Chapter 5

Trees and Dags

In this chapter, we consider various representations of abstract tree-like data. In general, such data are elements of (possibly many-sorted) initial or free algebras without laws. To illustrate the use of separation logic, however, it is simplest to limit our discussion to a particular form of abstract data.

For this purpose, as discussed in Section 1.7, we will use "S-expressions", which were the form of data used in early LISP [43]. The set S-exps of S-expressions is the least set such that

$$\begin{aligned} \tau \in \text{S-exps iff } & \tau \in \text{Atoms} \\ & \text{or } \tau = (\tau_0 \cdot \tau_1) \text{ where } \tau_0, \tau_1 \in \text{S-exps}. \end{aligned}$$

Here atoms are values that are not addresses, while $(\tau_0 \cdot \tau_1)$ is the LISP notation for an ordered pair. (Mathematically, S-expressions are the initial lawless algebra with an infinite number of constants and one binary operation.)

5.1 Trees

We use the word "tree" to describe a representation of S-expressions by two-field records, in which there is no sharing between the representation of subexpressions. More precisely, we define the predicate $\mathsf{tree}\,\tau\,(i)$ — read "i is (the root of) a tree representing the S-expression τ — by structural induction on τ:

$$\mathsf{tree}\,a\,(i) \text{ iff } \mathbf{emp} \wedge i = a \qquad \text{when } a \text{ is an atom}$$

$$\mathsf{tree}\,(\tau_0 \cdot \tau_1)\,(i) \text{ iff } \exists i_0, i_1.\ i \mapsto i_0, i_1 \ * \ \mathsf{tree}\,\tau_0\,(i_0) \ * \ \mathsf{tree}\,\tau_1\,(i_1).$$

One can show that the assertions $\mathsf{tree}\,\tau\,(i)$ and $\exists\tau.\ \mathsf{tree}\,\tau\,(i)$ are precise.

To illustrate the use of this definition, we define and verify a recursive procedure $\mathsf{copytree}(\mathsf{i};\mathsf{j})$ that nondestructively copies the tree i to j, i.e., that satisfies

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

The following is an annotated specification of the procedure body:

$$
\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\}\\
\quad \textbf{if } \mathsf{isatom}(\mathsf{i}) \textbf{ then}\\
\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\}\\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau) \ast (\mathbf{emp} \wedge \mathsf{i} = \tau))\}\\
\quad \mathsf{j} := \mathsf{i}\\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau) \ast (\mathbf{emp} \wedge \mathsf{j} = \tau))\}\\
\textbf{else}\\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{tree}\,(\tau_0 \cdot \tau_1)(\mathsf{i})\}\\
\quad \textbf{newvar } \mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1 \textbf{ in}\\
\qquad \big(\mathsf{i}_0 := [\mathsf{i}]\,;\, \mathsf{i}_1 := [\mathsf{i}+1]\,;\\
\qquad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast \mathsf{tree}\,\tau_0\,(\mathsf{i}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{i}_1))\}\\
\qquad \left.\left.\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\}\\
\mathsf{copytree}(\mathsf{i};\mathsf{j})\\
\{\mathsf{tree}\,\tau(\mathsf{i}) \ast \mathsf{tree}\,\tau(\mathsf{j})\}
\end{array}\right\} \Big/ \left[\begin{array}{c}
\tau \to \tau_0\\
\mathsf{i} \to \mathsf{i}_0\\
\mathsf{j} \to \mathsf{j}_0
\end{array}\right.\right\} \ast \left[\begin{array}{l}
\tau = (\tau_0 \cdot \tau_1) \wedge\\
(\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast\\
\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1))
\end{array}\right.\left.\vphantom{\begin{array}{l}a\\b\\c\end{array}}\right\} \exists \tau_0, \tau_1\\
\qquad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast\\
\qquad\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ast \mathsf{tree}\,\tau_0\,(\mathsf{j}_0))\}\\
\qquad \left.\left.\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\}\\
\mathsf{copytree}(\mathsf{i};\mathsf{j})\\
\{\mathsf{tree}\,\tau(\mathsf{i}) \ast \mathsf{tree}\,\tau(\mathsf{j})\}
\end{array}\right\} \Big/ \left[\begin{array}{c}
\tau \to \tau_1\\
\mathsf{i} \to \mathsf{i}_1\\
\mathsf{j} \to \mathsf{j}_1
\end{array}\right.\right\} \ast \left[\begin{array}{l}
\tau = (\tau_0 \cdot \tau_1) \wedge\\
(\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast\\
\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0) \ast\\
\quad \mathsf{tree}\,\tau_0\,(\mathsf{j}_0))
\end{array}\right.\left.\vphantom{\begin{array}{l}a\\b\\c\\d\end{array}}\right\} \exists \tau_0, \tau_1\\
\qquad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast\\
\qquad\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ast \mathsf{tree}\,\tau_0\,(\mathsf{j}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\}\\
\qquad \mathsf{j} := \mathbf{cons}(\mathsf{j}_0, \mathsf{j}_1)\\
\qquad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast \mathsf{j} \mapsto \mathsf{j}_0, \mathsf{j}_1 \ast\\
\qquad\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{i}_1) \ast \mathsf{tree}\,\tau_0\,(\mathsf{j}_0) \ast \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\}\\
\qquad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{tree}\,(\tau_0{\cdot}\tau_1)\,(\mathsf{i}) \ast \mathsf{tree}\,(\tau_0{\cdot}\tau_1)\,(\mathsf{j}))\}\big)\\
\{\mathsf{tree}\,\tau(\mathsf{i}) \ast \mathsf{tree}\,\tau(\mathsf{j})\}.
\end{array}
$$

