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.

These operations obey a variety of laws, including:

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

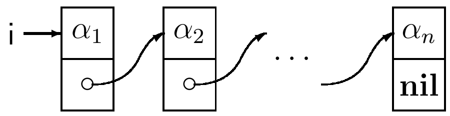
$$\epsilon^\dagger = \epsilon \qquad [a]^\dagger = [a] \qquad (\alpha{\cdot}\beta)^\dagger = \beta^\dagger{\cdot}\alpha^\dagger$$

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

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

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,



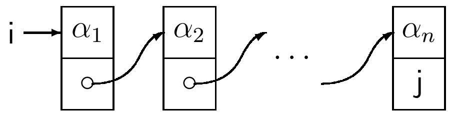
by structural induction on α:

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

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

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:

$$\begin{aligned} \mathsf{ntlseg}\ \epsilon\,(\mathsf{i}, \mathsf{j}) &\stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j} \\ \mathsf{ntlseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i}, \mathsf{k}) &\stackrel{\text{def}}{=} \mathsf{i} \neq \mathsf{k} \wedge \mathsf{i} \neq \mathsf{k} + 1 \wedge (\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{ntlseg}\ \alpha\,(\mathsf{j}, \mathsf{k})), \end{aligned}$$

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:

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

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:

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

or, more concisely:

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

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

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

(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:

$$
\begin{array}{ll}
\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\} & \\
\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\} & \\
\{\exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j} \ast (\mathsf{i} \mapsto \mathsf{a} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\} & \\
\mathsf{j} := [\mathsf{i}+1]\ ; & \text{(LKNOG)} \\
\{\mathsf{i}+1 \mapsto \mathsf{j} \ast (\mathsf{i} \mapsto \mathsf{a} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\} & \\
\{\mathsf{i} \mapsto \mathsf{a} \ast (\mathsf{i}+1 \mapsto \mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\} & \\
\mathbf{dispose}\ \mathsf{i}\ ; & \text{(DISG)} \\
\{\mathsf{i}+1 \mapsto \mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\} & \\
\mathbf{dispose}\ \mathsf{i}+1\ ; & \text{(DISG)} \\
\{\mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\} & \\
\mathsf{i} := \mathsf{j} & \text{(AS)} \\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{k})\}. &
\end{array}
$$

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:

$$
\begin{array}{ll}
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{a},\mathsf{k} \ast \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\} & \\
[\mathsf{j}+1] := \mathsf{l}\ ; & \text{(MUG)} \\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{a},\mathsf{l} \ast \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\} & \\
\mathbf{dispose}\ \mathsf{k}\ ; & \text{(DISG)} \\
\mathbf{dispose}\ \mathsf{k}+1 & \text{(DISG)} \\
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}) \ast \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\} & \\
\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\}. &
\end{array}
$$

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:

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

Note the use of (LKGE) for $j := [j + 1]$, with v, v', and v'' replaced by j, j', and j'', 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 12 (1) $\exists\alpha.\ \mathsf{list}\,\alpha\,\mathsf{i}$ *is a precise assertion.* (2) $\mathsf{list}\,\alpha\,\mathsf{i}$ *is a precise assertion.*

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

(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 replace s by $[\,\mathsf{i}\colon i\,]$, where $i = s(\mathsf{i})$. 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

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

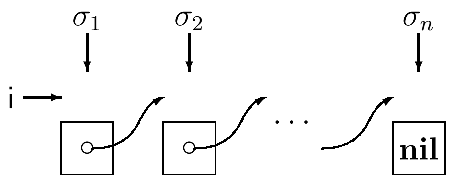
the induction hypothesis give $h_0' = h_1'$. It follows that $h_0 = h_1$.

