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 [99]. The set S-exps of S-expressions is the least set such that

\[ \tau \in \text{S-exps} \iff \tau \in \text{Atoms} \]

or \( \tau = (\tau_0 \cdot \tau_1) \) where \( \tau_0, \tau_1 \in \text{S-exps} \).

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 \( \text{tree } \tau (i) \) — read “\( i \) is (the root of) a tree representing the S-expression \( \tau \)” — by structural induction on \( \tau \):

\[
\text{tree } a (i) \iff \text{emp} \land i = a \quad \text{when } a \text{ is an atom}
\]

\[
\text{tree } (\tau_0 \cdot \tau_1) (i) \iff \exists i_0, i_1. \ i \mapsto i_0, i_1 \ast \text{tree } \tau_0 (i_0) \ast \text{tree } \tau_1 (i_1).
\]
One can show that the assertions tree τ (i) and ∃τ. tree τ (i) are precise.

To illustrate the use of this definition, we define and verify a recursive procedure copytree(j; i) that nondestructively copies the tree i to j, i.e., that satisfies {tree τ (i)} copytree(j; i){τ} {tree τ (i) * tree τ (j)}. (Here τ is a ghost parameter.) Our proof is an annotated specification that is an instance of the first premiss of the rule (SRPROC):

\[
\begin{align*}
& \{\text{tree } \tau(i)\} \text{ copytree}(j; i)\{\tau\} \{\text{tree } \tau(i) * \text{ tree } \tau(j)\} \leftarrow \\
& \{\text{tree } \tau(i)\} \\
& \text{if } \text{isatom}(i) \text{ then} \\
& \quad \{\text{isatom}(\tau) \land \text{emp} \land i = \tau\} \\
& \quad \{\text{isatom}(\tau) \land ((\text{emp} \land i = \tau) * (\text{emp} \land i = \tau))\} \\
& \quad j := i \\
& \quad \{\text{isatom}(\tau) \land ((\text{emp} \land i = \tau) * (\text{emp} \land j = \tau))\} \\
& \text{else} \\
& \quad \{\exists \tau_0, \tau_1. \ \tau = (\tau_0 \cdot \tau_1) \land \text{tree } (\tau_0 \cdot \tau_1)(i)\} \\
& \quad \text{newvar } i_0, i_1, j_0, j_1 \text{ in } (i_0 := [i]; i_1 := [i + 1]; \\
& \quad \{\text{tree } \tau_0 (i_0) * \text{ tree } \tau_1 (i_1)\} \\
& \quad \{\text{tree } \tau_0 (i_0)\} \\
& \quad \text{copytree}(j_0; i_0)\{\tau_0\}; \\
& \quad \{\text{tree } \tau_0 (i_0) * \text{ tree } \tau_0 (j_0)\} \\
& \quad \{\text{tree } \tau_1 (i_1)\} \\
& \quad \text{copytree}(j_1; i_1)\{\tau_1\}; \\
& \quad \{\text{tree } \tau (i_1) * \text{ tree } \tau_1 (j_1)\} \\
& \quad \{\text{tree } \tau_0 (i_0) * \text{ tree } \tau_1 (i_1) * \text{ tree } \tau_0 (j_0) * \text{ tree } \tau_1 (j_1)\} \\
& \quad \text{if } j := \text{cons}(j_0, j_1) \\
& \quad \{\text{tree } \tau_0 (i_0) * \text{ tree } \tau_1 (i_1) * \text{ tree } \tau_0 (j_0) * \text{ tree } \tau_1 (j_1)\} \\
& \quad \{\exists \tau_0, \tau_1. \ \tau = (\tau_0 \cdot \tau_1) \land (\text{tree } (\tau_0 \cdot \tau_1)(i) * \text{ tree } (\tau_0 \cdot \tau_1)(j))\}\} \\
& \{\text{tree } \tau(i) * \text{ tree } \tau(j)\}. 
\end{align*}
\]
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

\[
\text{copytree}(j; i) =
\begin{align*}
&\text{if isatom}(i) \text{ then } j := i \text{ else} \\
&\text{newvar } i_0, i_1, j_0, j_1 \text{ in} \\
&(i_0 := [i] ; i_1 := [i + 1] ; \\
&\text{copytree}(j_0; i_0) ; \text{copytree}(j_1; i_1) ; j := \text{cons}(j_0, j_1)).
\end{align*}
\] (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 \( \text{dag } \tau (i) \) — read “\( i \) is (the root of ) a dag representing the S-expression \( \tau \) — by structural induction on \( \tau \):

\[
\begin{align*}
\text{dag } a (i) &\iff i = a \quad \text{when } a \text{ is an atom} \\
\text{dag } (\tau_0 \cdot \tau_1) (i) &\iff \exists i_0, i_1. \ i \mapsto i_0, i_1 \ast (\text{dag } \tau_0 (i_0) \land \text{dag } \tau_1 (i_1)).
\end{align*}
\]

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 \( \text{dag } \tau (i) \) meant that the heap contained the dag representing \( \tau \) and nothing else, then \( \text{dag } \tau_0 (i_0) \land \text{dag } \tau_1 (i_1) \) would imply that \( \tau_0 \) and \( \tau_1 \) have the same representation (and are therefore the same S-expression). But we have dropped emp from the base case, so that \( \text{dag } \tau (i) \) only means that a dag representing \( \tau \) occurs somewhere within the heap. In fact,

**Proposition 16** (1) \( \text{dag } \tau (i) \) and (2) \( \exists \tau. \ \text{dag } \tau (i) \) are intuitionistic assertions.