Since this specification has the same pre- and post-condition as the assumed specification of the procedure call, we have closed the circle of recursive reasoning, and may define

$$\begin{array}{l}\mathsf{copytree}(\mathsf{i};\mathsf{j}) = \\ \quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\ \quad\quad \mathbf{newvar}\ \mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1\ \mathbf{in} \\ \quad\quad\quad \big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i}+1]\ ; \\ \quad\quad\quad \mathsf{copytree}(\mathsf{i}_0;\mathsf{j}_0)\ ;\ \mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\ \mathsf{j} := \mathbf{cons}(\mathsf{j}_0,\mathsf{j}_1)\big).\end{array} \tag{5.1}$$

5.2 Dags

We use the acronym "dag" (for "directed acyclic graph") to describe a representation for S-expressions by two-field records, in which sharing is permitted between the representation of subexpressions (but cycles are not permitted). More precisely, we define the predicate $\mathsf{dag}\,\tau\,(i)$ — read "i is (the root of) a dag representing the S-expression τ — by structural induction on τ:

$$\mathsf{dag}\,a\,(i) \text{ iff } i = a \qquad \text{when } a \text{ is an atom}$$

$$\mathsf{dag}\,(\tau_0 \cdot \tau_1)\,(i) \text{ iff } \exists i_0, i_1.\ \ i \mapsto i_0, i_1\ *\ (\mathsf{dag}\,\tau_0\,(i_0) \wedge \mathsf{dag}\,\tau_1\,(i_1)).$$

The essential change from the definition of tree is the use of ordinary rather than separating conjunction in the second line, which allows the dag's describing subtrees to share the same heap. However, if $\mathsf{dag}\,\tau\,(i)$ meant that the heap contained the dag representing τ and nothing else, then $\mathsf{dag}\,\tau_0\,(i_0) \wedge \mathsf{dag}\,\tau_1\,(i_1)$ would imply that τ_0 and τ_1 have the same representation (and are therefore the same S-expression). But we have dropped **emp** from the base case, so that $\mathsf{dag}\,\tau\,(i)$ only means that a dag representing τ occurs somewhere within the heap. In fact,

Proposition 13 *(1) $\mathsf{dag}\,\tau\,(i)$ and (2) $\exists\tau.\ \mathsf{dag}\,\tau\,(i)$ are intuitionistic assertions.*

PROOF (1) By induction on τ, using the fact that an assertion p is intuitionistic iff $p\ *\ \mathbf{true} \Rightarrow p$. If τ is an atom a, then

$$\begin{array}{l}\mathsf{dag}\,a\,(i)\ *\ \mathbf{true} \\ \quad \Rightarrow i = a\ *\ \mathbf{true} \\ \quad \Rightarrow i = a \\ \quad \Rightarrow \mathsf{dag}\,a\,(i),\end{array}$$

since $i = a$ is pure. Otherwise, $\tau = (\tau_0 \cdot \tau_1)$, and

$$\begin{aligned}
&\mathsf{dag}\,(\tau_0 \cdot \tau_1)\,(i) \;*\; \mathbf{true}\\
&\quad\Rightarrow \exists i_0, i_1.\;\; i \mapsto i_0, i_1 \;*\; (\mathsf{dag}\,\tau_0\,(i_0) \wedge \mathsf{dag}\,\tau_1\,(i_1)) \;*\; \mathbf{true}\\
&\quad\Rightarrow \exists i_0, i_1.\;\; i \mapsto i_0, i_1 \;*\\
&\qquad\quad ((\mathsf{dag}\,\tau_0\,(i_0) \;*\; \mathbf{true}) \wedge (\mathsf{dag}\,\tau_1\,(i_1) \;*\; \mathbf{true}))\\
&\quad\Rightarrow \exists i_0, i_1.\;\; i \mapsto i_0, i_1 \;*\; (\mathsf{dag}\,\tau_0\,(i_0) \wedge \mathsf{dag}\,\tau_1\,(i_1))\\
&\quad\Rightarrow \mathsf{dag}\,(\tau_0 \cdot \tau_1)\,(i),
\end{aligned}$$

by the induction hypothesis for τ_0 and τ_1.