(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:

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

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

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:

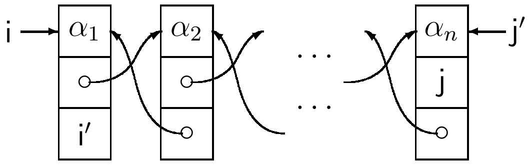
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i}\}$$
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i} \ast (\mathbf{emp} \wedge \mathbf{nil} = \mathbf{nil})\}$$
$$\mathsf{j} := \mathbf{nil}\ ; \qquad \text{(AS)}$$
$$\{\mathsf{listN}\ \sigma_0\ \mathsf{i} \ast (\mathbf{emp} \wedge \mathsf{j} = \mathbf{nil})\}$$
$$\{\mathsf{listN}\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

$$
\begin{array}{ll}
\quad \{\exists\sigma,\tau.\ (\mathsf{listN}\ \sigma\ \mathsf{k}\ *\ \mathsf{listN}\ \tau\ \mathsf{i}) \land \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau\} & \\
\quad \mathsf{j} := \mathsf{i}\ ; & \text{(AS)} \\
\quad \mathsf{i} := \mathsf{k} & \text{(ASan)} \\
\quad \{\exists\sigma,\tau.\ (\mathsf{listN}\ \sigma\ \mathsf{i}\ *\ \mathsf{listN}\ \tau\ \mathsf{j}) \land \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau\}\big) & \\
\{\exists\sigma,\tau.\ \mathsf{listN}\ \tau\ \mathsf{j} \land \sigma_0^\dagger = \sigma^\dagger{\cdot}\tau \land \sigma = \epsilon\} & \\
\{\mathsf{listN}\ \sigma_0^\dagger\ \mathsf{j}\} &
\end{array}
$$

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 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 — "(i,i') to (j,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:

$$
\begin{aligned}
\mathsf{dlseg}\ \epsilon\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') &\stackrel{\text{def}}{=} \mathbf{emp} \land \mathsf{i} = \mathsf{j} \land \mathsf{i}' = \mathsf{j}' \\
\mathsf{dlseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\stackrel{\text{def}}{=} \exists\mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j},\mathsf{i}'\ *\ \mathsf{dlseg}\ \alpha\,(\mathsf{j},\mathsf{i},\mathsf{k},\mathsf{k}').
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{dlseg}\ \mathsf{a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') &\Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \land \mathsf{i} = \mathsf{j}' \\
\mathsf{dlseg}\ \alpha{\cdot}\beta\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists\mathsf{j},\mathsf{j}'.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}')\ *\ \mathsf{dlseg}\ \beta\,(\mathsf{j},\mathsf{j}',\mathsf{k},\mathsf{k}') \\
\mathsf{dlseg}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists\mathsf{j}'.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k}',\mathsf{j}')\ *\ \mathsf{k}' \mapsto \mathsf{b},\mathsf{k},\mathsf{j}'.
\end{aligned}
$$

One can also define a doubly-linked list by

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

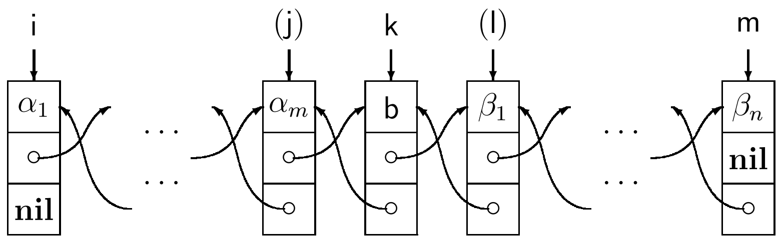
However, one cannot define dlist directly by structural induction, since no proper substructure of a doubly-linked list is a doubly-linked list.

Emptyness conditions can be formulated from either the forward or backward linkages:

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

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.



$$
\begin{array}{l}
\{\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})\}
\end{array}
$$

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.

4.6 Simple Procedures

Even simple programs for manipulating doubly-linked lists are sufficiently complex to provide illuminating examples of the use of small procedures. So we will digress from list structures to add a simple procedure mechanism to our programming language, and to formulate additional inference rules for the verification of such procedures.

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

- Parameters are variables and expressions, not commands or procedure names.
- Procedures are defined separately from the main program, rather than being declared within the program.
- 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.

(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.