**Proof** (1) By induction on \( \tau \), using the fact that an assertion \( p \) is intuitionistic iff \( p \ast \text{true} \Rightarrow p \). If \( \tau \) is an atom \( a \), then

\[
\begin{align*}
\text{dag } a (i) \ast \text{true} &\Rightarrow i = a \ast \text{true} \\
&\Rightarrow i = a \\
&\Rightarrow \text{dag } a (i),
\end{align*}
\]
since \( i = a \) is pure. Otherwise, \( \tau = (\tau_0 \cdot \tau_1) \), and

\[
\text{dag} (\tau_0 \cdot \tau_1) (i) \ast \text{true}
\]

\[
\Rightarrow \exists i_0, i_1. \ i \mapsto i_0, i_1 \ast (\text{dag} \ \tau_0 (i_0) \land \text{dag} \ \tau_1 (i_1)) \ast \text{true}
\]

\[
\Rightarrow \exists i_0, i_1. \ i \mapsto i_0, i_1 \ast ((\text{dag} \ \tau_0 (i_0) \ast \text{true}) \land (\text{dag} \ \tau_1 (i_1) \ast \text{true}))
\]

\[
\Rightarrow \exists i_0, i_1. \ i \mapsto i_0, i_1 \ast (\text{dag} \ \tau_0 (i_0) \land \text{dag} \ \tau_1 (i_1))
\]

\[
\Rightarrow \text{dag} (\tau_0 \cdot \tau_1) (i),
\]

by the induction hypothesis for \( \tau_0 \) and \( \tau_1 \).

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

\[
(\exists \tau. \ \text{dag} \ \tau (i)) \ast \text{true}
\]

\[
\Rightarrow \exists \tau. (\text{dag} \ \tau (i) \ast \text{true})
\]

\[
\Rightarrow \exists \tau. \ \text{dag} \ \tau (i).
\]

END OF PROOF

Moreover,

**Proposition 17**  (1) For all \( i, \tau_0, \tau_1, h_0, h_1 \), if \( h_0 \cup h_1 \) is a function, and

\[
[i: i \mid \tau: \tau_0], h_0 \models \text{dag} \ \tau (i) \quad \text{and} \quad [i: i \mid \tau: \tau_1], h_1 \models \text{dag} \ \tau (i),
\]

then \( \tau_0 = \tau_1 \) and

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

(2) \( \text{dag} \ \tau \ i \) is a supported assertion. (3) \( \exists \tau. \ \text{dag} \ \tau (i) \) is a supported assertion.

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

\[
h = [i: i_l \mid i + 1: i_r] \cdot h'
\]

\[
[i: i_l \mid \tau: \tau_l], h' \models \text{dag} \ \tau (i)
\]

\[
[i: i_r \mid \tau: \tau_r], h' \models \text{dag} \ \tau (i).
\]

(1) The proof is by structural induction on \( \tau_0 \). For the base case, suppose \( \tau_0 \) is an atom \( a \). Then by (a), \( i = a \). Moreover, if \( \tau_1 \) were not an atom, then
by (b) we would have the contradiction that \( i \) is not an atom. Thus \( \tau_1 \) must be atom \( a' \), and by (a), \( i = a' \), so that \( \tau_0 = \tau_1 = i = a = a' \). Then, also by (a), \([i : i | \tau : \tau_0], h \models \text{dag } \tau (i)\) holds for any \( h \).

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

\[
h_0 = [i : i_{0l} | i + 1 : i_{0r}] \cdot h'_0 \\
[i : i_{0l} | \tau : \tau_{0l}], h'_0 \models \text{dag } \tau (i) \\
[i : i_{0r} | \tau : \tau_{0r}], h'_0 \models \text{dag } \tau (i) .
\]

Moreover, if \( \tau_1 \) were an atom, then by (a) we would have the contradiction that \( i \) is an atom. Thus, \( \tau_1 \) must have the form \((\tau_{1l}, \tau_{1r})\), so that by (b) there are \( i_{1l}, i_{1r}, \) and \( h'_1 \) such that

\[
h_1 = [i : i_{1l} | i + 1 : i_{1r}] \cdot h'_1 \\
[i : i_{1l} | \tau : \tau_{1l}], h'_1 \models \text{dag } \tau (i) \\
[i : i_{1r} | \tau : \tau_{1r}], h'_1 \models \text{dag } \tau (i) .
\]

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 \([i : i_{0l} | i + 1 : i_{0r}] = [i : i_{1l} | i + 1 : i_{1r}]\), so that \( i_{0l} = i_{1l} \) and \( i_{0r} = i_{1r} \), and also,

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

Then, since

\[
[i : i_{0l} | \tau : \tau_{0l}], h'_0 \models \text{dag } \tau (i) \text{ and } [i : i_{1l} | \tau : \tau_{1l}], h'_1 \models \text{dag } \tau (i) ,
\]

the induction hypothesis for \( \tau_{0l} \) gives

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

and the induction hypothesis for \( \tau_{0r} \) gives

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

Thus, (b) gives

\[
[i : i | \tau : (\tau_{0l}, \tau_{0r})], h_0 \cap h_1 \models \text{dag } \tau (i) ,
\]
which, with $\tau_0 = (\tau_{0l} \cdot \tau_{0r}) = (\tau_{1l} \cdot \tau_{1r}) = \tau_1$, establishes (1).

