\exists v.\ q\}}$$

where $\Psi(\hat{\Gamma} \vdash \mathcal{A})$ proves $\Gamma \vdash \{p\}\ c\ \{q\}$.

To extend Ψ_0 to procedure calls, assume that $\hat{\Gamma}$ contains the hypothesis

$\{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\}\ \{q\}$. Then

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, h(w_1,\ldots,w_m;e'_1,\ldots,e'_n)\{e''_1,\ldots,e''_k\}\ \{r\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, h(w_1,\ldots,w_m;e'_1,\ldots,e'_n)\{e''_1,\ldots,e''_k\}), (q/\delta) \Rightarrow r$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, h(w_1,\ldots,w_m;e'_1,\ldots,e'_n)\{e''_1,\ldots,e''_k\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p/\delta\}),\ \overline{\Gamma \vdash \{p/\delta\}\ h(w_1,\ldots,w_m;e'_1,\ldots,e'_n)\ \{q/\delta\}},$$

where δ is the substitution

$$\delta = v_1 \to w_1,\ldots,v_m \to w_m, v'_1 \to e'_1,\ldots,v'_n \to e'_n, v''_1 \to e''_1 \ldots, v''_k \to e''_k.$$

Then for simple procedure definitions, we have:

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{let}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \mathcal{A}\ \textbf{in}\ \mathcal{A}'\ \{r\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{let}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \mathcal{A}\ \textbf{in}\ \mathcal{A}'), q' \Rightarrow r$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{let}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \mathcal{A}\ \textbf{in}\ \mathcal{A}') =$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p'\}),\ \dfrac{\begin{array}{l}\Psi(\hat{\Gamma} \vdash \{p\}\ \mathcal{A}\ \{q\})\\ \Psi(\hat{\Gamma}, \{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\}\ \{q\} \vdash \mathcal{A}')\end{array}}{\hat{\Gamma} \vdash \{p'\}\ \textbf{let}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n) = \mathrm{cd}(\mathcal{A})\ \textbf{in}\ c'\ \{q'\}},$$

where

$$\Psi(\hat{\Gamma} \vdash \mathcal{A}) \text{ proves } \Gamma \vdash \{p\}\ c\ \{q\}$$

$$\Psi(\hat{\Gamma}, \{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\}\ \{q\} \vdash \mathcal{A}') \text{ proves } \Gamma, \{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\},$$

and for simple recursive procedures:

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{letrec}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \textbf{in}\ \mathcal{A}'\ \{r\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{letrec}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \textbf{in}\ \mathcal{A}'), q' \Rightarrow r$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \textbf{letrec}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \textbf{in}\ \mathcal{A}') =$$

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p'\}),\ \dfrac{\begin{array}{l}\Psi(\hat{\Gamma}, \{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\}\ \{q\} \vdash \{p\}\ \mathcal{A}\ \{q\})\\ \Psi(\hat{\Gamma}, \{p\}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n)\{v''_1,\ldots,v''_k\}\ \{q\} \vdash \mathcal{A}')\end{array}}{\hat{\Gamma} \vdash \{p'\}\ \textbf{letrec}\ h(v_1,\ldots,v_m;v'_1,\ldots,v'_n) = \mathrm{cd}(\mathcal{A})\ \textbf{in}\ c'\ \{q'\}},$$

where

$$\Psi(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}') \text{ proves}$$
$$\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}.$$

Then Proposition 14 generalizes to:

If $\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\}$ is provable by $\mathcal{Q}$, then there is a a proper coherent proof sequence $\mathcal{R}$ from p to q, in which each hypothetical specification contains the context $\hat{\Gamma}$, such that:

1. *If $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p\})$ is defined, then:*
   - *(a)* $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \mathcal{A})$ *and* $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \mathcal{A}\, \{r\})$ *are defined,*
   - *(b)* $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \mathcal{A}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p\}), \mathcal{R}\,,$$
   - *(c)* $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \mathcal{A}\, \{r\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p\}), \mathcal{R}, q \Rightarrow r\,,$$
2. $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}) = \mathcal{R}\,,$$
3. $\Psi(\hat{\Gamma} \vdash \mathcal{A})$ *is a proof of* $\Gamma \vdash \{p\}\ c\ \{q\}\,,$
4. *The verification conditions in $\hat{\Gamma} \vdash \Psi(\mathcal{A})$ are the verification conditions in erase-annspec$(\mathcal{Q})$.*

4.7 An Extended Example: Sorting by Merging

Returning to the subject of list processing, we now consider the use of recursive procedures to sort a list by merging.

This example is partly motivated by a pet peeve of the author: Unlike most textbook examples of sorting lists by merging, this program has the $n \log n$ efficiency that one expects of any serious sorting program. (On the other hand, unlike programs that sort arrays by merging, it is an in-place algorithm.)

We begin by defining some simple concepts about sequences that will allow us to assert and prove that one sequence is a sorted rearrangement of another.

4.7.1 Some Concepts about Sequences

The *image* $\{\alpha\}$ of a sequence α is the set

$$\{\, \alpha_i \mid 1 \leq i \leq \#\alpha \,\}$$

of values occurring as components of α. It satisfies the laws:

$$\{\epsilon\} = \{\} \tag{4.1}$$

$$\{[x]\} = \{x\} \tag{4.2}$$

$$\{\alpha{\cdot}\beta\} = \{\alpha\} \cup \{\beta\} \tag{4.3}$$

$$\#\{\alpha\} \leq \#\alpha. \tag{4.4}$$

If ρ is a binary relation between values (e.g., integers), then ρ^* is the binary relation between sets of values such that

$$S \;\rho^*\; T \text{ iff } \forall x \in S.\ \forall y \in T.\ x \;\rho\; y.$$

Pointwise extension satisfies the laws:

$$S' \subseteq S \wedge S \;\rho^*\; T \;\Rightarrow\; S' \;\rho^*\; T \tag{4.5}$$

$$T' \subseteq T \wedge S \;\rho^*\; T \;\Rightarrow\; S \;\rho^*\; T' \tag{4.6}$$

$$\{\} \;\;\rho^*\;\; T \tag{4.7}$$

$$S \;\;\rho^*\;\; \{\} \tag{4.8}$$

$$\{x\} \;\rho^*\; \{y\} \;\Leftrightarrow\; x \;\rho\; y \tag{4.9}$$

$$(S \cup S') \;\rho^*\; T \;\Leftrightarrow\; S \;\rho^*\; T \wedge S' \;\rho^*\; T \tag{4.10}$$

$$S \;\rho^*\; (T \cup T') \;\Leftrightarrow\; S \;\rho^*\; T \wedge S \;\rho^*\; T'. \tag{4.11}$$

It is frequently useful to apply pointwise extension to a single side of a relation. For this purpose, rather than giving additional definitions and laws, it is sufficient to introduce the following abbreviations:

$$x \;\rho^*\; T \stackrel{\text{def}}{=} \{x\} \;\rho^*\; T \qquad S \;\rho^*\; y \stackrel{\text{def}}{=} S \;\rho^*\; \{y\}$$

We write **ord** α if the sequence α is ordered in nonstrict increasing order. Then **ord** satisfies

$$\#\alpha \leq 1 \;\Rightarrow\; \mathbf{ord}\ \alpha \tag{4.12}$$

$$\mathbf{ord}\ \alpha{\cdot}\beta \;\Leftrightarrow\; \mathbf{ord}\ \alpha \wedge \mathbf{ord}\ \beta \wedge \{\alpha\} \leq^* \{\beta\} \tag{4.13}$$

$$\mathbf{ord}\ [x]{\cdot}\alpha \;\Rightarrow\; x \leq^* \{[x]{\cdot}\alpha\} \tag{4.14}$$

$$\mathbf{ord}\ \alpha{\cdot}[x] \;\Rightarrow\; \{\alpha{\cdot}[x]\} \leq^* x. \tag{4.15}$$

We say that a sequence β is a *rearrangement* of a sequence α, written $\beta \sim \alpha$, iff there is a permutation (i.e., isomorphism) ϕ, from the domain (1 to $\#\beta$) of β to the domain (1 to $\#\alpha$) of α such that

$$\forall k.\ 1 \leq k \leq \#\beta \text{ implies } \beta_k = \alpha_{\phi(k)}.$$

Then

$$\alpha \sim \alpha \tag{4.16}$$

$$\alpha \sim \beta \;\Rightarrow\; \beta \sim \alpha \tag{4.17}$$

$$\alpha \sim \beta \wedge \beta \sim \gamma \;\Rightarrow\; \alpha \sim \gamma \tag{4.18}$$

$$\alpha \sim \alpha' \wedge \beta \sim \beta' \;\Rightarrow\; \alpha{\cdot}\beta \sim \alpha'{\cdot}\beta' \tag{4.19}$$

$$\alpha{\cdot}\beta \sim \beta{\cdot}\alpha \tag{4.20}$$

$$\alpha \sim \beta \;\Rightarrow\; \{\alpha\} = \{\beta\}. \tag{4.21}$$

$$\alpha \sim \beta \;\Rightarrow\; \#\alpha = \#\beta. \tag{4.22}$$

4.7.2 The Sorting Procedure mergesort

The basic idea behind sorting by merging is to divide the input list segment into two roughly equal halves, sort each half recursively, and then merge the results. Unfortunately, however, one cannot divide a list segment into two halves efficiently.