(2) Again, using the fact that an assertion p is intuitionistic,

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

END OF PROOF

Moreover,

Proposition 14 (1) *For all* i, τ_0, τ_1, h_0, h_1, *if* $h_0 \cup h_1$ *is a function, and*

$$[\,\mathsf{i}\!: i \mid \tau\!: \tau_0\,], h_0 \models \mathsf{dag}\,\tau\,(\mathsf{i}) \quad \textit{and} \quad [\,\mathsf{i}\!: i \mid \tau\!: \tau_1\,], h_1 \models \mathsf{dag}\,\tau\,(\mathsf{i}), \tag{5.2}$$

then $\tau_0 = \tau_1$ *and*

$$[\,\mathsf{i}\!: i \mid \tau\!: \tau_0\,], h_0 \cap h_1 \models \mathsf{dag}\,\tau\,(\mathsf{i}).$$

(2) $\mathsf{dag}\,\tau\,\mathsf{i}$ *is a supported assertion.* (3) $\exists \tau.\; \mathsf{dag}\,\tau\,(\mathsf{i})$ *is a supported assertion.*

Proof We first note that: (a) When a is an atom, $[\,\mathsf{i}\!: i \mid \tau\!: a\,], h \models \mathsf{dag}\,\tau\,(\mathsf{i})$ iff $i = a$. (b) $[\,\mathsf{i}\!: i \mid \tau\!: (\tau_l \cdot \tau_r)\,], h \models \mathsf{dag}\,\tau\,(\mathsf{i})$ iff i is not an atom and there are i_l, i_r, and h' such that

$$\begin{gathered}
h = [\,i\!: i_l \mid i+1\!: i_r\,] \cdot h'\\
[\,\mathsf{i}\!: i_l \mid \tau\!: \tau_l\,], h' \models \mathsf{dag}\,\tau\,(\mathsf{i})\\
[\,\mathsf{i}\!: i_r \mid \tau\!: \tau_r\,], h' \models \mathsf{dag}\,\tau\,(\mathsf{i}).
\end{gathered}$$

(1) The proof is by structural induction on τ_0. For the base case, suppose τ_0 is an atom a. Then by (a), $i = a$. Moreover, if τ_1 were not an atom, then

by (b) we would have the contradiction that i is not an atom. Thus τ_1 must be atom a', and by (a), $i = a'$, so that $\tau_0 = \tau_1 = i = a = a'$. Then, also by (a), $[\,\mathsf{i}\colon i \mid \tau\colon \tau_0\,], h \models \mathsf{dag}\ \tau\ (\mathsf{i})$ holds for any h.

For the induction step suppose $\tau_0 = (\tau_{0l}\cdot\tau_{0r})$. Then by (b), i is not an atom, and there are i_{0l}, i_{0r}, and h'_0 such that

$$
\begin{gathered}
h_0 = [\,i\colon i_{0l} \mid i+1\colon i_{0r}\,] \cdot h'_0 \\
[\,\mathsf{i}\colon i_{0l} \mid \tau\colon \tau_{0l}\,], h'_0 \models \mathsf{dag}\ \tau\ (\mathsf{i}) \\
[\,\mathsf{i}\colon i_{0r} \mid \tau\colon \tau_{0r}\,], h'_0 \models \mathsf{dag}\ \tau\ (\mathsf{i}).
\end{gathered}
$$

Moreover, if τ_1 were an atom, then by (a) we would have the contradiction that i is an atom. Thus, τ_1 must have the form $(\tau_{1l}\cdot\tau_{1r})$, so that by (b) there are i_{1l}, i_{1r}, and h'_1 such that

$$
\begin{gathered}
h_1 = [\,i\colon i_{1l} \mid i+1\colon i_{1r}\,] \cdot h'_1 \\
[\,\mathsf{i}\colon i_{1l} \mid \tau\colon \tau_{1l}\,], h'_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}) \\
[\,\mathsf{i}\colon i_{1r} \mid \tau\colon \tau_{1r}\,], h'_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}).
\end{gathered}
$$

Since $h_0 \cup h_1$ is a function, h_0 and h_1 must map i and $i+1$ into the same values. Thus $[\,i\colon i_{0l} \mid i+1\colon i_{0r}\,] = [\,i\colon i_{1l} \mid i+1\colon i_{1r}\,]$, so that $i_{0l} = i_{1l}$ and $i_{0r} = i_{1r}$, and also,

$$
h_0 \cap h_1 = [\,i\colon i_{0l} \mid i+1\colon i_{0r}\,]\cdot(h'_0 \cap h'_1).
$$

Then, since

$$
[\,\mathsf{i}\colon i_{0l} \mid \tau\colon \tau_{0l}\,], h'_0 \models \mathsf{dag}\ \tau\ (\mathsf{i}) \text{ and } [\,\mathsf{i}\colon i_{1l} \mid \tau\colon \tau_{1l}\,], h'_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}),
$$

the induction hypothesis for τ_{0l} gives

$$
\tau_{0l} = \tau_{1l} \quad \text{and} \quad [\,\mathsf{i}\colon i_{0l} \mid \tau\colon \tau_{0l}\,], h'_0 \cap h'_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}),
$$

and the induction hypothesis for τ_{0r} gives

$$
\tau_{0r} = \tau_{1r} \quad \text{and} \quad [\,\mathsf{i}\colon i_{0r} \mid \tau\colon \tau_{0r}\,], h'_0 \cap h'_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}).
$$

Thus, (b) gives

$$
[\,\mathsf{i}\colon i \mid \tau\colon (\tau_{0l}\cdot\tau_{0r})\,], h_0 \cap h_1 \models \mathsf{dag}\ \tau\ (\mathsf{i}),
$$

which, with $\tau_0 = (\tau_{0l}{\cdot}\tau_{0r}) = (\tau_{1l}{\cdot}\tau_{1r}) = \tau_1$, establishes (1).