(2) Since $\tau$ and $i$ are the only free variables in $\text{dag } \tau (i)$, we can regard $s$ and $[i: i \mid \tau: \tau]$, where $i = s(i)$ 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 $\exists \tau. \text{dag } \tau (i)$, we can regard $s$ and $[i: i]$, where $i = s(i)$, as equivalent stores. Then we can use the semantic equation for the existential quantifier to show that there are S-expressions $\tau_0$ and $\tau_1$ such that (5.2) holds. Then (1) and the semantic equation for existentials shows that $[i: i], h_0 \cap h_1 \models \text{dag } \tau (i)$, and (3) follows since $h_0 \cap h_1$ is a subset of both $h_0$ and $h_1$.

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

$$\text{Pr } p \overset{\text{def}}{=} p \land \neg(p \star \neg \text{emp}),$$

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

$$\text{dag } \tau (i) \land \neg(\text{dag } \tau (i) \star \neg \text{emp})$$

$$\left(\exists \tau. \text{dag } \tau (i)\right) \land \neg((\exists \tau. \text{dag } \tau (i)) \star \neg \text{emp}),$$

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

Now consider the procedure $\text{copytree}(j; i)$. 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

$$\{\text{dag } \tau(i)\} \text{copytree}(j; i)\{\tau\} \{\text{dag } \tau(i) \star \text{tree } \tau(j)\}.$$

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 $\text{copytree}$, then we will be unable to prove the necessary property of the first recursive call:

$$\{i \mapsto i_0, i_1 \star (\text{dag } \tau_{0}(i_0) \land \text{dag } \tau_{1}(i_1))\}$$

$$\text{copytree}(j_0; i_0)\{\tau_0\}$$

$$\{i \mapsto i_0, i_1 \star (\text{dag } \tau_{0}(i_0) \land \text{dag } \tau_{1}(i_1)) \star \text{tree } \tau_{0}(j_0)\}.$$

(Here, we ignore pure assertions in the pre- and postconditions that are irrelevant to this argument.) But the hypothesis is not strong enough to
5.2. DAGS

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 \( \text{copytree}(i_0, \tau; j_0) \) might change the state from

\[
\begin{array}{c}
\text{i}_0 \rightarrow\ 3 \\
\quad \rightarrow 4 \\
\text{i}_1 \rightarrow\ 5 \\
\quad \rightarrow 6
\end{array}
\]

into

\[
\begin{array}{c}
\text{i}_0 \rightarrow\ 5 \\
\quad \rightarrow 6 \\
\text{i}_1 \rightarrow\ 3 \\
\quad \rightarrow 4
\end{array}
\]

where \( \text{dag} \tau_1 (i_1) \) is false.

To circumvent this problem, we must strengthen the specification of \( \text{copytree} \) 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 and parameters denoting heaps. Suppose \( h_0 \) is such a variable, and the assertion \( \text{this}(h_0) \) is true just in the heap that is the value of \( h_0 \). Then we could specify

\[
\{ \text{this}(h_0) \land \text{dag} \tau(i) \} \ \text{copytree}(j; i) \{ \tau, h_0 \} \ \{ \text{this}(h_0) \ast \text{tree} \tau(j) \}.\]

2. Introduce ghost variables and parameters 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:

\[
\{ p \land \text{dag} \tau(i) \} \ \text{copytree}(j; i) \{ \tau, p \} \ \{ p \ast \text{tree} \tau(j) \}.\]

3. Introduce fractional permissions [79], or some other form of assertion that part of the heap is read-only or passive. Then one could define an assertion \( \text{passdag} \tau(i) \) describing a read-only heap containing a dag, and use it to specify that the initial heap is at no time altered by \( \text{copytree} \):

\[
\{ \text{passdag} \tau(i) \} \ \text{copytree}(j; i) \{ \tau \} \ \{ \text{passdag} \tau(i) \ast \text{tree} \tau(j) \}.\]

(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:

\[ \text{AStores}_A = A \rightarrow (\text{Heaps} \rightarrow B) \]
\[ \text{States}_{AV} = \text{AStores}_A \times \text{Stores}_V \times \text{Heaps}, \]

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:

\[ as, s, h \models a \iff as(a)(h). \]

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 \( AV(p) \) for the assertion variables occurring free in \( p \). (Since, we have not introduced any binders of assertion variables, all of their occurrences are free.)