A way around this difficulty is to give the lengths of the input segments to the commands for sorting and merging as explicit numbers. Thus such a segment is determined by a pointer to a list that begins with that segment, along with the length of the segment.

Suppose we define the abbreviation

$$\mathsf{lseg}\ \alpha\,(e, -) \stackrel{\mathrm{def}}{=} \exists x.\ \mathsf{lseg}\ \alpha\,(e, x).$$

Then we will define a sorting procedure $\mathsf{mergesort}$ satisfying the hypothesis

$$\begin{aligned} H_{\mathsf{mergesort}} \stackrel{\mathrm{def}}{=}\ & \{\mathsf{lseg}\ \alpha\,(\mathsf{i}, \mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\} \\ & \mathsf{mergesort}(\mathsf{i}, \mathsf{j}; \mathsf{n})\{\alpha, \mathsf{j}_0\} \\ & \{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\}. \end{aligned}$$

Here α and j_0 are ghost parameters — α is used in the postcondition to show that the output list segment represents a rearrangement of the input, while j_0 is used to show that $\mathsf{mergesort}$ will set j to the remainder of the input list following the segment to be sorted.

The subsidiary processs of merging will be performed by a procedure merge that satisfies

$$\begin{aligned} H_{\mathsf{m}erge} \stackrel{\mathrm{def}}{=}\ & \{(\mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\ & \qquad * \ (\mathsf{lseg}\ \beta_2\,(\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\} \\ & \mathsf{merge}(\mathsf{i}; \mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2})\{\beta_1, \beta_2\} \\ & \{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta\}. \end{aligned}$$

Here β_1 and β_2 are ghost parameters used to show that the output segment represents a rearrangement of the concatenation of the inputs.

Notice that we assume both the input segments for the sorting and merging commands are nonempty. (We will see that this avoids unnecessary testing since the recursive calls of the sorting command will never be given an empty segment.)

4.7.3 A Proof for mergesort

$$\begin{aligned} H_{\mathsf{mergesort}}, H_{\mathsf{merge}} \vdash\quad & \{\mathsf{lseg}\ \alpha\,(\mathsf{i}, \mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\} \\ & \mathbf{if}\ \mathsf{n} = 1\ \mathbf{then} \\ & \quad \{\mathsf{lseg}\ \alpha\,(\mathsf{i}, -) \wedge \mathbf{ord}\ \alpha \wedge \mathsf{i} \mapsto -, \mathsf{j}_0\} \\ & \quad \mathsf{j} := [\mathsf{i} + 1] \\ & \quad \{\mathsf{lseg}\ \alpha\,(\mathsf{i}, -) \wedge \mathbf{ord}\ \alpha \wedge \mathsf{j} = \mathsf{j}_0\} \\ & \mathbf{else} \\ & \quad \vdots \end{aligned}$$

$$
\begin{array}{l}
\quad\vdots \\
\textbf{else newvar}\ \mathsf{n1}\ \textbf{in newvar}\ \mathsf{n2}\ \textbf{in newvar}\ \mathsf{i1}\ \textbf{in newvar}\ \mathsf{i2}\ \textbf{in} \\
\quad \big(\mathsf{n1} := \mathsf{n} \div 2\ ; \mathsf{n2} := \mathsf{n} - \mathsf{n1}\ ; \mathsf{i1} := \mathsf{i}\ ; \\
\quad \{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2)\ *\ \mathsf{lseg}\ \alpha_2\ (\mathsf{i}_2, \mathsf{j}_0)) \\
\qquad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\} \\
\quad \left.\begin{array}{l}
\left.\begin{array}{l}
\{\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\
\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i1}, -) \wedge \beta \sim \alpha_1 \wedge \mathbf{ord}\ \beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\
\{\exists \beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}
\end{array}\right\} \\
\quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\
\qquad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)
\end{array}\right\} \exists \alpha_1, \alpha_2, \mathsf{i}_2 \\
\quad \{\exists \alpha_1, \alpha_2, \beta_1.\ (\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -)\ *\ \mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0)) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \\
\qquad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\} \\
\quad \left.\begin{array}{l}
\left.\begin{array}{l}
\{\mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0) \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n}_2 \geq 1\} \\
\ \mathsf{mergesort}(\mathsf{i2}, \mathsf{j}; \mathsf{n2})\{\alpha_2, \mathsf{j}_0\}\ ; \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i2}, -) \wedge \beta \sim \alpha_2 \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\} \\
\{\exists \beta_2.\ \mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \beta_2 \sim \alpha_2 \wedge \mathbf{ord}\ \beta_2 \wedge \mathsf{j} = \mathsf{j}_0\}
\end{array}\right\} \\
\quad *\ (\mathsf{lseg}\ \beta_1(\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \\
\qquad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)
\end{array}\right\} \exists \alpha_1, \alpha_2, \beta_1 \\
\quad \{\exists \alpha_1, \alpha_2, \beta_1, \beta_2.\ ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\qquad *\ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \beta_2 \sim \alpha_2 \wedge \mathbf{ord}\ \beta_2 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \\
\qquad \wedge \alpha = \alpha_1{\cdot}\alpha_2 \wedge \mathsf{j} = \mathsf{j}_0\} \\
\quad \{\exists \beta_1, \beta_2.\ ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\qquad *\ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \wedge \alpha \sim \beta_1{\cdot}\beta_2 \wedge \mathsf{j} = \mathsf{j}_0\} \\
\quad \left.\begin{array}{l}
\left.\begin{array}{l}
\{(\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\quad *\ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\} \\
\mathsf{merge}(\mathsf{i}; \mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2})\{\beta_1, \beta_2\} \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta\}
\end{array}\right\} \\
\qquad\qquad\qquad\qquad *\ (\mathbf{emp} \wedge \alpha \sim \beta_1{\cdot}\beta_2 \wedge \mathsf{j} = \mathsf{j}_0)
\end{array}\right\} \exists \beta_1, \beta_2 \\
\quad \{\exists \beta_1, \beta_2, \beta.\ \mathsf{lseg}\ \beta(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta \wedge \alpha \sim \beta_1{\cdot}\beta_2 \wedge \mathsf{j} = \mathsf{j}_0\}\big) \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{array}
$$

Just after the variable declarations in this procedure, some subtle reasoning about arithmetic is needed. To determine the division of the input list segment, the variables $\mathsf{n1}$ and $\mathsf{n2}$ must be set to two positive integers whose sum is n. (Moreover, for efficiency, these variables must be nearly equal.)