Thus a *simple procedure definition* has the form

$$h(v_1, \cdots, v_m; v'_1, \cdots, v'_n) = c,$$

where $v'_1, \cdots, v'_n$ are the variables modified by the *body* c, and $v_1, \cdots, v_m$ are the other free variables of c. Then a procedure call (of a procedure defined as above) is a command of the form

$$h(e_1, \cdots, e_m; v''_1, \cdots, v''_n),$$

and $v''_1, \cdots, v''_n$ are said to be the variables modified by the procedure call.

Given a procedure definition of the above form, if one can prove the specification $\{p\}\ c\ \{q\}$, about the body of the definition, then one can use the specification $\{p\}\ h(v_1, \cdots .v_m; v'_1, \cdots, v'_n)\ \{q\}$ in proving a specification of a command that contains calls of h. This is captured by the inference rule

- Simple Procedures (SPROC)
  When $h(v_1, \cdots .v_m; v'_1, \cdots, v'_n) = c$,

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ h(v_1, \cdots .v_m; v'_1, \cdots, v'_n)\ \{q\}.}$$

From the conclusion of this rule, one can reason about other calls by using the rule for variable substitution (SUB). (The restrictions on substitutions in this rule prevent aliasing.)

If c contains recursive calls, then for partial correctness, one can assume $\{p\}\ h(v_1, \cdots .v_m; v'_1, \cdots, v'_n)\ \{q\}$ in proving $\{p\}\ c\ \{q\}$. Thus, we have:

- Simple Recursive Procedures (SRPROC)
  When $h(v_1, \cdots .v_m; v'_1, \cdots, v'_n) = c$,

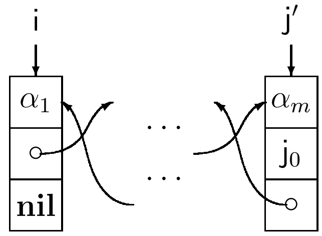
$$\frac{\begin{array}{c}\{p\}\ h(v_1, \cdots .v_m; v'_1, \cdots, v'_n)\ \{q\}\\ \vdots \\ \{p\}\ c\ \{q\}\end{array}}{\{p\}\ h(v_1, \cdots .v_m; v'_1, \cdots, v'_n)\ \{q\}.}$$

The notation here means that the consequence of the rule can be used as an assumption in proving the premiss. Thus one begins by assuming a "guess" of the specification of procedure calls, and then closes the circle by proving, under this assumption, that the procedure body satisfies this guess. As a consequence, the guess is valid, and can be used in reasoning about all calls of the procedure.

In both (SPROC) and (SRPROC), we assume that the procedure definition $h(v_1, \cdots .v_m; v'_1, \cdots, v'_n) = c$ is external to the program itself, and defines the meaning of calls of h throughout the program. Later, we will introduce *contexts* to deal with declarations within the program of identifiers such as procedure identifiers.

4.7 More about Doubly-Linked List Segments

We can now write some tiny procedures for examining and changing doubly-linked list segments. We begin with a conditional command that examines a left-end segment



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

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

If we define a procedure with the above command as its body:

$$\begin{array}{l}
\mathsf{lookuprpt}(\mathsf{i}, \mathsf{j}'; \mathsf{j}) = \\
\quad \mathbf{if}\ \mathsf{j}' = \mathbf{nil}\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else}\ \mathsf{j} := [\mathsf{j}' + 1]
\end{array}$$

then the procedure rule (SPROC) gives

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

Notice that the parameter list here makes it clear that the procedure call will only modify the variable j, and will not even evaluate j_0, since it does not appear in the parameter list. In fact, j_0 is a ghost variable of this specification. (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. We again begin with a conditional command:

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

so that the procedure

$$
\begin{array}{l}
\mathsf{setrpt}(\mathsf{j}, \mathsf{j}'; \mathsf{i}) = \\
\quad \mathbf{if}\ \mathsf{j}' = \mathbf{nil}\ \mathbf{then}\ \mathsf{i} := \mathsf{j}\ \mathbf{else}\ [\mathsf{j}' + 1] := \mathsf{j}
\end{array}
$$

satisfies

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

Here only the variable i can be modified. (It is curious that the previous procedure $\mathsf{lookuprpt}$ also satisfies the same specification, but since the variables it can modify are different, it must have a quite different effect.)