**Proposition 18** (Generalized Partial Substitution Law for Assertions) Suppose \( p \) is an assertion, and let \( \delta \) abbreviate the substitution

\[ a_1 \rightarrow p_1, \ldots, a_m \rightarrow p_m, v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n, \]

Then let \( s \) be a store such that

\[ (\text{FV}(p) - \{v_1, \ldots, v_n\}) \cup \text{FV}(p_1, \ldots, p_m, e_1, \ldots, e_n) \subseteq \text{dom} s, \]
and let as be an assertion store such that
\[(\text{AV}(p) - \{a_1, \ldots, a_m\}) \cup \text{AV}(p_1, \ldots, p_m) \subseteq \text{dom as},\]
and let
\[
\hat{s} = [s \mid v_1: [e_1]_{\exp s} \mid \ldots \mid v_n: [e_n]_{\exp s}]
\]
\[
\hat{\text{as}} = [as \mid a_1: \lambda h. (as, s, h \models p_1) \mid \ldots \mid a_m: \lambda h. (as, s, h \models p_m)]
\]
Then
\[as, s, h \models (p/\delta) \iff \hat{\text{as}}, \hat{s}, h \models p.\]

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)**
  \[
  \frac{\{p\} \ c \ \{q\}}{\{p/\delta\} (c/\delta) \ \{q/\delta\}},
  \]
  where \(\delta\) is the substitution
  \[
  \delta = a_1 \rightarrow p_1, \ldots, a_m \rightarrow p_m, v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n;
  \]
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 \(p_j\).

- **Substitution (SUBan)**
  \[
  \frac{\{p\} \ c \ \{q\}}{\{\mathcal{A}\}/\delta \gg \{p/\delta\} (c/\delta) \ \{q/\delta\}},
  \]
  where \(\delta\) is the substitution
  \[
  \delta = a_1 \rightarrow p_1, \ldots, a_m \rightarrow p_m, v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n;
  \]
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 \{ a \} x := y \{ a \}, for example, we can substitute \( a \rightarrow (y = z), x \rightarrow x, y \rightarrow y \) to obtain

\[ \{ y = z \} x := y \{ y = z \}, \]

but we cannot substitute \( a \rightarrow (x = z), x \rightarrow x, y \rightarrow y \) to obtain

\[ \{ x = z \} x := y \{ x = z \}. \]

We must also extend the rules for annotated specifications of procedure calls and definitions to permit assertion variables to be used as ghost parameters. The details are left to the reader.

### 5.4 Copying Dags to Trees

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

\[
\{ p \land \text{dag} \, \tau(i) \} \text{copytree}(j; i) \{ \tau, p \} \{ p \ast \text{tree} \, \tau(j) \}.
\]

In this specification, we can substitute \( \text{dag} \, \tau(i) \) for \( p \) to obtain the weaker specification

\[
\{ \text{dag} \, \tau(i) \} \text{copytree}(j; i) \{ \tau, \text{dag} \, \tau(i) \} \{ \text{dag} \, \tau(i) \ast \text{tree} \, \tau(j) \},
\]

but, as we have seen, the latter specification is too weak to serve as a recursion hypothesis.
Our proof is again an annotated instance of the first premiss of (SRPROC):

\[
\{ p \land \text{dag } \tau(i) \} \xrightarrow{\text{copytree}(j; i)} \{ \tau, p \} \{ p \ast \text{tree } \tau(j) \} \vdash
\{ p \land \text{dag } \tau(i) \}
\]

if isatom(i) then

\[
\begin{align*}
&\{ p \land \text{isatom}(\tau) \land \tau = i \} \\
&\{ p \ast (\text{isatom}(\tau) \land \tau = i \land \text{emp}) \} \\
&j := i \\
&\{ p \ast (\text{isatom}(\tau) \land \tau = j \land \text{emp}) \}
\end{align*}
\]

else

\[
\{ \exists \tau_0, \tau_1. \, \tau = (\tau_0 \cdot \tau_1) \land p \land \text{dag } (\tau_0 \cdot \tau_1)(i) \} \\
\text{newvar } i_0, i_1, j_0, j_1 \text{ in}
\begin{align*}
&\left( i_0 := [i] ; i_1 := [i + 1] ; \right. \\
&\{ p \land (i \mapsto i_0, i_1 \ast (\text{dag } \tau_0(i_0) \land \text{dag } \tau_1(i_1))) \} \\
&\{ p \land (\text{true} \ast (\text{dag } \tau_0(i_0) \land \text{dag } \tau_1(i_1))) \} \\
&\{ p \land ((\text{true} \ast \text{dag } \tau_0(i_0)) \land (\text{true} \ast \text{dag } \tau_1(i_1))) \} \\
&\{ p \land \text{dag } \tau_1(i_1) \land \text{dag } \tau_0(i_0) \} \quad (\ast) \\
&\text{copytree}(j_0; i_0)\{ \tau_0, p \land \text{dag } \tau_1(i_1) \} \\
&\{ (p \land \text{dag } \tau_1(i_1)) \ast \text{tree } \tau_0(j_0) \} \\
&\{ p \land \text{dag } \tau_1(i_1) \} \\
&\text{copytree}(j_1; i_1)\{ \tau_1, p \} \quad \ast \text{tree } \tau_0(j_0) \\
&\{ p \ast \text{tree } \tau_1(j_1) \} \\
&\{ p \ast \text{tree } \tau_0(j_0) \ast \text{tree } \tau_1(j_1) \} \\
&j := \text{cons}(j_0, j_1) \\
&\{ p \ast j \mapsto j_0, j_1 \ast \text{tree } \tau_0(j_0) \ast \text{tree } \tau_1(j_1) \} \\
&\{ \exists \tau_0, \tau_1. \, (\tau = (\tau_0 \cdot \tau_1) \land p) \ast \text{tree } (\tau_0 \cdot \tau_1)(j) \} \\
&\{ p \ast \text{tree } \tau(j) \}.
\end{align*}
\]