(2) Since τ and i are the only free variables in $\mathsf{dag}\,\tau\,(\mathsf{i})$, we can regard s and $[\,\mathsf{i}\colon i \mid \tau\colon\tau\,]$, where $i = s(\mathsf{i})$ and $\tau = s(\tau)$, as equivalent stores. Then (2) follows since $h_0 \cap h_1$ is a subset of both h_0 and h_1.

(3) Since i is the only free variable in $\exists\tau.\ \mathsf{dag}\,\tau\,(\mathsf{i})$, we can regard s and $[\,\mathsf{i}\colon i\,]$, where $i = s(\mathsf{i})$, as equivalent stores. Then we can use the semantic equation for the existential quantifier to show that there are S-expressions τ_0 and τ_1 such that (5.2) holds. Then (1) and the semantic equation for existentials shows that $[\,\mathsf{i}\colon i\,], h_0 \cap h_1 \models \mathsf{dag}\,\tau\,(\mathsf{i})$, and (3) follows since $h_0 \cap h_1$ is a subset of both h_0 and h_1. END OF PROOF

It follows that we can use the "precising" operation of Section 2.3.6,

$$\Pr p \stackrel{\text{def}}{=} p \wedge \neg(p \ * \ \neg\,\mathbf{emp}),$$

to convert $\mathsf{dag}\,\tau\,(\mathsf{i})$ and $\exists\tau.\ \mathsf{dag}\,\tau\,(\mathsf{i})$ into the precise assertions

$$\mathsf{dag}\,\tau\,(\mathsf{i}) \wedge \neg(\mathsf{dag}\,\tau\,(\mathsf{i}) \ * \ \neg\,\mathbf{emp})$$
$$(\exists\tau.\ \mathsf{dag}\,\tau\,(\mathsf{i})) \wedge \neg((\exists\tau.\ \mathsf{dag}\,\tau\,(\mathsf{i})) \ * \ \neg\,\mathbf{emp}),$$

each of which asserts that the heap contains the dag at i and nothing else.

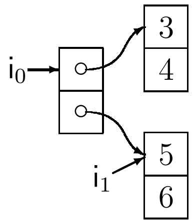
Now consider the procedure $\mathsf{copytree}(\mathsf{i};\mathsf{j})$. It creates a new tree rooted at j, but it never modifies the structure at i, nor does it compare any pointers for equality (or any other relation). So we would expect it to be impervious to sharing in the structure being copied, and thus to satisfy

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

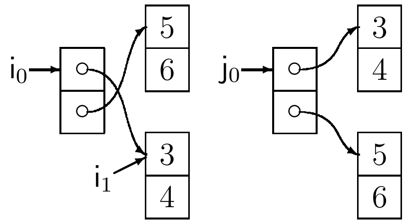
In fact, this specification is satisfied, but if we try to mimic the proof in the previous section, we encounter a problem. If we take the above specification as the hypothesis about recursive calls of $\mathsf{copytree}$, then we will be unable to prove the necessary property of the first recursive call:

$$\begin{array}{l}\{\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ * \ (\mathsf{dag}\,\tau_0(\mathsf{i}_0) \wedge \mathsf{dag}\,\tau_1(\mathsf{i}_1))\}\\ \mathsf{copytree}(\mathsf{i}_0;\mathsf{j}_0)\\ \{\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ * \ (\mathsf{dag}\,\tau_0(\mathsf{i}_0) \wedge \mathsf{dag}\,\tau_1(\mathsf{i}_1)) \ * \ \mathsf{tree}\,\tau_0(\mathsf{j}_0)\}.\end{array}$$

(Here, we ignore pure assertions in the pre- and postconditions that are irrelevant to this argument.) But the hypothesis is not strong enough to imply this. For example, suppose $\tau_0 = ((3 \cdot 4) \cdot (5 \cdot 6))$ and $\tau_1 = (5 \cdot 6)$. Then, even though it satisfies the hypothesis, the call $\mathsf{copytree}(\mathsf{i}_0; \mathsf{j}_0)$ might change the state from



into



where $\mathsf{dag}\, \tau_1\, (\mathsf{i}_1)$ is false.

To circumvent this problem, we must strengthen the specification of copytree to specify that a call of the procedure does not change the heap that exists when it begins execution. There are (at least) three possible approaches:

1. Introduce ghost variables denoting heaps. Suppose h_0 is such a variable, and the assertion $\mathbf{this}(\mathsf{h}_0)$ is true just in the heap that is the value of h_0. Then we could specify

$$\{\mathbf{this}(\mathsf{h}_0) \land \mathsf{dag}\, \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathbf{this}(\mathsf{h}_0)\ *\ \mathsf{tree}\, \tau(\mathsf{j})\}.$$

2. Introduce ghost variables denoting assertions (or semantically, denoting properties of heaps). Then we could use an assertion variable p to specify that every property of the initial heap is also a property of the final subheap excluding the newly created copy:

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

3. As discussed later, introduce fractional permissions, or some other form of assertion that part of the heap is read-only or *passive*. Then one could define an assertion $\mathsf{passdag}\, \tau(\mathsf{i})$ describing a read-only heap containing a dag, and use it to specify that the initial heap is at no time altered by copytree:

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

(The stipulation "at no time" will become important when we consider concurrent computation.)

Here we will explore the second approach.