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

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

satisfies

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

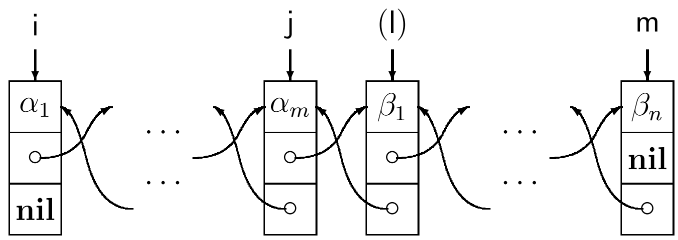
and the procedure

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

satisfies

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

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

Here there are three procedure calls, justified by the use of the rules for substitution, the frame rule, and in the first case, the rule for existential quantification. Notice that the actual procedure calls (obtained by applying the indicate substitution to the formal parameters) are

$$
\mathsf{lookuprpt}(\mathsf{i},\mathsf{j};\mathsf{l}) \qquad \mathsf{setrpt}(\mathsf{k},\mathsf{j};\mathsf{i}) \qquad \mathsf{setlpt}(\mathsf{l},\mathsf{k};\mathsf{m}).
$$

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

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

we extend the partial substitution

$$\mathsf{j}'\to\mathsf{j},\mathsf{j}\to\mathsf{l}$$

to a total substitution

$$\mathsf{j}'\to\mathsf{j},\mathsf{j}\to\mathsf{l},\mathsf{i}\to\mathsf{i},\mathsf{j}_0\to\mathsf{j}_0,\alpha\to\alpha,$$

on all variables free in the procedure specification, and check that the only variable (j) modified by $\mathsf{lookuprpt}(\mathsf{i},\mathsf{j}';\mathsf{j})$ is mapped by the total substitution into a variable (l) that does not occur free in any expression ($\mathsf{j},\mathsf{i},\mathsf{j}_0,\alpha$) obtained by applying the total substitution to any other free variable of the procedure specification ($\mathsf{j}',\mathsf{i},\mathsf{j}_0,\alpha$). Then we use Substitution Rule (SUB) to infer the result

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

of applying the total substitution to the procedure specification.

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

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

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

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

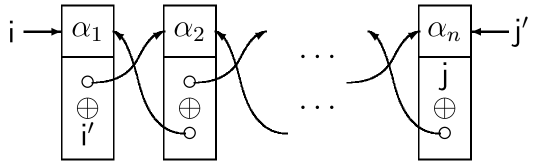
It is interesting to note that the Hoare triples specifying lookuprpt and setrpt have the same precondition, and the postcondition of lookuprpt implies that of setrpt, so that, by (WC), lookuprpt satisfies the same triple as setrpt (though not vice-versa):

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

Much of difference between the procedures is only revealed by their parameter lists. For example, if we replaced the call $\mathsf{setrpt}(\mathsf{k},\mathsf{j};\mathsf{i})$ in the insertion program by the call $\mathsf{lookuprpt}(\mathsf{i},\mathsf{j};\mathsf{k})$, which satisfies the same triple, the application of the frame rule will fail, since the call will modify the actual parameter k, which occurs free in $\mathsf{k} \mapsto \mathsf{a},\mathsf{l},\mathsf{j}\ *\ \mathsf{dlseg}\,\beta\,(\mathsf{l},\mathsf{j},\mathbf{nil},\mathsf{m})$.

4.8 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

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

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

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

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

$$\begin{aligned}
\mathsf{xlseg}\ \mathsf{a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') &\Leftrightarrow \mathsf{i} \mapsto \mathsf{a}, (\mathsf{j} \oplus \mathsf{i}') \wedge \mathsf{i} = \mathsf{j}' \\
\mathsf{xlseg}\ \alpha{\cdot}\beta\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists \mathsf{j},\mathsf{j}'.\ \mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ * \ \mathsf{xlseg}\ \beta\,(\mathsf{j},\mathsf{j}',\mathsf{k},\mathsf{k}') \\
\mathsf{xlseg}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists \mathsf{j}'.\ \mathsf{xlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k}',\mathsf{j}') \ * \ \mathsf{k}' \mapsto \mathsf{b}, (\mathsf{k} \oplus \mathsf{j}'),
\end{aligned}$$

while the emptyness conditions are the same:

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

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:

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

Thus the procedure

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

satisfies

$$
\begin{array}{l}
\{\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\}\\
\mathsf{xsetrpt}(\mathsf{j},\mathsf{j}',\mathsf{k};\mathsf{i})\\
\{\mathsf{xlseg}\,\alpha\,(\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j}')\}.
\end{array}
$$

(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 xsetrpt.