(Here, the assertion marked (\ast) is obtained from the preceding assertion by using true \ast \text{dag } \tau(i) \Rightarrow \text{dag } \tau(i), which holds since \text{dag } \tau(i) is intuitionistic.)
5.5 Substitution in S-expressions

To investigate further programs dealing with trees and dags, we consider the substitution of S-expressions for atoms in S-expressions. We write \( \tau /a \rightarrow \tau' \) for the result of substituting \( \tau' \) for the atom \( a \) in \( \tau \), which is defined by structural induction on \( \tau \):

\[
\begin{align*}
a/a & \rightarrow \tau' = \tau' \\
b/a & \rightarrow \tau' = b \quad \text{when } b \in \text{Atoms} - \{a\} \\
(\tau_0 \cdot \tau_1)/a & \rightarrow \tau' = ((\tau_0/a \rightarrow \tau') \cdot (\tau_1/a \rightarrow \tau')).
\end{align*}
\]

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.

5.5.1 Substitution with Copying

We will define a procedure that, given a tree representing \( \tau \) and a dag representing \( \tau' \), produces a tree representing \( \tau/a \rightarrow \tau' \), i.e.,

\[
\{\text{tree } \tau (i) * \text{ dag } \tau' (j)\} \\
\text{subst}(i; a, j)\{\tau, \tau'\} \\
\{\text{tree } \tau/a \rightarrow \tau' (i) * \text{ dag } \tau' (j)\}.
\]

The procedure \texttt{copytree} will be used to copy the dag at \( j \) each time the atom \( a \) is encountered in the tree at \( i \).

The following is an annotated specification of the procedure body, in which \( D \) abbreviates the assertion \( \text{dag } \tau' (j) \):
5.5. **SUBSTITUTION IN S-EXPRESSIONS**

\[
\{\text{tree } \tau(i) \ast D\} \quad \text{subst}(i; a, j)\{\tau, \tau'\} \quad \{\text{tree } (\tau / a \rightarrow \tau') (i) \ast D\} \\
\{\text{tree } \tau(i) \ast D\}
\]

if isatom(i) then
\[
\{((\text{isatom}(\tau) \land \tau = i \land \text{emp}) \ast D\}
\]
if \(i = a\) then
\[
\{((\text{isatom}(\tau) \land \tau = a \land \text{emp}) \ast D\}
\]
\[
\{((\tau / a \rightarrow \tau') = \tau' \land \text{emp}) \ast D\}
\]
\[
\{D\}
\]
\[
\text{copytree}(i; j)\{\tau'\} \ast ((\tau / a \rightarrow \tau') = \tau' \land \text{emp})
\]
\[
\{D \ast \text{tree } \tau'(i)\}
\]
\[
\{\text{tree } (\tau / a \rightarrow \tau')(i) \ast D\}
\]
else
\[
\{((\text{isatom}(\tau) \land \tau \neq a \land \tau = i \land \text{emp}) \ast D\}
\]
\[
\{((\tau / a \rightarrow \tau') = \tau \land \text{isatom}(\tau) \land \tau = i \land \text{emp}) \ast D\}
\]
skip
\[
\{\text{tree } (\tau / a \rightarrow \tau')(i) \ast D\}
\]
else
\[
\{\exists \tau_0, \tau_1, i_0, i_1. \quad \tau = (\tau_0 \cdot \tau_1) \land (i \mapsto i_0, i_1 \ast \text{tree } \tau_0 (i_0) \ast \text{tree } \tau_1 (i_1) \ast D)\}
\]
\[
\text{newvar} \quad i_0, i_1 \quad \text{in} \quad \left(i_0 := [i]; i_1 := [i + 1]; \right)
\]
\[
\{\text{tree } \tau_0 (i_0) \ast \text{tree } \tau_1 (i_1) \ast D\}
\]
\[
\{\text{tree } \tau_0 (i_0) \ast D\}
\]
\[
\text{subst}(i_0; a, j)\{\tau_0, \tau'\}; \ast \text{tree } \tau_1 (i_1)
\]
\[
\{\text{tree } (\tau_0 / a \rightarrow \tau')(i_0) \ast D\}
\]
\[
\{\text{tree } \tau_1 (i_1) \ast D\}
\]
\[
\text{subst}(i_1; a, j)\{\tau_1, \tau'\}; \ast \text{tree } (\tau_0 / a \rightarrow \tau')(i_0)
\]
\[
\{\text{tree } (\tau_1 / a \rightarrow \tau')(i_1) \ast D\}
\]
\[
\{\text{tree } (\tau_0 / a \rightarrow \tau')(i_0) \ast \text{tree } (\tau_1 / a \rightarrow \tau')(i_1) \ast D\}
\]
\[
[i] := i_0; [i + 1] := i_1
\]
\[
\{\tau = (\tau_0 \cdot \tau_1) \land (i \mapsto i_0, i_1 \ast \text{tree } (\tau_0 / a \rightarrow \tau')(i_0) \ast \text{tree } (\tau_1 / a \rightarrow \tau')(i_1) \ast D)\}
\]
\[
\{\exists \tau_0, \tau_1. \quad \tau = (\tau_0 \cdot \tau_1) \land (\text{tree } ((\tau_0 / a \rightarrow \tau') \cdot (\tau_1 / a \rightarrow \tau')) (i) \ast D)\}
\]
\[
\{\text{tree } (\tau / a \rightarrow \tau')(i) \ast D\}.
\]
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