At this point, the length n of the input list segment is at least two. Then $2 \leq \mathsf{n}$ and $0 \leq \mathsf{n} - 2$, so that $2 \leq \mathsf{n} \leq 2 \times \mathsf{n} - 2$, and since division by two is monotone:

$$1 = 2 \div 2 \leq \mathsf{n} \div 2 \leq (2 \times \mathsf{n} - 2) \div 2 = \mathsf{n} - 1.$$

Thus if $\mathsf{n1} = \mathsf{n} \div 2$ and $\mathsf{n2} = \mathsf{n} - \mathsf{n1}$, we have

$$1 \leq \mathsf{n1} \leq \mathsf{n} - 1 \qquad 1 \leq \mathsf{n2} \leq \mathsf{n} - 1 \qquad \mathsf{n1} + \mathsf{n2} = \mathsf{n}.$$

The reasoning about procedure calls is expressed very concisely. Consider, for example, the first call of $\mathsf{mergesort}$. From the hypothesis $H_{\mathsf{m}ergesort}$, (GCALLan) is used to infer

$$\begin{aligned}&\{\mathsf{lseg}\ \alpha_1\,(\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\}\\ &\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\\ &\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i1}, -) \wedge \beta \sim \alpha_1 \wedge \mathbf{ord}\ \beta \wedge \mathsf{i2} = \mathsf{i}_2\}\}.\end{aligned}$$

Then β is renamed β_1 in the postcondition:

$$\begin{aligned}&\{\mathsf{lseg}\ \alpha_1\,(\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\}\\ &\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\\ &\{\exists\beta_1.\ \mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}.\end{aligned}$$

Next, the frame rule is used to infer

$$\begin{aligned}&\{(\mathsf{lseg}\ \alpha_1\,(\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)\\ &\quad * \ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0)\\ &\quad\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\}\\ &\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\\ &\{(\exists\beta_1.\ \mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2)\\ &\quad * \ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0)\\ &\quad\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\}.\end{aligned}$$

Then the rule (EQ) for existential quantification gives

$$\begin{aligned}
&\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \land \#\alpha_1 = \mathsf{n1} \land \mathsf{n1} \geq 1) \\
&\quad * \ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\
&\qquad \land \#\alpha_1 = \mathsf{n1} \land \mathsf{n1} \geq 1 \land \#\alpha_2 = \mathsf{n2} \land \mathsf{n2} \geq 1 \land \alpha = \alpha_1 {\cdot} \alpha_2)\} \\
&\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\
&\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\exists \beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \land \beta_1 \sim \alpha_1 \land \mathbf{ord}\ \beta_1 \land \mathsf{i2} = \mathsf{i}_2) \\
&\quad * \ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\
&\qquad \land \#\alpha_1 = \mathsf{n1} \land \mathsf{n1} \geq 1 \land \#\alpha_2 = \mathsf{n2} \land \mathsf{n2} \geq 1 \land \alpha = \alpha_1 {\cdot} \alpha_2)\}.
\end{aligned}$$

Finally, $\mathsf{i2} = \mathsf{i}_2$ is used to eliminate i_2 in the postcondition, and pure terms are rearranged in both the pre- and postconditions:

$$\begin{aligned}
&\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2)\ *\ \mathsf{lseg}\ \alpha_2\ (\mathsf{i}_2, \mathsf{j}_0)) \\
&\quad \land \#\alpha_1 = \mathsf{n1} \land \mathsf{n1} \geq 1 \land \#\alpha_2 = \mathsf{n2} \land \mathsf{n2} \geq 1 \land \alpha = \alpha_1 {\cdot} \alpha_2\} \\
&\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\
&\{\exists \alpha_1, \alpha_2, \beta_1.\ ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -)\ *\ (\mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0)) \land \beta_1 \sim \alpha_1 \land \mathbf{ord}\ \beta_1 \\
&\quad \land \#\alpha_1 = \mathsf{n1} \land \mathsf{n1} \geq 1 \land \#\alpha_2 = \mathsf{n2} \land \mathsf{n2} \geq 1 \land \alpha = \alpha_1 {\cdot} \alpha_2\}.
\end{aligned}$$

4.7.4 A Proof for merge

Our program for merging is more complex than necessary, in order to avoid unnecessary resetting of pointers. For this purpose, it keeps track of which of the two list segments remaining to be merged is pointed to by the end of the list segment of already-merged items.

The procedure merge is itself nonrecursive, but it calls a subsidiary tail-recursive procedure named merge1, which is described by the hypothesis

$$
\begin{array}{l}
H_{\mathsf{merge1}} = \\
\quad \{\exists \beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad (\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{i1}) \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -) \ * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2 \\
\qquad \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta \\
\qquad \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1} \leq \mathsf{a2}\} \\
\quad \mathsf{merge1}(;\mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2}, \mathsf{a2})\{\beta_1, \beta_2, \mathsf{i}\} \\
\quad \{\exists \beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}.
\end{array}
$$

(Note that merge1 does not modify any variables.)

The body of merge1 is verified by the annotated specification:

$$
\begin{array}{l}
H_{\mathsf{merge1}} \vdash \\
\quad \{\exists \beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad (\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{i1}) \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -) \ * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2 \\
\qquad \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta \\
\qquad \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1} \leq \mathsf{a2}\} \\
\quad \mathbf{if}\ \mathsf{n1} = 1\ \mathbf{then}\ [\mathsf{i1} + 1] := \mathsf{i2} \\
\qquad \{\exists \beta, \mathsf{a1}, \mathsf{j2}, \gamma_2. \\
\qquad\quad (\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{i1}) \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{i2} \ * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
\qquad\quad \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2 \\
\qquad\quad \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1} \leq \mathsf{a2}\} \\
\qquad\quad \vdots
\end{array}
$$

else newvar j **in newvar** a1 **in**

$\big($n1 := n1 − 1 ; j := i1 ; i1 := [j + 1] ; a1 := [i1] ;

$\{\exists\beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2.$
$(\mathsf{lseg}\ \beta\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}', \mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1'\,(\mathsf{j1}, -)$
$*\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -))$
$\wedge \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$
$\wedge\, \mathbf{ord}\,(\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta$
$\wedge\, \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1}' \leq \mathsf{a2}\}$

$\{\exists\beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2.$
$(\mathsf{lseg}\ \beta\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}', \mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1'\,(\mathsf{j1}, -)$
$*\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -))$
$\wedge \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge (\beta{\cdot}\mathsf{a1}'){\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$
$\wedge\, \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\beta{\cdot}\mathsf{a1}') \wedge \{\beta{\cdot}\mathsf{a1}'\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}$

if a1 ≤ a2 **then**

$\{\exists\beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2.$
$(\mathsf{lseg}\ \beta\,(\mathsf{i},\mathsf{i1})\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -)\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -))$
$\wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$
$\wedge\, \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta$
$\wedge\, \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1} \leq \mathsf{a2}\}$

merge1(; n1, n2, i1, i2, a2)$\{\beta_1, \beta_2, \mathsf{i}\}$

$\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}$

else ([j + 1] := i2 ;

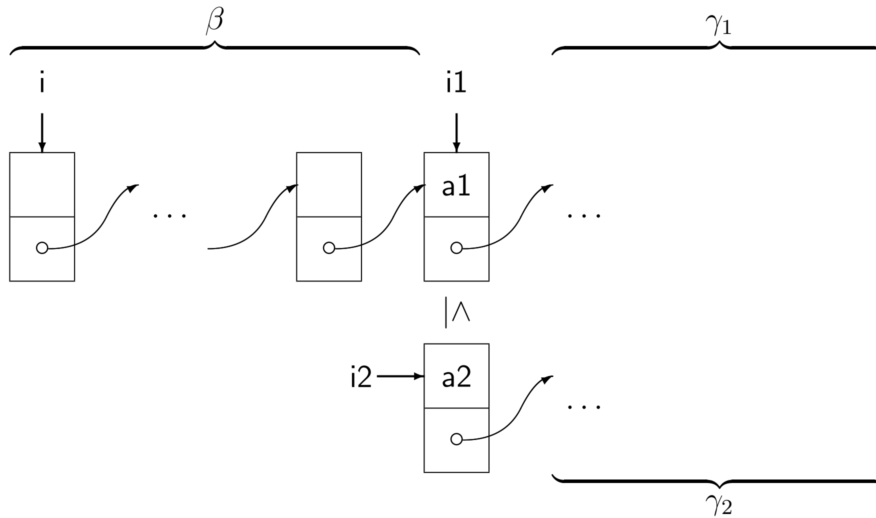
$\{\exists\beta, \mathsf{a2}, \mathsf{j2}, \gamma_2, \mathsf{j1}, \gamma_1.$
$(\mathsf{lseg}\ \beta\,(\mathsf{i},\mathsf{i2})\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -))$
$\wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \beta{\cdot}\mathsf{a2}{\cdot}\gamma_2{\cdot}\mathsf{a1}{\cdot}\gamma_1 \sim \beta_2{\cdot}\beta_1$
$\wedge\, \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,\beta$
$\wedge\, \{\beta\} \leq^* \{\mathsf{a2}{\cdot}\gamma_2\} \cup \{\mathsf{a1}{\cdot}\gamma_1\} \wedge \mathsf{a2} \leq \mathsf{a1}\}$