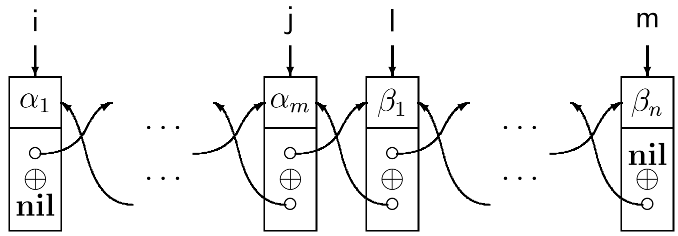
Similarly, one can define a procedure

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

that changes the last address in the backward linkage of a right-hand segment, as specified by:

$$\begin{array}{l}
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathsf{i}',\mathbf{nil},\mathsf{j}')\} \\
\mathsf{xsetlpt}(\mathsf{i},\mathsf{i}',\mathsf{k};\mathsf{j}') \\
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathsf{k},\mathbf{nil},\mathsf{j}')\}.
\end{array}$$

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



$$\begin{array}{l}
\{\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})\} \\
\quad \left.\begin{array}{l}
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{j},\mathsf{j}')\} \\
\mathsf{xsetrpt}(\mathsf{j},\mathsf{j}',\mathsf{k};\mathsf{i}) \\
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathbf{nil},\mathsf{k},\mathsf{j}')\}
\end{array}\right\} / \left[\begin{array}{c} \mathsf{j}\to\mathsf{l} \\ \mathsf{j}'\to\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})\} \\
\quad \left.\begin{array}{l}
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathsf{i}',\mathbf{nil},\mathsf{j}')\} \\
\mathsf{xsetlpt}(\mathsf{i},\mathsf{i}',\mathsf{k};\mathsf{j}') \\
\{\mathsf{xlseg}\ \alpha\ (\mathsf{i},\mathsf{k},\mathbf{nil},\mathsf{j}')\}
\end{array}\right\} / \left[\begin{array}{c} \mathsf{i}\to\mathsf{l} \\ \mathsf{i}'\to\mathsf{j} \\ \mathsf{j}'\to\mathsf{m} \\ \alpha\to\beta \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})\}
\end{array}$$

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 $\mathsf{lookuprpt}$.)

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.

4.9 Sorting by Merging — Using Lists

In what follows, we define some simple mathematical concepts about sequences, and then use them to prove a recursive program for sorting by merging.

I have not had time yet to add text to the bare bones of this material.

4.9.1 Images

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

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

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

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

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

4.9.2 Pointwise Extension of Relations

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:

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

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

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

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

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

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

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

The following abbreviations are convenient:

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

4.9.3 Ordering

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

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

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

4.9.4 Rearrangement

We say that a sequence β is a *rearrangement* of a sequence α, written $\beta \sim \alpha$, iff $\#\beta = \#\alpha$ and there is a permutation ϕ such that

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

Then

$$\alpha \sim \alpha \tag{4.14}$$

$$\alpha \sim \beta \;\Rightarrow\; \beta \sim \alpha \tag{4.15}$$

$$\alpha \sim \beta \wedge \beta \sim \gamma \;\Rightarrow\; \alpha \sim \gamma \tag{4.16}$$

$$\alpha \sim \alpha' \wedge \beta \sim \beta' \;\Rightarrow\; \alpha{\cdot}\beta \sim \alpha'{\cdot}\beta' \tag{4.17}$$