5.3 Assertion Variables

To extend our language to encompass assertion variables, we introduce this new type of variable as an additional form of assertion. Then we extend the concept of state to include an *assertion store* mapping assertion variables into properties of heaps:

$$\mathrm{AStores}_A = A \to (\mathrm{Heaps} \to \mathbf{B})$$
$$\mathrm{States}_{AV} = \mathrm{AStores}_A \times \mathrm{Stores}_V \times \mathrm{Heaps},$$

where A denotes a finite set of *assertion variables*.

Since assertion variables are always ghost variables, assertion stores have no effect on the execution of commands, but they affect the meaning of assertions. Thus, when as is an assertion store, s is a store, h is a heap, and p is an assertion whose free assertion variables all belong to the domain of as and whose free variables all belong to the domain of s, we write

$$as, s, h \models p$$

(instead of $s, h \models p$) to indicate that the state as, s, h *satisfies* p.

The formulas in Section 2.1 defining the relation of satisfaction are all generalized by changing the left side to $as, s, h \models$ and leaving the rest of the formula unchanged. Then we add a formula for the case where an assertion variable a is used as an assertion:

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

This generalization leads to a generalization of the substitution law, in which substitutions map assertion variables into assertions, as well as ordinary variables into expressions. We write $\mathrm{AV}(p)$ for the assertion variables occurring free in p. (Since, we have not (yet) introduced any binders of assertion variables, all of their occurrences are free.)

Proposition 15 *(Generalized Partial Substitution Law for Assertions) Suppose p is an assertion, and let δ abbreviate the substitution*

$$a_1 \to p_1, \ldots, a_m \to p_m, v_1 \to e_1, \ldots, v_n \to e_n,$$

Then let s be a store such that

$$(\mathrm{FV}(p) - \{v_1, \ldots, v_n\}) \cup \mathrm{FV}(p_1, \ldots, p_m, e_1, \ldots, e_n) \subseteq \mathrm{dom}\, s,$$

and let as be an assertion store such that

$$(\mathrm{AV}(p) - \{a_1, \ldots, a_m\}) \cup \mathrm{AV}(p_1, \ldots, p_m) \subseteq \mathrm{dom}\ as,$$

and let

$$\begin{aligned} \widehat{s} &= [\, s \mid v_1\colon [\![e_1]\!]_{\mathrm{exp}} s \mid \ldots \mid v_n\colon [\![e_n]\!]_{\mathrm{exp}} s \,] \\ \widehat{as} &= [\, as \mid a_1\colon \lambda h.\ (as, s, h \models p_1) \mid \ldots \mid a_m\colon \lambda h.\ (as, s, h \models p_m) \,] \end{aligned}$$

Then

$$as, s, h \models (p/\delta) \text{ iff } \widehat{as}, \widehat{s}, h \models p.$$

The definition of Hoare triples remains unchanged, except that one uses — and quantifies over — the new enriched notion of states. Command execution neither depends upon nor alters the new assertion-store component of these states.

The inference rules for substitution (in both the setting of explicit proofs and of annotated specifications) must also be extended:

- Substitution (SUB)

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

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

- Substitution (SUBan)

$$\frac{p' \Rightarrow (p/\delta) \qquad \begin{matrix}\{p\} \\ \hat{c} \\ \{q\}\end{matrix} \gg c \qquad (q/\delta) \Rightarrow q'}{\begin{matrix}\{p'\} \\ \left.\begin{matrix}\{p\} \\ \hat{c} \\ \{q\}\end{matrix}\right\} /\delta \\ \{q'\}\end{matrix} \gg c/\delta,}$$

where δ is the substitution $a_1 \to p_1, \ldots, a_m \to p_m, v_1 \to e_1, \ldots, v_n \to e_n$, $a_1, \ldots, a_m$ are the assertion variables occurring in p or q, $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j or in any p_j.

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

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

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

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

5.4 Copying Dags to Trees

Now we can prove that the procedure copytree defined by (5.1) satisfies

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

In this specification, we can take p to be $\mathsf{dag}\ \tau(\mathsf{i})$, to obtain the weaker specification

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

but, as we have seen, the latter specification is too weak to serve as a recursion hypothesis.

The following is an annotated specification proving that the body of copytree meets the stronger assumed specification. (The assertion marked $(*)$ is obtain by using $\textbf{true}\ *\ \mathsf{dag}\ \tau\ (\mathsf{i}) \Rightarrow \mathsf{dag}\ \tau\ (\mathsf{i})$, which holds since $\mathsf{dag}\ \tau\ (\mathsf{i})$ is intuitionistic.)

$\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}$

$\mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}$

$\quad\{\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = \mathsf{i}\}$

$\quad\{\mathsf{p} \ast (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp})\}$

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

$\quad\{\mathsf{p} \ast (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{j} \wedge \mathbf{emp})\}$

$\mathbf{else}$

$\quad\{\exists\tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\ (\tau_0 \cdot \tau_1)(\mathsf{i})\}$

$\quad\mathbf{newvar}\ \mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1\ \mathbf{in}$

$\qquad\big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i}+1]\ ;$

$\qquad\{\exists\tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$

$\qquad\quad \mathsf{p} \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 \ast (\mathsf{dag}\ \tau_0\ (\mathsf{i}_0) \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)))\}$

$\qquad\{\exists\tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$

$\qquad\quad \mathsf{p} \wedge (\mathbf{true} \ast (\mathsf{dag}\ \tau_0\ (\mathsf{i}_0) \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)))\}$