merge1(; n2, n1, i2, i1, a1))$\{\beta_2, \beta_1, \mathsf{i}\}\big)$

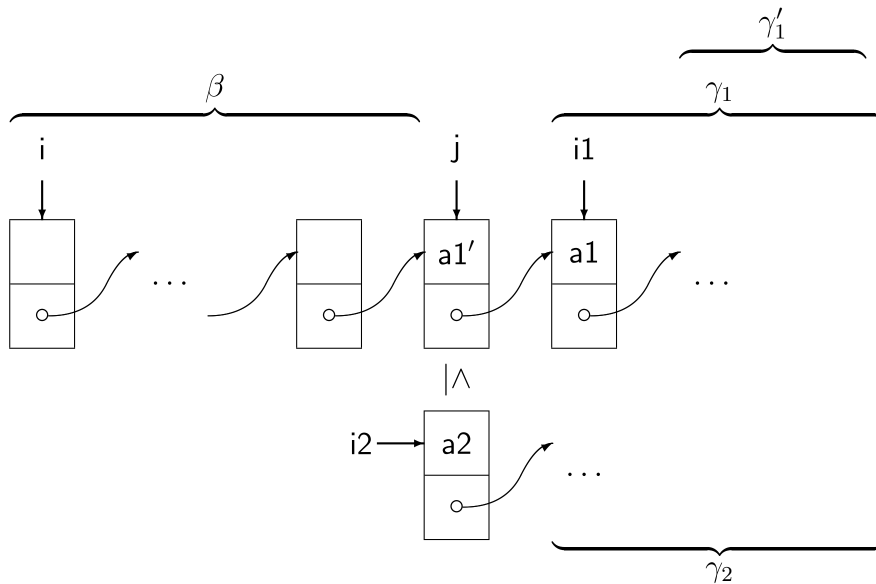
$\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}$

$\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}$

When merge1 is called, the precondition of H_{merge1} describes the situation:



In the **else** clause of the outer conditional command, we know that the γ_1 is nonempty. The effect of the variable declarations, assignments, and lookups is to move i1 and a1 one step down the list segment representing γ_1, and to save the old values of these variables in the program variable j and the existentially quantified variable a1. The remainder of the sequence γ after the first element a1 is denoted by the existentially quantifier γ':



This situation is described by the first assertion in the **else** clause. The step to the next assertion is obtained by the following argument about ordering:

$$\left.\begin{array}{l}\mathbf{ord}\,(\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta \\ \quad \wedge \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\} \wedge \mathsf{a1}' \leq \mathsf{a2}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\beta{\cdot}\mathsf{a1}') \\ \qquad \wedge \{\beta{\cdot}\mathsf{a1}'\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}\end{array}\right.$$

since

1.	$\mathbf{ord}\,(\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1')$	(assumption)
*2.	$\mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1')$	(4.13),1
3.	$\mathsf{a1}' \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\}$	(4.13),1
4.	$\mathsf{a1}' \leq \mathsf{a2}$	(assumption)
*5.	$\mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2)$	(assumption)
6.	$\mathsf{a1}' \leq^* \{\mathsf{a2}{\cdot}\gamma_2\}$	(4.14),4,5
7.	$\mathsf{a1}' \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}$	(4.11),3,6
8.	$\{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}$	(assumption)
9.	$\{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}$	(4.3),(4.6),8
*10.	$\{\beta{\cdot}\mathsf{a1}'\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}$	(4.10),(4.3),7,9
11.	$\mathbf{ord}\,\beta$	(assumption)
12.	$\mathbf{ord}\,\mathsf{a1}'$	(4.12)
13.	$\{\beta\} \leq^* \mathsf{a1}'$	(4.3),(4.6),8
*14.	$\mathbf{ord}\,(\beta{\cdot}\mathsf{a1}')$	(4.13),11,12,13

(Here the asterisks indicate conclusions.)

After the test of $\mathsf{a1} \leq \mathsf{a2}$, the pointer at the end of the list segment representing $\beta{\cdot}\mathsf{a1}'$ is either left unchanged at $\mathsf{i1}$ or reset to $\mathsf{i2}$. Then the meaning of the existentially quantified β and γ_1 are taken to be the former meanings of $\beta{\cdot}\mathsf{a1}'$ and γ_1', and either $\mathsf{a1}$ or $\mathsf{a2}$ is existentially quantified (since its value will not be used further in the program).

4.7.5 merge with goto commands

By using **goto** commands, it is possible to program merge without recursive calls:

$$
\begin{array}{l}
\mathsf{merge}(\mathsf{i};\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2})\{\beta_1,\beta_2\} = \\
\quad \textbf{newvar}\ \mathsf{a1}\ \textbf{in}\ \textbf{newvar}\ \mathsf{a2}\ \textbf{in}\ \textbf{newvar}\ \mathsf{j}\ \textbf{in} \\
\quad \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ; \\
\quad \textbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \textbf{then}\ \mathsf{i} := \mathsf{i1}\ ;\ \textbf{goto}\ \ell 1\ \textbf{else}\ \mathsf{i} := \mathsf{i2}\ ;\ \textbf{goto}\ \ell 2\ ; \\
\ell 1\!: \textbf{if}\ \mathsf{n1} = 1\ \textbf{then}\ [\mathsf{i1}+1] := \mathsf{i2}\ ;\ \textbf{goto}\ \mathsf{out}\ \textbf{else} \\
\quad \mathsf{n1} := \mathsf{n1} - 1\ ;\ \mathsf{j} := \mathsf{i1}\ ;\ \mathsf{i1} := [\mathsf{j}+1]\ ;\ \mathsf{a1} := [\mathsf{i1}]; \\
\quad \textbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \textbf{then}\ \textbf{goto}\ \ell 1\ \textbf{else}\ [\mathsf{j}+1] := \mathsf{i2}\ ;\ \textbf{goto}\ \ell 2\ ; \\
\ell 2\!: \textbf{if}\ \mathsf{n2} = 1\ \textbf{then}\ [\mathsf{i2}+1] := \mathsf{i1}\ ;\ \textbf{goto}\ \mathsf{out}\ \textbf{else} \\
\quad \mathsf{n2} := \mathsf{n2} - 1\ ;\ \mathsf{j} := \mathsf{i2}\ ;\ \mathsf{i2} := [\mathsf{j}+1]\ ;\ \mathsf{a2} := [\mathsf{i2}]; \\
\quad \textbf{if}\ \mathsf{a2} \leq \mathsf{a1}\ \textbf{then}\ \textbf{goto}\ \ell 2\ \textbf{else}\ [\mathsf{j}+1] := \mathsf{i1}\ ;\ \textbf{goto}\ \ell 1\ ; \\
\mathsf{out}\!: \big)
\end{array}
$$

The absence of recursive calls makes this procedure far more efficient than that given in the preceeding section. In the author's opinion, it is also easier to understand (when properly annotated).

Although we have not formalized the treatment of **goto**'s and labels in separation logic (or Hoare logic), it is essentially straightforward (except for jumps out of blocks or procedure bodies). One associates an assertion with each label that should be true whenever control passes to the label.

When a command is labeled, its precondition is associated with the label. When the end of a block or a procedure body is labeled, the postcondition of the block or procedure body is associated with the label.

The assertion associated with a label becomes the precondition of every **goto** command that addresses the label. The postcondition of **goto** commands is **false**, since these commands never returns control.