$$\alpha{\cdot}\beta \sim \beta{\cdot}\alpha \tag{4.18}$$

$$\alpha \sim \beta \;\Rightarrow\; \{\alpha\} = \{\beta\}. \tag{4.19}$$

4.9.5 Specifications for Sorting by Merging

We define the abbreviation

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

Then we will define procedures satisfying

$$\begin{array}{l}
\{\mathsf{lseg}\ \alpha\,(\mathsf{i}, \mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\}\\
\mathsf{mergesort}(\mathsf{n}; \mathsf{i}, \mathsf{j})\\
\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{array}$$

and

$$\begin{array}{l}
\{(\mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)\\
\quad *\ (\mathsf{lseg}\ \beta_2\,(\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\}\\
\mathsf{merge}(\mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2}; \mathsf{i})\\
\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta\}.
\end{array}$$

4.9.6 An Arithmetical Note

Suppose $\mathsf{n} \geq 2$. Then $2 \leq \mathsf{n} \leq 2 \times \mathsf{n} - 2$, 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 $\mathsf{n2} = \mathsf{n} - \mathsf{n1}$, then

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

4.9.7 A Proof for mergesort

$$
\begin{array}{l}
\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\}\\
\textbf{if}\ \mathsf{n} = 1\ \textbf{then}\\
\quad \{\mathsf{lseg}\ \alpha\,(\mathsf{i},-) \wedge \textbf{ord}\ \alpha \wedge \mathsf{i} \mapsto -,\mathsf{j}_0\}\\
\quad \mathsf{j} := [\mathsf{i}+1]\\
\quad \{\mathsf{lseg}\ \alpha\,(\mathsf{i},-) \wedge \textbf{ord}\ \alpha \wedge \mathsf{j} = \mathsf{j}_0\}\\
\textbf{else}\\
\big(\mathsf{n1} := \mathsf{n} \div 2\,;\, \mathsf{n2} := \mathsf{n} - \mathsf{n1}\,;\, \mathsf{i1} := \mathsf{i}\,;\\
\{\exists \alpha_1,\alpha_2,\mathsf{i}_2.\ ((\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) \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \wedge \alpha = \alpha_1{\cdot}\alpha_2\}\\
\mathsf{mergesort}(\mathsf{n1};\mathsf{i1},\mathsf{i2})\,;\\
\{\exists \alpha_1,\alpha_2,\beta_1.\ ((\mathsf{lseg}\ \beta_1\,(\mathsf{i1},-) \wedge \beta_1 \sim \alpha_1 \wedge \textbf{ord}\ \beta_1 \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)\\
\quad * \ (\mathsf{lseg}\ \alpha_2\,(\mathsf{i2},\mathsf{j}_0) \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \wedge \alpha = \alpha_1{\cdot}\alpha_2\}\\
\mathsf{mergesort}(\mathsf{n2};\mathsf{i2},\mathsf{j})\,;\\
\{\exists \alpha_1,\alpha_2,\beta_1,\beta_2.\ ((\mathsf{lseg}\ \beta_1\,(\mathsf{i1},-) \wedge \beta_1 \sim \alpha_1 \wedge \textbf{ord}\ \beta_1 \wedge \#\alpha_1 = \mathsf{n1}\\
\qquad \wedge\ \mathsf{n1} \geq 1)\\
\quad * \ (\mathsf{lseg}\ \beta_2\,(\mathsf{i2},-) \wedge \beta_2 \sim \alpha_2 \wedge \textbf{ord}\ \beta_2 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1))\\
\ \ \wedge\ \alpha = \alpha_1{\cdot}\alpha_2 \wedge \mathsf{j} = \mathsf{j}_0\}\\
\{\exists \beta_1,\beta_2.\ ((\mathsf{lseg}\ \beta_1\,(\mathsf{i1},-) \wedge \textbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)\\
* \ (\mathsf{lseg}\ \beta_2\,(\mathsf{i2},-) \wedge \textbf{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\}\\
\mathsf{merge}(\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2};\mathsf{i})\\
\{\exists \beta,\beta_1,\beta_2.\ \mathsf{lseg}\ \beta(\mathsf{i},-) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \textbf{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 \textbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{array}
$$