\[
\text{subst}(i; a, j) =
\]

\[
\begin{align*}
\text{if} & \quad \text{isatom}(i) \quad \text{then} \quad \text{if} \quad i = a \quad \text{then} \quad \text{copytree}(i; j) \quad \text{else} \quad \text{skip} \\
\text{else} & \quad \text{newvar} \quad i_0, i_1 \quad \text{in} \quad (i_0 := [i]; i_1 := [i + 1]; \\
& \quad \text{subst}(i_0; a, j); \text{subst}(i_1; a, j); [i] := i_0 ; [i + 1] := i_1).
\end{align*}
\]

In fact, this procedure satisfies a stronger specification than 5.3. One can use an assertion variable to specify that the dag representing the tree \(\tau'\) is not changed by executing \(\text{subst}\):

\[
\{\text{tree } \tau (i) \ast (p \land \text{dag } \tau' (j))\}
\]

\[
\text{subst}(i; a, j)\{\tau, \tau', p\}
\]

\[
\{\text{tree } (\tau/a \rightarrow \tau') (i) \ast (p \land \text{dag } \tau' (j))\}.
\]

The proof is obtained from the annotated specification given above by taking \(D\) to be \(p \land \text{dag } \tau' (j)\) and adding \(p\) as a ghost parameter to the calls of \(\text{subst}\).

### 5.5.2 Substitution without Copying

An interesting variation on substitution into S-expressions occurs when the representation of the S-expression \(\tau'\) being substituted is not copied, but the corresponding atom is replaced by pointers to the representation of \(\tau'\). In this case, the representation of the expression \(\tau\) undergoing substitution changes from a tree to a dag.

The procedure itself is a minor variation on that in the preceding section:

\[
\text{subst2}(i; a, j) =
\]

\[
\begin{align*}
\text{if} & \quad \text{isatom}(i) \quad \text{then} \quad \text{if} \quad i = a \quad \text{then} \quad i := j \quad \text{else} \quad \text{skip} \\
\text{else} & \quad \text{newvar} \quad i_0, i_1 \quad \text{in} \quad (i_0 := [i]; i_1 := [i + 1]; \\
& \quad \text{subst2}(i_0; a, j); \text{subst2}(i_1; a, j); [i] := i_0 ; [i + 1] := i_1),
\end{align*}
\]

but finding the best specification of this procedure is subtle. A naive specification would be

\[
\{\text{tree } \tau (i) \ast \text{dag } \tau' (j)\}
\]

\[
\text{subst2}(i; a, j)\{\tau, \tau'\}
\]

\[
\{\text{dag } (\tau/a \rightarrow \tau') (i) \land \{\text{dag } \tau' (j)\}\}.
\]
5.5. **SUBSTITUTION IN S-EXPRESSIONS**

A stronger specification, however, separates the dag at \( j \), representing \( \tau' \), from the rest of the heap, which would represent \( \tau/a \rightarrow \tau' \) if one added any dag at \( j \) representing \( \tau' \):

\[
\{ \text{tree } \tau (i) \ast \text{dag } \tau' (j) \} \\
\text{subst2}(i; a, j)\{ \tau, \tau' \} \\
\{ (\text{dag } \tau' (j) \ast \text{dag } (\tau/a \rightarrow \tau') (i)) \ast \text{dag } \tau' (j) \}.
\]

But this representation is not local, since the part of the heap representing \( \tau' \) is not touched by the procedure, and is not part of its footprint. A local specification would be

\[
\{ \text{tree } \tau (i) \} \\
\text{subst2}(i; a, j)\{ \tau, \tau' \} \\
\{ \text{dag } \tau' (j) \ast \text{dag } (\tau/a \rightarrow \tau') (i) \}.
\]

From this specification, one can regain the naive one by using the frame rule, the axiom schema

\[(q \ast \ast p) \ast q \Rightarrow p \land (q \ast \text{true})\]

(which is a straightforward consequence of the schema (2.5)), and the fact that \( i \ast \text{true} \Rightarrow i \) when \( i \) is intuitionistic:

\[
\{ \text{tree } \tau (i) \ast \text{dag } \tau' (j) \} \\
\{ \text{tree } \tau (i) \} \\
\text{subst2}(i; a, j)\{ \tau, \tau' \} \\
\{ \text{dag } \tau' (j) \ast \text{dag } (\tau/a \rightarrow \tau') (i) \ast \text{dag } \tau' (j) \} \\
\{ (\text{dag } \tau' (j) \ast \text{dag } (\tau/a \rightarrow \tau') (i)) \ast \text{dag } \tau' (j) \} \\
\{ \text{dag } (\tau/a \rightarrow \tau') (i) \land (\text{dag } \tau' (j) \ast \text{true}) \} \\
\{ \text{dag } (\tau/a \rightarrow \tau') (i) \land \text{dag } \tau' (j) \}
\]

Before proceeding further, we must derive several inference rules. For brevity, we elide applications of the associative and commutative laws for separating conjunction.