The following is an annotated specification of the procedure. The assertions for the labels $\ell 1$ and $\ell 2$ correspond to the preconditions of the two recursive calls in the procedure in the previous section. The assertion for the label out is the postcondition of the procedure body.

$$
\begin{array}{l}
\mathsf{merge}(\mathsf{i};\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2})\{\beta_1,\beta_2\} = \\
\quad \{(\mathsf{lseg}\ \beta_1\ (\mathsf{i1},-) \land \mathbf{ord}\ \beta_1 \land \#\beta_1 = \mathsf{n1} \land \mathsf{n1}{\geq}1) \\
\qquad * (\mathsf{lseg}\ \beta_2\ (\mathsf{i2},-) \land \mathbf{ord}\ \beta_2 \land \#\beta_2 = \mathsf{n2} \land \mathsf{n2}{\geq}1)\} \\
\quad \mathbf{newvar}\ \mathsf{a1}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{a2}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{j}\ \mathbf{in} \\
\quad \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ; \\
\quad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ \mathsf{i} := \mathsf{i1}\ ;\ \mathbf{goto}\ \ell1\ \mathbf{else}\ \mathsf{i} := \mathsf{i2}\ ;\ \mathbf{goto}\ \ell2\ ; \\
\\
\ell1\text{:}\ \{\exists\beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad (\mathsf{lseg}\ \beta\ (\mathsf{i},\mathsf{i1})\ *\ \mathsf{i1} \mapsto \mathsf{a1},\mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1},-) \\
\qquad\quad *\ \mathsf{i2} \mapsto \mathsf{a2},\mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2},-)) \\
\qquad \land \#\gamma_1 = \mathsf{n1} - 1 \land \#\gamma_2 = \mathsf{n2} - 1 \\
\qquad \land \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \land \mathbf{ord}\ (\mathsf{a1}{\cdot}\gamma_1) \land \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \\
\qquad \land \mathbf{ord}\ \beta \land \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \land \mathsf{a1} \leq \mathsf{a2}\} \\
\quad \mathbf{if}\ \mathsf{n1} = 1\ \mathbf{then}\ [\mathsf{i1}+1] := \mathsf{i2}\ ;\ \mathbf{goto}\ \mathsf{out}\ \mathbf{else} \\
\quad \mathsf{n1} := \mathsf{n1} - 1\ ;\ \mathsf{j} := \mathsf{i1}\ ;\ \mathsf{i1} := [\mathsf{j}+1]\ ;\ \mathsf{a1} := [\mathsf{i1}]; \\
\quad \{\exists\beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2. \\
\qquad (\mathsf{lseg}\ \beta\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}',\mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1},\mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1'\ (\mathsf{j1},-) \\
\qquad\quad *\ \mathsf{i2} \mapsto \mathsf{a2},\mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2},-)) \\
\qquad \land \#\gamma_1' = \mathsf{n1} - 1 \land \#\gamma_2 = \mathsf{n2} - 1 \\
\qquad \land \beta{\cdot}\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \land \mathbf{ord}\ (\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \land \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \\
\qquad \land \mathbf{ord}\ \beta \land \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \land \mathsf{a1}' \leq \mathsf{a2}\} \\
\quad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ \mathbf{goto}\ \ell1\ \mathbf{else}\ [\mathsf{j}+1] := \mathsf{i2}\ ;\ \mathbf{goto}\ \ell2\ ; \\
\\
\qquad \vdots
\end{array}
$$

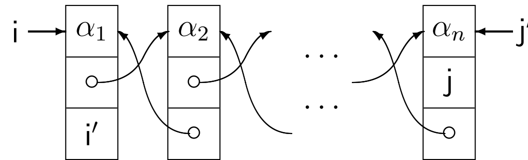
$$\vdots$$

$$\ell 2\colon \{\exists \beta, \mathsf{a2}, \mathsf{j2}, \gamma_2, \mathsf{j1}, \gamma_1.$$
$$(\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{i2}) \ast \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ast \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)$$
$$\ast\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ast \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -))$$
$$\wedge\ \#\gamma_2 = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1$$
$$\wedge\ \beta{\cdot}\mathsf{a2}{\cdot}\gamma_2{\cdot}\mathsf{a1}{\cdot}\gamma_1 {\sim} \beta_2{\cdot}\beta_1 \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1)$$
$$\wedge\ \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a2}{\cdot}\gamma_2\} \cup \{\mathsf{a1}{\cdot}\gamma_1\} \wedge \mathsf{a2} \leq \mathsf{a1}\}$$
$$\mathbf{if}\ \mathsf{n2} = 1\ \mathbf{then}\ [\mathsf{i2} + 1] := \mathsf{i1}\ ;\ \mathbf{goto}\ \mathsf{out}\ \mathbf{else}$$
$$\mathsf{n2} := \mathsf{n2} - 1\ ;\ \mathsf{j} := \mathsf{i2}\ ;\ \mathsf{i2} := [\mathsf{j} + 1]\ ;\ \mathsf{a2} := [\mathsf{i2}];$$
$$\{\exists \beta, \mathsf{a2}', \mathsf{j2}, \gamma_2', \mathsf{j1}, \gamma_1.$$
$$(\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{a2}', \mathsf{i2} \ast \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ast \mathsf{lseg}\ \gamma_2'\,(\mathsf{j2}, -)$$
$$\ast\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ast \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -))$$
$$\wedge\ \#\gamma_2' = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1$$
$$\wedge\ \beta{\cdot}\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2'{\cdot}\mathsf{a1}{\cdot}\gamma_1 {\sim} \beta_2{\cdot}\beta_1 \wedge \mathbf{ord}\,(\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2') \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1)$$
$$\wedge\ \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2'\} \cup \{\mathsf{a1}{\cdot}\gamma_1\} \wedge \mathsf{a2}' \leq \mathsf{a1}\}$$
$$\mathbf{if}\ \mathsf{a2} \leq \mathsf{a1}\ \mathbf{then\ goto}\ \ell 2\ \mathbf{else}\ [\mathsf{j} + 1] := \mathsf{i1}\ ;\ \mathbf{goto}\ \ell 1\ ;$$

$$\mathsf{out}\colon)$$
$$\{\exists \beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}.$$

4.8 Doubly-Linked List Segments

A doubly-linked list segment is a collection of three-field records with both a forward linkage using the second fields and a backward linkage using the third fields. To capture this concept, we write $\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}')$ to describe the situation

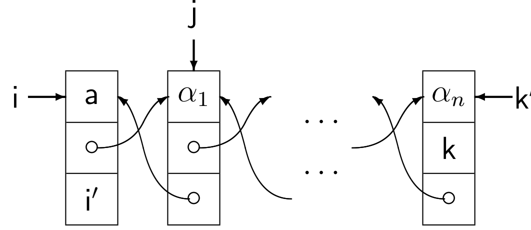


The author knows of no way to verbalize this predicate succinctly — "$(\mathsf{i}, \mathsf{i}')$ to $(\mathsf{j}, \mathsf{j}')$ is a doubly-linked list representing α" is both unwieldly and unlikely to summon the right picture to mind. But, as usual, the concept can be defined precisely by structural induction:

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

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

The second of these equations is illustrated by the following diagram:



Much as with simple list segments, one can prove the properties

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

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

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

One can also define a doubly-linked list by

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

However, one cannot define dlist directly by structural induction, since no proper substructure of a doubly-linked list is a doubly-linked list.

Also as with simple list segments, one can derive emptyness conditions, but now these conditions can be formulated for either the forward or backward linkages:

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

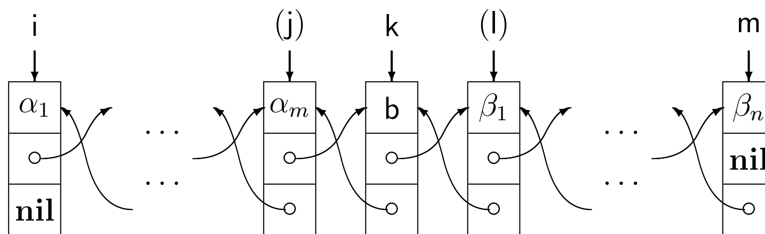
$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{j}' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}))$$

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

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

Nevertheless, when $\mathsf{i} = \mathsf{j} \wedge \mathsf{i}' = \mathsf{j}'$, one may have either an empty segment or nonempty touching (i.e., cyclic) one. (One can also define nontouching segments.)

To illustrate programming with doubly-linked lists, we begin with a program for deleting an element at an arbitrary address k in a list. Here j and l are existentially quantified variables that address the neighbors of the element to be removed, or have the value **nil** if these neighbors do not exist; they are made into program variables by lookup operations at the beginning of the program.