Thus we may define

$$
\begin{array}{l}
\mathsf{mergesort}(\mathsf{n};\mathsf{i},\mathsf{j}) =\\
\quad \textbf{if}\ \mathsf{n} = 1\ \textbf{then}\ \mathsf{j} := [\mathsf{i}+1]\ \textbf{else}\\
\qquad \big(\mathsf{n1} := \mathsf{n} \div 2\,;\, \mathsf{n2} := \mathsf{n} - \mathsf{n1}\,;\, \mathsf{i1} := \mathsf{i}\,;\\
\qquad \mathsf{mergesort}(\mathsf{n1};\mathsf{i1},\mathsf{i2})\,;\, \mathsf{mergesort}(\mathsf{n2};\mathsf{i2},\mathsf{j})\,;\\
\qquad \mathsf{merge}(\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2};\mathsf{i})\big).
\end{array}
$$

4.9.8 A Proof for merge

Suppose

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

Then

$$
\begin{array}{l}
\{(\mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \mathbf{ord}\,\beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)\\
\quad * \ (\mathsf{lseg}\ \beta_2\,(\mathsf{i2}, -) \wedge \mathbf{ord}\,\beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\}\\
\mathbf{newvar}\ \mathsf{a1}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{a2}\ \mathbf{in}\ \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ;\\
\quad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ (\mathsf{i} := \mathsf{i1}\ ;\ \mathsf{merge1}(\mathsf{n1} - 1, \mathsf{n2} - 1, \mathsf{i1}, \mathsf{i2}, \mathsf{a2}))\\
\quad \mathbf{else}\ (\mathsf{i} := \mathsf{i2}\ ;\ \mathsf{merge1}(\mathsf{n2} - 1, \mathsf{n1} - 1, \mathsf{i2}, \mathsf{i1}, \mathsf{a1}))\big)\\
\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}.
\end{array}
$$

Thus we may define

$$
\begin{array}{l}
\mathsf{merge}(\mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2}; \mathsf{i}) =\\
\quad \mathbf{newvar}\ \mathsf{a1}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{a2}\ \mathbf{in}\ \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ;\\
\qquad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ (\mathsf{i} := \mathsf{i1}\ ;\ \mathsf{merge1}(\mathsf{n1} - 1, \mathsf{n2} - 1, \mathsf{i1}, \mathsf{i2}, \mathsf{a2}))\\
\qquad \mathbf{else}\ (\mathsf{i} := \mathsf{i2}\ ;\ \mathsf{merge1}(\mathsf{n2} - 1, \mathsf{n1} - 1, \mathsf{i2}, \mathsf{i1}, \mathsf{a1}))\big)
\end{array}
$$

4.9.9 A Proof for merge1

$$
\begin{array}{l}
\{\exists\beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2.\\
\quad (\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}\\
\qquad * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -))\\
\quad \wedge \#\gamma_1 = \mathsf{n1} \wedge \#\gamma_2 = \mathsf{n2} \wedge \mathsf{a1} \leq \mathsf{a2} \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2\\
\quad \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\}\}\\
\mathbf{if}\ \mathsf{n1} = 0\ \mathbf{then}\ [\mathsf{i1} + 1] := \mathsf{i2}\\
\quad \{\exists\beta, \mathsf{a1}, \mathsf{j2}, \gamma_2.\\
\qquad (\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 \wedge \#\gamma_2 = \mathsf{n2} \wedge \mathsf{a1} \leq \mathsf{a2} \wedge \beta{\cdot}\mathsf{a1}{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2\\
\qquad \wedge\, \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}\\
\mathbf{else\ newvar}\ \mathsf{j1}\ \mathbf{in\ newvar}\ \mathsf{a3}\ \mathbf{in}\\
\quad \big(\mathsf{j1} := [\mathsf{i1} + 1]\,;\, \mathsf{a3} := [\mathsf{j1}]\,;\\
\quad \{\exists\beta, \mathsf{a1}, \mathsf{j3}, \gamma_3, \mathsf{j2}, \gamma_2.\\
\qquad (\mathsf{lseg}\ \beta\,(\mathsf{i}, \mathsf{i1}) \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{j1} \mapsto \mathsf{a3}, \mathsf{j3} \ * \ \mathsf{lseg}\ \gamma_3\,(\mathsf{j3}, -)\\
\qquad\quad * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -))\\
\quad \wedge \#\gamma_3 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} \wedge \mathsf{a1} \leq \mathsf{a2} \wedge \beta{\cdot}\mathsf{a1}{\cdot}\mathsf{a3}{\cdot}\gamma_3{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2\\
\quad \wedge\, \mathbf{ord}\,(\mathsf{a1}{\cdot}\mathsf{a3}{\cdot}\gamma_3) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,\beta\\
\quad \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\mathsf{a3}{\cdot}\gamma_3\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}\\
\quad \mathbf{if}\ \mathsf{a3} \leq \mathsf{a2}\ \mathbf{then}\ \mathsf{merge1}(\mathsf{n1} - 1, \mathsf{n2}, \mathsf{j1}, \mathsf{i2}, \mathsf{a2})\\
\quad \mathbf{else}\ ([\mathsf{i1} + 1] := \mathsf{i2}\,;\, \mathsf{merge1}(\mathsf{n2}, \mathsf{n1} - 1, \mathsf{i2}, \mathsf{j1}, \mathsf{a3}))\big)\\
\{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}
\end{array}
$$