$\qquad\{\exists\tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$

$\qquad\quad \mathsf{p} \wedge ((\mathbf{true} \ast \mathsf{dag}\ \tau_0\ (\mathsf{i}_0)) \wedge (\mathbf{true} \ast \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)))\}$

$\qquad\{\exists\tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1) \wedge \mathsf{dag}\ \tau_0\ (\mathsf{i}_0)\}$ $(\ast)$

$$\left.\begin{array}{l}\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\\ \{\mathsf{p} \ast \mathsf{tree}\ \tau(\mathsf{j})\}\end{array}\right\} \Big/ \left[\begin{array}{l}\mathsf{p} \rightarrow \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}\\ \qquad\qquad \wedge\ \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)\\ \tau \rightarrow \tau_0\\ \mathsf{i} \rightarrow \mathsf{i}_0\\ \mathsf{j} \rightarrow \mathsf{j}_0\end{array}\right] \exists\tau_0, \tau_1$$

$\qquad\{\exists\tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)) \ast \mathsf{tree}\ \tau_0\ (\mathsf{j}_0)\}$

$$\left.\left.\begin{array}{l}\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\\ \{\mathsf{p} \ast \mathsf{tree}\ \tau(\mathsf{j})\}\end{array}\right\} \Big/ \left[\begin{array}{l}\mathsf{p} \rightarrow \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}\\ \tau \rightarrow \tau_1\\ \mathsf{i} \rightarrow \mathsf{i}_1\\ \mathsf{j} \rightarrow \mathsf{j}_1\end{array}\right] \ast \mathsf{tree}\ \tau_0\ (\mathsf{j}_0)\right\} \exists\tau_0, \tau_1$$

$\qquad\{\exists\tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}) \ast \mathsf{tree}\ \tau_0\ (\mathsf{j}_0) \ast \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\}$

$\qquad\mathsf{j} := \mathbf{cons}(\mathsf{j}_0, \mathsf{j}_1)$

$\qquad\{\exists\tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}) \ast$

$\qquad\quad \mathsf{j} \mapsto \mathsf{j}_0, \mathsf{j}_1 \ast \mathsf{tree}\ \tau_0\ (\mathsf{j}_0) \ast \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\}$

$\qquad\{\exists\tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}) \ast \mathsf{tree}\ (\tau_0 \cdot \tau_1)\ (\mathsf{j})\}\big)$

$\{\mathsf{p} \ast \mathsf{tree}\ \tau(\mathsf{j})\}$

5.5 Substitution in S-expressions

To obtain further programs dealing with trees and dags, we consider the substitution of S-expressions for atoms in S-expressions. We write $\tau/\mathsf{a} \to \tau'$ for the result of substituting τ' for the atom a in τ, which is defined by structural induction on τ:

$$\begin{aligned} a/a \to \tau' &= \tau' \\ b/a \to \tau' &= b \qquad \text{when } b \in \text{Atoms} - \{a\} \\ (\tau_0 \cdot \tau_1)/a \to \tau' &= ((\tau_0/a \to \tau') \cdot (\tau_1/a \to \tau')). \end{aligned}$$

Although we are using the same notation, this operation is different from the substitution for variables in expressions, assertions, or commands. In particular, there is no binding or renaming.

We will define a procedure that, given a tree representing τ and a dag representing τ', produces a tree representing $\tau/\mathsf{a} \to \tau'$, i.e.,

$$\{\mathsf{tree}\,\tau\,(\mathsf{i}) \ast (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\}\mathsf{subst1}(\mathsf{a},\mathsf{j}\,;\mathsf{i})\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i}) \ast \mathsf{p}\}.$$

Then the following is an annotated specification for the procedure body:

$$\begin{array}{l}
\{\mathsf{tree}\,\tau\,(\mathsf{i}) \ast (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\textbf{if } \mathsf{isatom}(\mathsf{i}) \textbf{ then} \\
\quad \{(\mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp}) \ast (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\quad \textbf{if } \mathsf{i} = \mathsf{a} \textbf{ then} \\
\quad\quad \{(\mathsf{isatom}(\tau) \wedge \tau = \mathsf{a} \wedge \mathbf{emp}) \ast (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\quad\quad \{((\tau/\mathsf{a} \to \tau') = \tau' \wedge \mathbf{emp}) \ast (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\quad\quad \left.\begin{array}{l} \{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\} \\ \mathsf{copytree}(\mathsf{i};\mathsf{j}) \\ \{\mathsf{p} \ast \mathsf{tree}\,\tau(\mathsf{j})\} \end{array}\right\} \Big/ \left[\begin{array}{c} \tau \to \tau' \\ \mathsf{i} \to \mathsf{j} \\ \mathsf{j} \to \mathsf{i} \end{array}\right. \left.\vphantom{\begin{array}{c} \tau \\ \mathsf{i} \\ \mathsf{j} \end{array}}\right\} \ast ((\tau/\mathsf{a} \to \tau') = \tau' \wedge \mathbf{emp}) \\
\quad\quad \{((\tau/\mathsf{a} \to \tau') = \tau' \wedge \mathbf{emp}) \ast \mathsf{tree}\,\tau'\,(\mathsf{i}) \ast \mathsf{p}\} \\
\quad \textbf{else} \\
\quad\quad \{(\mathsf{isatom}(\tau) \wedge \tau \neq \mathsf{a} \wedge \tau = \mathsf{i} \wedge \mathbf{emp}) \ast \mathsf{p}\} \\
\quad\quad \{((\tau/\mathsf{a} \to \tau') = \tau \wedge \mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp}) \ast \mathsf{p}\} \\
\quad\quad \textbf{skip}
\end{array}$$