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

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

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

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

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

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

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

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

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

$\mathbf{else}$

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

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

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

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

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

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

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

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

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

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

$\mathbf{else}$

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

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

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

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

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

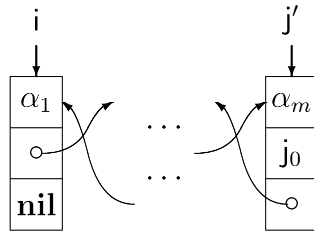
$\{\mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l}, \mathsf{j}, \mathbf{nil}, \mathsf{m})\}$

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

After looking up the values of j and l, and deallocating the record at k, the program either resets the forward pointer in the lefthand neighbor to l (the

forward pointer of the deallocated record) or, if the left subsegment is empty, it sets i to l. Then it performs a symmetric operation on the right subsegment.

To develop another example, we introduce some tiny nonrecursive procedures for examining and changing doubly-linked list segments. We begin with a procedure that examines a left-end segment



and sets the variable j to the last address (or nil) on the forward linkage:

$$
\begin{array}{l}
\mathsf{lookuprpt}(\mathsf{j};\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{j}_0\} = \\
\quad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\} \\
\quad \mathbf{if}\ \mathsf{j}' = \mathbf{nil}\ \mathbf{then} \\
\quad\quad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}') \wedge \mathsf{i} = \mathsf{j}_0\} \\
\quad\quad \mathsf{j} := \mathsf{i} \\
\quad \mathbf{else} \\
\quad\quad \{\exists\alpha',\mathsf{b},\mathsf{k}'.\ \ \alpha = \alpha'{\cdot}\mathsf{b} \wedge (\mathsf{dlseg}\ \alpha'\,(\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}')\ *\ \mathsf{j}' \mapsto \mathsf{b},\mathsf{j}_0,\mathsf{k}')\} \\
\quad\quad \mathsf{j} := [\mathsf{j}' + 1] \\
\quad\quad \{\exists\alpha',\mathsf{b},\mathsf{k}'.\ \ \alpha = \alpha'{\cdot}\mathsf{b} \wedge \\
\quad\quad\quad (\mathsf{dlseg}\ \alpha'\,(\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}')\ *\ \mathsf{j}' \mapsto \mathsf{b},\mathsf{j}_0,\mathsf{k}') \wedge \mathsf{j} = \mathsf{j}_0\} \\
\quad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}') \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{array}
$$

Notice that the parameter list here makes it clear that the procedure call will only modify the variable j, and will not even evaluate the ghost parameters α or j_0. This information is an essential requirement for the procedure; otherwise the specification could be met by the assignment $\mathsf{j} := \mathsf{j}_0$.

Next, we define a procedure that changes a left-end segment so that the

last address in the forward linkage is the value of the variable j:

$$
\begin{array}{l}
\mathsf{setrpt}(\mathsf{i};\mathsf{j},\mathsf{j}')\{\alpha,\mathsf{j}_0\} = \\
\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\} \\
\quad \mathbf{if}\ \mathsf{j}' = \mathbf{nil}\ \mathbf{then} \\
\qquad \{\alpha = \epsilon \wedge \mathbf{emp} \wedge \mathsf{j}' = \mathbf{nil}\} \\
\qquad \mathsf{i} := \mathsf{j} \\
\qquad \{\alpha = \epsilon \wedge \mathbf{emp} \wedge \mathsf{j}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}\} \\
\quad \mathbf{else} \\
\qquad \{\exists \alpha', \mathsf{b}, \mathsf{k}'.\ \alpha = \alpha'{\cdot}\mathsf{b} \wedge (\mathsf{dlseg}\ \alpha'\ (\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}') \ *\ \mathsf{j}' \mapsto \mathsf{b},\mathsf{j}_0,\mathsf{k}')\} \\
\qquad [\mathsf{j}'+1] := \mathsf{j} \\
\qquad \{\exists \alpha', \mathsf{b}, \mathsf{k}'.\ \alpha = \alpha'{\cdot}\mathsf{b} \wedge (\mathsf{dlseg}\ \alpha'\ (\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}') \ *\ \mathsf{j}' \mapsto \mathsf{b},\mathsf{j},\mathsf{k}')\} \\
\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\}.
\end{array}
$$

It is interesting to note that the Hoare triples specifying $\mathsf{lookuprpt}$ and setrpt have the same precondition, and the postcondition of $\mathsf{lookuprpt}$ implies that of setrpt, Thus, by weakening consequent, we find that $\mathsf{lookuprpt}$ satisfies the same specification as setrpt (though not vice-versa):

$$
\begin{array}{l}
\{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\} \\
\mathsf{lookuprpt}(\mathsf{j};\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{j}_0\} \\
\{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\}.
\end{array}
$$

The real difference between the procedures is revealed by their parameter lists: $\mathsf{lookuprpt}(\mathsf{j};\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{j}_0\}$ modifies only j, while $\mathsf{setrpt}(\mathsf{i};\mathsf{j},\mathsf{j}')\{\alpha,\mathsf{j}_0\}$ modifies only i.

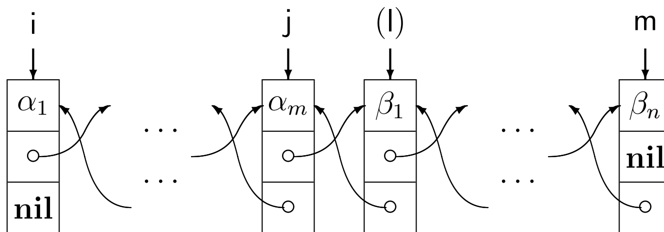
One can define a symmetric pair of procedures that act upon right-end segments. By similar proofs, one can establish

$$
\begin{array}{l}
\mathsf{lookuplpt}(\mathsf{i}';\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{i}'_0\} = \\
\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}')\} \\
\quad \mathbf{if}\ \mathsf{i} = \mathbf{nil}\ \mathbf{then}\ \mathsf{i}' := \mathsf{j}'\ \mathbf{else}\ \mathsf{i}' := [\mathsf{i}+2] \\
\quad \{\mathsf{dlseg}\ \alpha\ (\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}') \wedge \mathsf{i}' = \mathsf{i}'_0\}
\end{array}
$$

and

$$\begin{array}{l}\mathsf{setlpt}(\mathsf{j}';\mathsf{i},\mathsf{i}')\{\alpha,\mathsf{i}'_0\} = \\ \quad \{\mathsf{dlseg}\;\alpha\,(\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}')\} \\ \quad \mathbf{if}\;\mathsf{i} = \mathbf{nil}\;\mathbf{then}\;\mathsf{j}' := \mathsf{i}'\;\mathbf{else}\;[\mathsf{i}+2] := \mathsf{i}' \\ \quad \{\mathsf{dlseg}\;\alpha\,(\mathsf{i},\mathsf{i}',\mathbf{nil},\mathsf{j}')\}\end{array}$$

