An Introduction to Separation Logic

(Preliminary Draft)

John C. Reynolds
Computer Science Department
Carnegie Mellon University

ITU University, Copenhagen

October 20–22, 2008, corrected October 23

Chapter 1

An Overview

Separation logic is a novel system for reasoning about imperative programs. It extends Hoare logic with enriched assertions that can describe the separation of storage and other resources concisely. The original goal of the logic was to facilitate reasoning about shared mutable data structures, i.e., structures where updatable fields can be referenced from more than one point. More recently, the logic has been extended to deal with shared-variable concurrency and information hiding, and the notion of separation has proven applicable to a wider conceptual range, where access to memory is replaced by permission to exercise capabilities, or by knowledge of structure. In a few years, the logic has become a significant research area, with a growing literature produced by a variety of researchers.

1.1 An Example of the Problem

The use of shared mutable data structures is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size.

For conventional logics, the problem with sharing is that it is the default in the logic, while nonsharing is the default in programming, so that declaring all of the instances where sharing does not occur — or at least those instances necessary for correctness — can be extremely tedious.

For example, consider the following program, which performs an in-place

reversal of a list:

$$LREV \stackrel{\text{def}}{=} \mathsf{j} := \mathbf{nil}\,;\, \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\,;\, [\mathsf{i}+1] := \mathsf{j}\,;\, \mathsf{j} := \mathsf{i}\,;\, \mathsf{i} := \mathsf{k}).$$

(Here the notation $[e]$ denotes the contents of the storage at address e.)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β:

$$\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta,$$

where the predicate $\mathbf{list}\,\alpha\,\mathsf{i}$ is defined by induction on the length of α:

$$\mathsf{list}\,\epsilon\,\mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a}\cdot\alpha)\,\mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\,\alpha\,\mathsf{j}$$

(and $\hookrightarrow$ can be read as "points to").

Unfortunately, however, this is not enough, since the program will malfunction if there is any sharing between the lists i and j. To prohibit this we must extend the invariant to assert that only **nil** is reachable from both i and j:

$$\begin{aligned}&(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta)\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i}, \mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j}, \mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}),\end{aligned} \tag{1.1}$$

where

$$\begin{aligned}\mathbf{reachable}(\mathsf{i}, \mathsf{j}) &\stackrel{\text{def}}{=} \exists n \geq 0.\ \mathbf{reachable}_n(\mathsf{i}, \mathsf{j})\\ \mathbf{reachable}_0(\mathsf{i}, \mathsf{j}) &\stackrel{\text{def}}{=} \mathsf{i} = \mathsf{j}\\ \mathbf{reachable}_{n+1}(\mathsf{i}, \mathsf{j}) &\stackrel{\text{def}}{=} \exists \mathsf{a}, \mathsf{k}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{k} \wedge \mathbf{reachable}_n(\mathsf{k}, \mathsf{j}).\end{aligned}$$

Even worse, suppose there is some other list x, representing a sequence γ, that is not supposed to be affected by the execution of our program. Then it must not share with either i or j, so that the invariant becomes

$$\begin{aligned}&(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta)\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i}, \mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j}, \mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})\\ &\quad \wedge \mathsf{list}\,\gamma\,\mathsf{x}\\ &\quad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{x}, \mathsf{k})\\ &\qquad\quad \wedge (\mathbf{reachable}(\mathsf{i}, \mathsf{k}) \vee \mathbf{reachable}(\mathsf{j}, \mathsf{k})) \Rightarrow \mathsf{k} = \mathbf{nil}).\end{aligned} \tag{1.2}$$

Even in this trivial situation, where all sharing is prohibited, it is evident that this form of reasoning scales poorly.

In separation logic, however, this kind of difficulty can be avoided by using a novel logical operation $P * Q$, called the *separating conjunction*, that asserts that P and Q hold for *disjoint* portions of the addressable storage. Since the prohibition of sharing is built into this operation, Invariant (1.1) can be written more succinctly as

$$(\exists\alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} * \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta, \tag{1.3}$$

and Invariant (1.2) as

$$(\exists\alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} * \mathsf{list}\,\beta\,\mathsf{j} * \mathsf{list}\,\gamma\,\mathsf{x}) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta. \tag{1.4}$$