To derive: \( \text{emp} \Rightarrow (p \ast \ast p) \) (5.5)
1. \( (\text{emp} \ast p) \Rightarrow p \)  
   \((p \ast \text{emp} \Rightarrow p)\)

2. \( \text{emp} \Rightarrow (p \ast p) \)  
   \((\text{currying, 1})\)

To derive: \[
\frac{p \Rightarrow q \quad p \Rightarrow r}{p \Rightarrow (q \land r)} \quad (5.6)
\]

1. \( p \Rightarrow q \)  
   (assumption)

2. \( p \Rightarrow r \)  
   (assumption)

3. \( q \Rightarrow (r \Rightarrow (q \land r)) \)  
   \((p \Rightarrow (q \Rightarrow (p \land q)))\)

4. \( p \Rightarrow (r \Rightarrow (q \land r)) \)  
   \((\text{trans impl, 1, 3})\)

5. \( (p \Rightarrow (r \Rightarrow (q \land r))) \Rightarrow ((p \Rightarrow r) \Rightarrow (p \Rightarrow (q \land r))) \)  
   \(((p \Rightarrow (q \Rightarrow (p \land q))) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)))\)

6. \( (p \Rightarrow r) \Rightarrow (p \Rightarrow (q \land r)) \)  
   \((\text{modus ponens, 4, 5})\)

7. \( p \Rightarrow (q \land r) \)  
   \((\text{modus ponens, 2, 6})\)

To derive: \[
(p \ast i_0) \ast (p \ast i_1) \Rightarrow (p \ast (i_0 \land i_1)) \quad \text{when } i_0 \text{ and } i_1 \text{ are intuitionistic.} \quad (5.7)
\]

1. \( (p \ast i_0) \Rightarrow (p \ast i_0) \)  
   \((p \Rightarrow p)\)

2. \( (p \ast i_0) \ast p \Rightarrow i_0 \)  
   \((\text{decuring, 1})\)

3. \( (p \ast i_1) \Rightarrow \text{true} \)  
   \((p \Rightarrow \text{true})\)

4. \( (p \ast i_0) \ast (p \ast i_1) \ast p \Rightarrow i_0 \ast \text{true} \)  
   \((\text{monotonicity, 2, 3})\)

5. \( i_0 \ast \text{true} \Rightarrow i_0 \)  
   \((i \ast p \Rightarrow i)\)
5.5. SUBSTITUTION IN S-EXPRESSIONS

6. \((p \rightarrow i_0) \ast (p \rightarrow i_1) \ast p \Rightarrow i_0\) (trans impl, 4, 5)

7. \((p \rightarrow i_0) \ast (p \rightarrow i_1) \ast p \Rightarrow i_1\) (similarly)

8. \((p \rightarrow i_0) \ast (p \rightarrow i_1) \ast p \Rightarrow (i_0 \land i_1)\) (5.6, 6, 7)

9. \((p \rightarrow i_0) \ast (p \rightarrow i_1) \Rightarrow (p \rightarrow (i_0 \land i_1))\) (currying, 8)

To derive: \(p \ast (q \rightarrow r) \Rightarrow (q \rightarrow (p \ast r))\) (5.8)

1. \(p \Rightarrow p\) (\(p \Rightarrow p\))

2. \((q \rightarrow r) \Rightarrow (q \rightarrow r)\) (\(p \Rightarrow p\))

3. \((q \rightarrow r) \ast q \Rightarrow r\) (decurrying, 2)

4. \(p \ast (q \rightarrow r) \ast q \Rightarrow p \ast r\) (monotonicity, 1, 3)

5. \(p \ast (q \rightarrow r) \Rightarrow (q \rightarrow (p \ast r))\) (currying, 4)
Now we are prepared to prove (5.4). Let $D$ abbreviate $\text{dag } \tau'(j)$. Then the body of the procedure $\text{subst2}$ will meet the specification

$$\{	ext{tree } \tau(i)\} \text{subst2}(i; a, j)\{\tau, \tau'\} \{D \rightarrow \text{dag } (\tau/a \rightarrow \tau')(i)\} \vdash$$