Now we can use these procedures in a program for inserting an element into a doubly-linked list after the element at address j:



$$
\begin{array}{l}
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\}\ \mathsf{lookuprpt}(\mathsf{j};\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{j}_0\}\ \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\wedge \mathsf{j}=\mathsf{j}_0\},\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\}\ \mathsf{setrpt}(\mathsf{i};\mathsf{j},\mathsf{j}')\{\alpha,\mathsf{j}_0\}\ \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\},\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}')\ \mathsf{lookuplpt}(\mathsf{i}';\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{i}'_0\}\ \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}')\wedge \mathsf{i}'=\mathsf{i}'_0\},\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}'_0,\mathbf{nil},\mathsf{j}')\ \mathsf{setlpt}(\mathsf{j}';\mathsf{i},\mathsf{i}')\{\alpha,\mathsf{i}'_0\}\ \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathbf{nil},\mathsf{j}')\}\ \vdash\\
\qquad \{\exists \mathsf{l}.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\qquad\quad \left.\begin{array}{l}
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\quad \left.\begin{array}{l}
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\}\\
\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})\{\alpha,\mathsf{j}_0\}\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\wedge \mathsf{l}=\mathsf{j}_0\}
\end{array}\right\}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})\\
\{(\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m}))\wedge \mathsf{l}=\mathsf{j}_0\}\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}
\end{array}\right\}\ \exists\,\mathsf{j}_0\\
\qquad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\qquad \mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{l},\mathsf{j})\ ;\\
\qquad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ *\ \mathsf{k}\mapsto \mathsf{a},\mathsf{l},\mathsf{j}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\qquad\quad \left.\begin{array}{l}
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\}\\
\mathsf{setrpt}(\mathsf{i};\mathsf{k},\mathsf{j})\{\alpha,\mathsf{l}\}\\
\{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\}
\end{array}\right\}\ *\ \mathsf{k}\mapsto \mathsf{a},\mathsf{l},\mathsf{j}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\\
\qquad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto \mathsf{a},\mathsf{l},\mathsf{j}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\qquad\quad \left.\begin{array}{l}
\{\mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\
\mathsf{setlpt}(\mathsf{m};\mathsf{l},\mathsf{k})\{\beta,\mathsf{j}\}\\
\{\mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})\}
\end{array}\right\}\ *\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto \mathsf{a},\mathsf{l},\mathsf{j}\\
\qquad \{\mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto \mathsf{a},\mathsf{l},\mathsf{j}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})\}\\
\qquad \{\mathsf{dlseg}\ \alpha{\cdot}\mathsf{a}{\cdot}\beta\,(\mathsf{i},\mathbf{nil},\mathbf{nil},\mathsf{m})\}
\end{array}
$$

Here there are calls of three of the four procedures we have defined, justified by the use of (GCALLan), the frame rule, and in the first case, the rule for existential quantification.

The annotations for these procedure calls are quite compressed. For example, those for the the call of lookuprpt embody the following argument:

Beginning with the hypothesis

$$\begin{aligned}&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\}\\&\mathsf{lookuprpt}(\mathsf{j};\mathsf{i},\mathsf{j}')\{\alpha,\mathsf{j}_0\}\\&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j}')\wedge\mathsf{j}=\mathsf{j}_0\},\end{aligned}$$

we use (GCALLan) to infer

$$\begin{aligned}&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\}\\&\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})\{\alpha,\mathsf{j}_0\}\\&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\wedge\mathsf{l}=\mathsf{j}_0\}.\end{aligned}$$

Next, after checking that the only variable l modified by $\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})$ does not occur free in $\mathsf{dlseg}\,\beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})$, we use the frame rule, the purity of $\mathsf{l}=\mathsf{j}_0$, and obvious properties of equality to obtain

$$\begin{aligned}&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})\}\\&\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})\{\alpha,\mathsf{j}_0\}\\&\{(\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\wedge\mathsf{l}=\mathsf{j}_0)\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})\}\\&\{(\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m}))\wedge\mathsf{l}=\mathsf{j}_0\}\\&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}.\end{aligned}$$

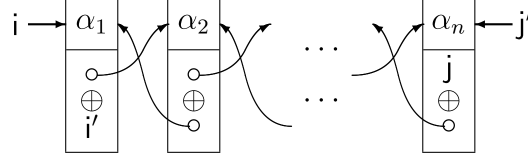
Finally, after checking that j_0 is not modified by $\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})$, we use the rule for existential quantification, as well as predicate-calculus rules for renaming and eliminating existential quantifiers, to obtain

$$\begin{aligned}&\{\exists\mathsf{l}.\;\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\&\{\exists\mathsf{j}_0.\;\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j}_0,\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{j}_0,\mathsf{j},\mathbf{nil},\mathsf{m})\}\\&\mathsf{lookuprpt}(\mathsf{l};\mathsf{i},\mathsf{j})\{\alpha,\mathsf{j}_0\}\\&\{\exists\mathsf{j}_0.\;\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\&\{\mathsf{dlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}.\end{aligned}$$

We can now illustrate how the difference in the variables modified by $\mathsf{lookuprpt}$ and setrpt affects the usage of these procedures. For example, if we replaced the call $\mathsf{setrpt}(\mathsf{i};\mathsf{k},\mathsf{j})\{\alpha,\mathsf{l}\}$ in the insertion program by the call $\mathsf{lookuprpt}(\mathsf{k};\mathsf{i},\mathsf{j})\{\alpha,\mathsf{l}\}$, which satisfies the same triple, the application of the frame rule would fail, since the call would modify the actual parameter k, which occurs free in $\mathsf{k}\mapsto\mathsf{a},\mathsf{l},\mathsf{j}\;*\;\mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})$.

4.9 Xor-Linked List Segments

An *xor-linked* list segment is a variation of the doubly-linked case where, in place of the forward and backward address fields, one stores in a single field the exclusive **or** of these values. We write $\mathsf{xlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}')$ to describe the situation



which is defined inductively by

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

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

where $\oplus$ denotes the exclusive **or** operation.

The basic properties are very similar to those of doubly-linked segments:

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

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

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

while the emptyness conditions are the same:

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

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

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

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

as is the definition of lists:

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

To illustrate programming with xor-linked list segments, we define a procedure analogous to setrpt in the previous section, which changes a left-end

segment so that the last address in the forward linkage is the value of the variable j. As before, the procedure body is a conditional command:

$$
\begin{array}{l}
\mathsf{xsetrpt}(\mathsf{i};\mathsf{j},\mathsf{j}',\mathsf{k})\{\alpha\} = \\
\quad \{\mathsf{xlseg}\;\alpha\;(\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\} \\
\quad \mathbf{if}\;\mathsf{j}' = \mathbf{nil}\;\mathbf{then} \\
\qquad \{\alpha = \epsilon \wedge \mathbf{emp} \wedge \mathsf{j}' = \mathbf{nil}\} \\
\qquad \mathsf{i} := \mathsf{k} \\
\qquad \{\alpha = \epsilon \wedge \mathbf{emp} \wedge \mathsf{j}' = \mathbf{nil} \wedge \quad \mathsf{i} = \mathsf{k}\} \\
\quad \mathbf{else} \\
\qquad \{\exists \alpha', \mathsf{b}, \mathsf{k}'.\;\; \alpha = \alpha' {\cdot} \mathsf{b} \wedge (\mathsf{xlseg}\;\alpha'\;(\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}') \;*\; \mathsf{j}' \mapsto \mathsf{b}, (\mathsf{j} \oplus \mathsf{k}'))\} \\
\qquad \left.\left.\begin{array}{l}
\{\mathsf{j}' \mapsto \mathsf{b}, (\mathsf{j} \oplus \mathsf{k}')\} \\
\mathbf{newvar}\;\mathsf{x}\;\mathbf{in} \\
\quad (\mathsf{x} := [\mathsf{j}' + 1]\;; \\
\quad \{\mathsf{x} = \mathsf{j} \oplus \mathsf{k}' \wedge \mathsf{j}' \mapsto \mathsf{b}, -\} \\
\quad [\mathsf{j}' + 1] := \mathsf{x} \oplus \mathsf{j} \oplus \mathsf{k} \\
\quad \{\mathsf{x} = \mathsf{j} \oplus \mathsf{k}' \wedge \mathsf{j}' \mapsto \mathsf{b}, \mathsf{x} \oplus \mathsf{j} \oplus \mathsf{k}\} \\
\quad \{\mathsf{j}' \mapsto \mathsf{b}, \mathsf{j} \oplus \mathsf{k}' \oplus \mathsf{j} \oplus \mathsf{k}\}) \\
\{\mathsf{j}' \mapsto \mathsf{b}, \mathsf{k} \oplus \mathsf{k}'\}
\end{array}\right\} * \left(\begin{array}{l}
\alpha = \alpha' {\cdot} \mathsf{b} \wedge \\
\mathsf{xlseg}\;\alpha'\;(\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}')
\end{array}\right)\right\} \exists \alpha', \mathsf{b}, \mathsf{k}' \\
\qquad \{\exists \alpha', \mathsf{b}, \mathsf{k}'.\;\; \alpha = \alpha' {\cdot} \mathsf{b} \wedge (\mathsf{xlseg}\;\alpha'\;(\mathsf{i},\mathbf{nil},\mathsf{j}',\mathsf{k}') \;*\; \mathsf{j}' \mapsto \mathsf{b}, (\mathsf{k} \oplus \mathsf{k}'))\} \\
\quad \{\mathsf{xlseg}\;\alpha\;(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j}')\}.
\end{array}
$$

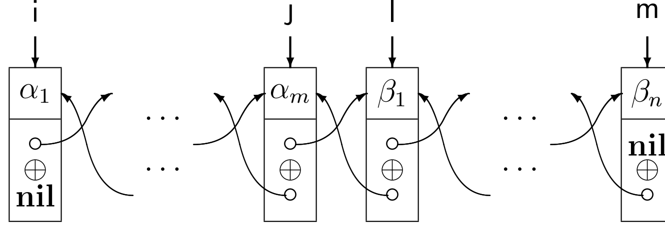
(Notice that what was the ghost variable j_0 in the specification of setrpt has become a variable j that is evaluated (but not modified) by $\mathsf{xsetrpt}$.