$$
\begin{array}{l}
\textbf{else newvar } \mathsf{i}_0, \mathsf{i}_1 \textbf{ in } \big(\\
\quad \{\exists \tau_0, \tau_1, \mathsf{i}_0, \mathsf{i}_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ * \\
\qquad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \mathsf{i}_0 := [\mathsf{i}]\,;\, \mathsf{i}_1 := [\mathsf{i}+1]\,; \\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto -, -\ * \\
\qquad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \left.\begin{array}{l}
\{\mathsf{tree}\,\tau\,(\mathsf{i})\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\mathsf{subst1}(\mathsf{a}, \mathsf{j}\,;\, \mathsf{i}) \\
\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i})\ *\ \mathsf{p}\}
\end{array}\right\} \Big/ \left.\left[\begin{array}{l}
\mathsf{p} \to \tau = (\tau_0 \cdot \tau_1) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ * \\
\qquad\quad (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))) \\
\tau \to \tau_0 \\
\ \mathsf{i} \to \mathsf{i}_0
\end{array}\right.\right\} \exists \tau_0, \tau_1 \\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto -, -\ * \\
\qquad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,(\tau_0/\mathsf{a} \to \tau')\,(\mathsf{i}_0)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \left.\begin{array}{l}
\{\mathsf{tree}\,\tau\,(\mathsf{i})\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\mathsf{subst1}(\mathsf{a}, \mathsf{j}\,;\, \mathsf{i}) \\
\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i})\ *\ \mathsf{p}\}
\end{array}\right\} \Big/ \left.\left[\begin{array}{l}
\mathsf{p} \to \tau = (\tau_0 \cdot \tau_1) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ * \\
\qquad\quad \mathsf{tree}\,(\tau_0/\mathsf{a} \to \tau')\,(\mathsf{i}_0)\ *\ \mathsf{p}) \\
\tau \to \tau_1 \\
\ \mathsf{i} \to \mathsf{i}_1
\end{array}\right.\right\} \exists \tau_0, \tau_1 \\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto -, -\ * \\
\qquad \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,(\tau_0/\mathsf{a} \to \tau')\,(\mathsf{i}_0)\ *\ \mathsf{p})\} \\
\quad [\mathsf{i}] := \mathsf{i}_0\,;\, [\mathsf{i}+1] := \mathsf{i}_1 \\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ * \\
\qquad \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,(\tau_0/\mathsf{a} \to \tau')\,(\mathsf{i}_0)\ *\ \mathsf{p})\} \\
\quad \{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{tree}\,((\tau_0/\mathsf{a} \to \tau') \cdot (\tau_1/\mathsf{a} \to \tau'))\,(\mathsf{i})\ *\ \mathsf{p})\}\big) \\
\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i})\ *\ \mathsf{p}\}
\end{array}
$$

Since the pre- and postcondition in this annotated specification match those in the assumption about procedure calls, we have shown that the assumption is satisfied by the procedure

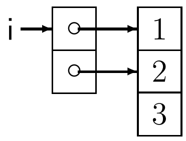
$$\begin{aligned}&\mathsf{subst1}(\mathsf{a},\mathsf{j}\,;\mathsf{i}) = \\ &\quad \textbf{if } \mathsf{isatom}(\mathsf{i}) \textbf{ then if } \mathsf{i} = \mathsf{a} \textbf{ then } \mathsf{copytree}(\mathsf{j}\,;\mathsf{i}) \textbf{ else skip} \\ &\quad \textbf{else newvar } \mathsf{i}_0, \mathsf{i}_1 \textbf{ in } (\mathsf{i}_0 := [\mathsf{i}]\,; \mathsf{i}_1 := [\mathsf{i}+1]\,; \\ &\qquad \mathsf{subst1}(\mathsf{a},\mathsf{j}\,;\mathsf{i}_0)\,; \mathsf{subst1}(\mathsf{a},\mathsf{j}\,;\mathsf{i}_1)\,; [\mathsf{i}] := \mathsf{i}_0\,; [\mathsf{i}+1] := \mathsf{i}_1).\end{aligned}$$

5.6 Skewed Sharing

Unfortunately, the definition we have given for dag permits a phenomenon called *skewed sharing*, where two records can overlap without being identical. For example,

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

holds when



Skewed sharing is not a problem for the algorithms we have seen so far, which only examine dags while ignoring their sharing structure. But it causes difficulties with algorithms that modify dags or depend upon the sharing structure.

A straightforward solution that controls skewed sharing is to add to each state a mapping ϕ from the domain of the heap to natural numbers, called the *field count*. Then, when $v := \mathbf{cons}(e_1, \ldots, e_n)$ creates a n-element record, the field count of the first field is set to n, while the field count of the remaining fields are set to zero. Thus if a is the address of the first field (i.e., the value assigned to the variable v), the field count is extended so that

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

The field count is an example of a *heap auxiliary*, i.e. an attribute of the heap that can be described by assertions but plays no role in the execution of commands. (Later, when we introduce *permissions*, we will see another example of a heap auxiliary.)