$$\{	ext{tree } \tau(i)\}$$

if $\text{isatom}(i)$ then

$$\{\text{isatom}(\tau) \land \tau = i \land \text{emp}\}$$

if $i = a$ then

$$\{\text{isatom}(\tau) \land \tau = a \land \text{emp}\}$$

$$\{(\tau/a \rightarrow \tau') = \tau' \land \text{emp}\}$$

$$\{(\tau/a \rightarrow \tau') = \tau' \land (D \rightarrow D)\} \quad (5.5)$$

$$\{D \rightarrow \text{dag } (\tau/a \rightarrow \tau')(j)\}$$

$$i := j$$

$$\{D \rightarrow \text{dag } (\tau/a \rightarrow \tau')(i)\}$$

else

$$\{\text{isatom}(\tau) \land \tau \neq a \land \tau = i \land \text{emp}\}$$

$$\{(\tau/a \rightarrow \tau') = \tau \land \text{isatom}(\tau) \land \tau = i\}$$

$$\{(\tau/a \rightarrow \tau') = \tau \land \text{dag } \tau(i)\}$$

$$\{\text{dag } (\tau/a \rightarrow \tau')(i)\}$$

$$\text{skip}$$

$$\{D \rightarrow \text{dag } (\tau/a \rightarrow \tau')(i)\} \quad \text{(Section 2.3.4)}$$

\[\vdots\]
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,

$$\dag ((1 \cdot 2) \cdot (2 \cdot 3)) (i)$$
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 $\phi$ from the domain of the heap to natural numbers, called the field count. Then, when $v := \text{cons}(e_1, \ldots, e_n)$ creates an $n$-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. In other words, 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

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

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.

To describe the field count, we introduce a new assertion of the form $e \mapsto e'$, with the meaning

$$
s, h, \phi \models e \mapsto e' \text{ iff } \text{dom } h = \{ \llbracket e \rrbracket_{\exp}s \} \text{ and } h(\llbracket e \rrbracket_{\exp}s) = \llbracket e' \rrbracket_{\exp}s \text{ and } \phi(\llbracket e \rrbracket_{\exp}s) = \llbracket e' \rrbracket_{\exp}s.
$$

We also introduce the following abbreviations:

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

$$
e \overset{\text{def}}{=} e \mapsto e' * \text{true}
$$

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

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

$$
\text{iff } e \mapsto e_1, \ldots, e_n * \text{true}.
$$
Axiom schema for reasoning about these new assertions include:

\[
\begin{align*}
&\text{If } e \xrightarrow{[n]} e' \Rightarrow e \leftrightarrow e' \\
&\text{and } e \xrightarrow{[m]} - \land e \xrightarrow{[n]} - \Rightarrow m = n \\
&2 \leq k \leq n \land e \xrightarrow{[n]} - \Rightarrow e + k - 1 \xrightarrow{[0]} - \\
&\text{then } e \mapsto e_1, \ldots, e_m \land e' \mapsto e'_1, \ldots, e'_n \land e \neq e' \Rightarrow \\
&e \mapsto e_1, \ldots, e_m \ast e' \mapsto e'_1, \ldots, e'_n \ast \text{true.}
\end{align*}
\]

(The last of these axiom schema 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 (FCCONSNO)

  \[
  \{\text{emp}\} v := \text{cons}(\overline{v}) \{v \mapsto \overline{v}\},
  \]

  where \(v \not\in \text{FV}(\overline{v})\).

- **Mutation:** the local form (FCMUL)

  \[
  \{e \mapsto -\} [e] := e' \{e \mapsto e'\},
  \]

- **Lookup:** the local nonoverwriting form (FCLKNO)

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

  where \(v \not\in \text{FV}(e, \overline{e})\).

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, making the axiom schema in (5.9) unsound. For example, the command

\[
\begin{align*}
\text{j} := \text{cons}(1, 2) \; \text{; dispose} \; \text{j + 1} \; \text{; k} := \text{cons}(3, 4) \; \text{; i} := \text{cons}(j, k)
\end{align*}
\]
could produce the skewed sharing illustrated at the beginning of this section if the new record allocated by the second \texttt{cons} operation were placed at locations $j+1$ and $j+2$.

A simple solution (reminiscent of the \texttt{free} operation in C) is to replace \texttt{dispose} $e$ with a command \texttt{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 \texttt{cons}. This restriction restores the soundness of (5.9).

The relevant inference rules are:

- The local form (FCDISL)

  \[
  \{ e \mapsto ^{\dagger} -n \} \texttt{dispose} (e, n) \{ \texttt{emp} \}.
  \]

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

  \[
  \{(e \mapsto ^{\dagger} -n) * r \} \texttt{dispose} (e, n) \{ r \}.
  \]

(Here $-n$ denotes a list of $n$ occurrences of $-$.)
Exercise 1

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

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

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 flatten that mutates a tree denoting an S-expression $\tau$ into a singly-linked list segment denoting the flattening of $\tau$. 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 $\tau$, the procedure should be given as input, in addition to the tree representing $\tau$, a single two-cell, which will become the initial cell of the list segment that is constructed.

More precisely, the procedure should satisfy

$$
\{\text{tree } \tau \ (i) \ * \ j \mapsto -, -\} \\
\text{flatten}(; \ i, j, k)\{\tau\} \\
\{\text{ls} \ |\tau| \ (j, k)\}.
$$

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

Exercise 2

Show that tree $\tau \ (i)$ and $\exists \tau. \ \text{tree } \tau \ (i)$ are precise.

Exercise 3

Show that tree $\tau \ (i) \Rightarrow \text{dag } \tau \ (i)$ is valid.