Similarly, one can define a procedure

$$
\begin{array}{l}
\mathsf{xsetlpt}(\mathsf{j}';\mathsf{i},\mathsf{i}',\mathsf{k})\{\alpha\} = \\
\quad \{\mathsf{xlseg}\;\alpha\;(\mathsf{i},\mathsf{i}',\mathbf{nil},\mathsf{j}')\} \\
\quad \mathbf{if}\;\mathsf{i} = \mathbf{nil}\;\mathbf{then}\;\mathsf{j}' := \mathsf{k}\;\mathbf{else} \\
\qquad \mathbf{newvar}\;\mathsf{x}\;\mathbf{in}\;(\mathsf{x} := [\mathsf{i} + 1]\;; [\mathsf{i} + 1] := \mathsf{x} \oplus \mathsf{i}' \oplus \mathsf{k}) \\
\quad \{\mathsf{xlseg}\;\alpha\;(\mathsf{i},\mathsf{k},\mathbf{nil},\mathsf{j}')\}
\end{array}
$$

that changes the last address in the backward linkage of a right-hand segment.

Then one can use these procedures in a program for inserting an element into an xor-linked list:



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

$$\mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{l}\oplus\mathsf{j})\ ;$$

$$\{\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\ *\ \mathsf{k}\mapsto\mathsf{a},(\mathsf{l}\oplus\mathsf{j})\ *\ \mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}$$

$$\left.\begin{array}{l}\{\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{l},\mathsf{j})\}\\ \mathsf{xsetrpt}(\mathsf{i};\mathsf{l},\mathsf{j},\mathsf{k})\{\alpha\}\\ \{\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\}\end{array}\right\}\ *\ \mathsf{k}\mapsto\mathsf{a},(\mathsf{l}\oplus\mathsf{j})\ *\ \mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})$$

$$\{\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto\mathsf{a},(\mathsf{l}\oplus\mathsf{j})\ *\ \mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}$$

$$\left.\begin{array}{l}\{\mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})\}\\ \mathsf{xsetlpt}(\mathsf{m};\mathsf{l},\mathsf{j},\mathsf{k})\{\beta\}\\ \{\mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})\}\end{array}\right\}\ *\ \mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto\mathsf{a},(\mathsf{l}\oplus\mathsf{j})$$

$$\{\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j})\ *\ \mathsf{k}\mapsto\mathsf{a},(\mathsf{l}\oplus\mathsf{j})\ *\ \mathsf{xlseg}\ \beta\,(\mathsf{l},\mathsf{k},\mathbf{nil},\mathsf{m})\}$$

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

Notice that, to know where to insert the new element, this program must be provided with the addresses of both the preceding and following elements. Typically, in working with xor-linked lists, one must work with pairs of addresses of adjacent elements. (As a minor consequence, the insertion program does not need to use a procedure analogous to lookuprpt.)

Finally, we note that xor-linked segments have the extraordinary property that, as can be proved by induction on α,

$$\mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \Leftrightarrow \mathsf{xlseg}\ \alpha^{\dagger}\,(\mathsf{j}',\mathsf{j},\mathsf{i}',\mathsf{i}),$$

and thus

$$\mathsf{xlist}\ \alpha\,(\mathsf{i},\mathsf{j}') \Leftrightarrow \mathsf{xlist}\ \alpha^{\dagger}\,(\mathsf{j}',\mathsf{i}).$$

Thus xor-linked segments can be reversed in constant time, and xor-linked lists can be reversed without changing the heap.

Exercise 1

Write an annotated specification for a program that will remove an element from a cyclic buffer and assign it to y. The program should satisfy

$$\{\exists\beta.\ (\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\mathsf{a}\cdot\alpha \wedge \mathsf{n} = \#\mathsf{a}{\cdot}\alpha + \#\beta \wedge \mathsf{m} > 0\}$$

$$\cdots$$

$$\{\exists\beta.\ (\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{lseg}\ \beta\,(\mathsf{j},\mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} = \#\alpha + \#\beta \wedge \mathsf{y} = \mathsf{a}\}.$$

Exercise 2

Prove that $\exists\alpha.\ \mathsf{ntlseg}\ \alpha\,(\mathsf{i},\mathsf{j})$ is a precise assertion.

Exercise 3

When

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

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

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 4

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,

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

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

Prove the assertion

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

Exercise 5

Write nonrecursive procedures for manipulating braced list segments, that satisfy the following hypotheses. In each case, give an annotated specification of the body that proves it is a correct implementation of the procedure. In a few cases, you may wish to use the procedures defined in previous cases.

1. A procedure for looking up the final pointer:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{lookuppt}(\mathsf{k}; \mathsf{i}, \mathsf{j})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0) \wedge \mathsf{k} = \mathsf{k}_0\}.$$

   (This procedure should not alter the heap.)

2. A procedure for setting the final pointer:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{setpt}(\mathsf{i}; \mathsf{j}, \mathsf{k})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k})\}.$$

   (This procedure should not allocate or deallocate heap storage.)

3. A procedure for appending an element on the left:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{appleft}(\mathsf{i}, \mathsf{j}; \mathsf{a})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

4. A procedure for deleting an element on the left:

   $$\{\mathsf{brlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{delleft}(\mathsf{i}, \mathsf{j};)\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

5. A procedure for appending an element on the right:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{appright}(\mathsf{i}, \mathsf{j}; \mathsf{a})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha{\cdot}\mathsf{a}\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

6. A procedure for concatenating two segments:

   $$\begin{array}{l}\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0) * \mathsf{brlseg}\ \beta\ (\mathsf{i}', \mathsf{j}', \mathsf{k}_0')\}\\ \mathsf{conc}(\mathsf{i}, \mathsf{j}; \mathsf{i}', \mathsf{j}')\{\alpha, \beta, \mathsf{k}_0, \mathsf{k}_0'\}\\ \{\mathsf{brlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0')\}.\end{array}$$

   (This procedure should not allocate or deallocate heap storage.)

Exercise 6

Rewrite the program in Section 4.8, for deleting an element from a doubly-linked list, to use the procedures setrpt and setlpt. Give an annotated specification. The program should satisfy:

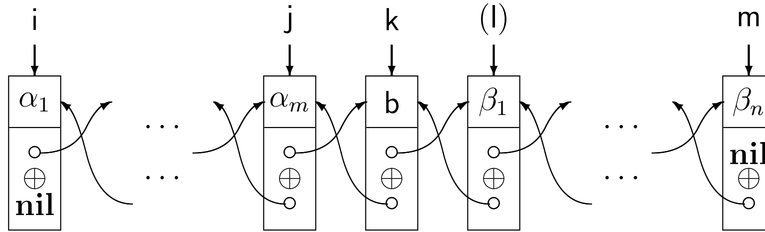
$$\{\exists \mathsf{j}, \mathsf{l}.\ \ \mathsf{dlseg}\ \alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$$

$$\cdots$$

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

Exercise 7

Give an annotated specification for a program to delete the element at k from an xor-linked list.



The program should use the procedures xsetrpt and xsetlpt, and should satisfy:

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

$$\cdots$$

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

Exercise 8

Prove, by structural induction on α, that

$$\mathsf{xlseg}\ \alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Leftrightarrow \mathsf{xlseg}\ \alpha^{\dagger}\,(\mathsf{j}', \mathsf{j}, \mathsf{i}', \mathsf{i}).$$