Thus we may define

$$
\begin{array}{l}
\mathsf{merge1}(\mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2}, \mathsf{a2}) =\\
\quad \mathbf{if}\ \mathsf{n1} = 0\ \mathbf{then}\ [\mathsf{i1} + 1] := \mathsf{i2}\\
\quad \mathbf{else\ newvar}\ \mathsf{j1}\ \mathbf{in\ newvar}\ \mathsf{a3}\ \mathbf{in}\\
\qquad \big(\mathsf{j1} := [\mathsf{i1} + 1]\,;\, \mathsf{a3} := [\mathsf{j1}]\,;\\
\qquad \mathbf{if}\ \mathsf{a3} \leq \mathsf{a2}\ \mathbf{then}\ \mathsf{merge1}(\mathsf{n1} - 1, \mathsf{n2}, \mathsf{j1}, \mathsf{i2}, \mathsf{a2})\\
\qquad \mathbf{else}\ ([\mathsf{i1} + 1] := \mathsf{i2}\,;\, \mathsf{merge1}(\mathsf{n2}, \mathsf{n1} - 1, \mathsf{i2}, \mathsf{j1}, \mathsf{a3}))\big)
\end{array}
$$

Exercise 7

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 8

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

Exercise 9

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 10

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,

$$\mathsf{brlseg}\ \epsilon\ (\mathsf{i},\mathsf{j},\mathsf{k}) \stackrel{\mathrm{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{k} \wedge \mathsf{j} = \mathbf{nil}$$
$$\mathsf{brlseg}\ \alpha{\cdot}a\ (\mathsf{i},\mathsf{j},\mathsf{k}) \stackrel{\mathrm{def}}{=} \mathbf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}.$$

Prove the assertion

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

Exercise 11

Write nonrecursive procedures for manipulating braced list segments, that satisfy the following specifications. 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:

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

   (This procedure should not alter the heap.)

2. A procedure for setting the final pointer:

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

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

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

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

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

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

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

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

6. A procedure for concatenating two segments:

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

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

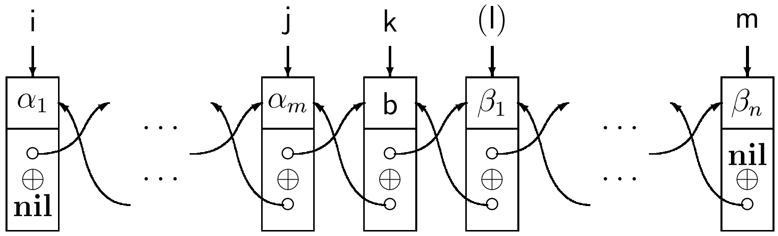
Exercise 12

Rewrite the program in Section 4.5, 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 13

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 14

Prove, by structural induction on α, that

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