To describe the field count, we introduce a new assertion of the form $e \overset{[\hat{e}]}{\mapsto} e'$, with the meaning

$$
\begin{aligned}
&s, h, \phi \models e \overset{[\hat{e}]}{\mapsto} e' \text{ iff} \\
&\qquad \operatorname{dom} h = \{[\![e]\!]_{\text{exp}} s\} \text{ and } h([\![e]\!]_{\text{exp}} s) = [\![e']\!]_{\text{exp}} s \text{ and } \phi([\![e]\!]_{\text{exp}} s) = [\![\hat{e}]\!]_{\text{exp}} s.
\end{aligned}
$$

We also introduce the following abbreviations:

$$e \overset{[\hat{e}]}{\mapsto} - \overset{\text{def}}{=} \exists x'.\ e \overset{[\hat{e}]}{\mapsto} x' \quad \text{where } x' \text{ not free in } e$$

$$e \overset{[\hat{e}]}{\hookrightarrow} e' \overset{\text{def}}{=} e \overset{[\hat{e}]}{\mapsto} e' * \mathbf{true}$$

$$e \overset{!}{\mapsto} e_1, \dots, e_n \overset{\text{def}}{=} e \overset{[n]}{\mapsto} e_1 * e+1 \overset{[0]}{\mapsto} e_2 * \cdots * e+n-1 \overset{[0]}{\mapsto} e_n$$

$$
\begin{aligned}
e \overset{!}{\hookrightarrow} e_1, \dots, e_n &\overset{\text{def}}{=} e \overset{[n]}{\hookrightarrow} e_1 * e+1 \overset{[0]}{\hookrightarrow} e_2 * \cdots * e+n-1 \overset{[0]}{\hookrightarrow} e_n \\
&\text{iff } e \overset{!}{\mapsto} e_1, \dots, e_n * \mathbf{true}.
\end{aligned}
$$

Axiom schema for reasoning about these new assertions include:

$$e \overset{[\mathsf{n}]}{\mapsto} e' \Rightarrow e \mapsto e'$$

$$e \overset{[\mathsf{m}]}{\hookrightarrow} - \wedge e \overset{[\mathsf{n}]}{\hookrightarrow} - \Rightarrow \mathsf{m} = \mathsf{n}$$

$$2 \le \mathsf{k} \le \mathsf{n} \wedge e \overset{[\mathsf{n}]}{\hookrightarrow} - \wedge e + \mathsf{k} - 1 \hookrightarrow - \Rightarrow e + \mathsf{k} - 1 \overset{[0]}{\hookrightarrow} -$$

$$
\begin{aligned}
&e \overset{!}{\hookrightarrow} e_1, \dots, e_m \wedge e' \overset{!}{\hookrightarrow} e'_1, \dots, e'_n \wedge e \neq e' \Rightarrow \\
&\qquad\qquad e \overset{!}{\mapsto} e_1, \dots, e_m * e' \overset{!}{\mapsto} e'_1, \dots, e'_n * \mathbf{true}.
\end{aligned}
$$

(The last of these axiom schemas makes it clear that skewed sharing has been prohibited.)

The inference rules for allocation, mutation, and lookup remain sound, but they are supplemented with additional rules for these commands that take field counts into account. We list only the simplest forms of these rules:

- Allocation: the local nonoverwriting form (FCCONSNOL)

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

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

- Mutation: the local form (FCMUL)

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

- Lookup: the local nonoverwriting form (FCLKNOL)

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

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

The operation of deallocation, however, requires more serious change. If one can deallocate single fields, the use of field counts can be disrupted by deallocating a part of record, since the storage allocator may reallocate the same address as the head of a new record. For example, the command

$$\mathsf{j} := \mathbf{cons}(1, 2)\ ;\ \mathbf{dispose}\ \mathsf{j} + 1\ ;\ \mathsf{k} := \mathbf{cons}(3, 4)\ ;\ \mathsf{i} := \mathbf{cons}(\mathsf{j}, \mathsf{k})$$

could produce the skewed sharing illustrated at the beginning of this section if the new record allocated by the second **cons** operation were placed at locations $\mathsf{j} + 1$ and $\mathsf{j} + 2$.

A simple solution (reminiscent of the **free** operation in C) is to replace **dispose** e with an command **dispose** (e, n) that disposes of an entire n-field record — and then to require that this record must have been created by an execution of **cons**. The relevant inference rules are:

- The local form (FCDISL)

$$\frac{}{\{e \overset{!}{\mapsto} -^n\}\ \mathbf{dispose}\,(e, n)\ \{\mathbf{emp}\}.}$$

- The global (and backward-reasoning) form (FCDISG)

$$\frac{}{\{(e \overset{!}{\mapsto} -^n)\ *\ r\}\ \mathbf{dispose}\,(e, n)\ \{r\}.}$$

(Here $-^n$ denotes a list of n occurrences of $-$.)

Exercise 15

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

$$\begin{aligned} |a| &= [a] \quad \text{when } a \text{ is an atom} \\ |(t_0 \cdot t_1)| &= |\tau_0| \cdot |\tau_1|. \end{aligned}$$

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

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

More precisely, the procedure should satisfy

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

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