A more general advantage is the support that separation logic gives to *local reasoning*, which underlies the scalability of the logic. For example, one can use (1.3) to prove a *local* specification:

$$\{\mathsf{list}\,\alpha\,\mathsf{i}\}\ \mathit{LREV}\ \{\mathsf{list}\,\alpha^\dagger\,\mathsf{j}\}.$$

In separation logic, this specification implies, not only that the program expects to find a list at i representing α, but also that this list is the *only* addressable storage touched by the execution of *LREV* (often called the *footprint* of *LREV*). If *LREV* is a part of a larger program that also manipulates some separate storage, say containing the list k, then one can use an inference rule due to O'Hearn, called the *frame rule*, to infer directly that the additional storage is unaffected by *LREV*:

$$\{\mathsf{list}\,\alpha\,\mathsf{i} * \mathsf{list}\,\gamma\,\mathsf{k}\}\ \mathit{LREV}\ \{\mathsf{list}\,\alpha^\dagger\,\mathsf{j} * \mathsf{list}\,\gamma\,\mathsf{k}\},$$

thereby avoiding the extra complexity of Invariant (1.4).

In a realistic situation, of course, *LREV* might be a substantial subprogram, and the description of the separate storage might also be voluminous. Nevertheless, one can still reason *locally* about LREV, i.e., while ignoring the separate storage, and then scale up to the combined storage by using the frame rule.

There is little need for local reasoning in proving toy examples. But it provides scalability that is critical for more complex programs.

1.2 Background

A partial bibliography of early work on reasoning about shared mutable data structure is given in Reference [1].

The central concept of separating conjunction is implicit in Burstall's early idea of a "distinct nonrepeating tree system" [2]. In lectures in the fall of 1999, the author described the concept explicitly, and embedded it in a flawed extension of Hoare logic [3, 4]. Soon thereafter, a sound intuitionistic version of the logic was discovered independently by Ishtiaq and O'Hearn [5] and by the author [1]. Realizing that this logic was an instance of the logic of bunched implications [6, 7], Ishtiaq and O'Hearn also introduced a *separating implication* $P \mathrel{-\!\!*} Q$.

The intuitionistic character of this logic implied a monotonicity property: that an assertion true for some portion of the addressable storage would remain true for any extension of that portion, such as might be created by later storage allocation.

In their paper, however, Ishtiaq and O'Hearn also presented a classical version of the logic that does not impose this monotonicity property, and can therefore be used to reason about explicit storage deallocation; they showed that this version is more expressive than the intuitionistic logic, since the latter can be translated into the classical logic.

O'Hearn also went on to devise the frame rule and illustrate its importance [8, 9, 10, 5].

Originally, in both the intuitionistic and classical version of the logic, addresses were assumed to be disjoint from integers, and to refer to entire records rather than particular fields, so that address arithmetic was precluded. Later, the author generalized the logic to permit reasoning about unrestricted address arithmetic, by regarding addresses as integers which refer to individual fields [8, 11]. It is this form of the logic that will be described and used in most of these notes.

Since these logics are based on the idea that the structure of an assertion can describe the separation of storage into disjoint components, we have come to use the term *separation logic*, both for the extension of predicate calculus with the separation operators and for the resulting extension of Hoare logic.

At present, separation logic has been used to manually verify a variety of small programs, as well as a few that are large enough to demonstrate the potential of local reasoning for scalability [12, 10, 13, 14, 15]. In addition:

1. It has been shown that deciding the validity of an assertion in separation logic is not recursively enumerable, even when address arithmetic and the characteristic operation **emp**, $\mapsto$, $*$, and $-\!\!*$, but not $\hookrightarrow$ are prohibited [16, 10]. On the other hand, it has also been shown that, if the characteristic operations are permitted but quantifiers are prohibited, then the validity of assertions is algorithmically decidable within the complexity class PSPACE [16].

2. An iterated form of separating conjunction has been introduced to reason about arrays [17].

3. The logic has been extended to procedures with global variables, where a "hypothetical frame rule" permits reasoning with information hiding [18, 19]. Recently, a further extension to higher-order procedures (in the sense of Algol-like languages) has been developed [20].

4. The logic has been integrated with data refinement [21, 22], and with object-oriented programming (i.e., with a subset of Java) [23, 24].

5. The logic has been extended to shared-variable concurrency with conditional critical regions, where one can reason about the transfer of ownership of storage from one process to another [25, 26]. Further extensions have been made to nonblocking algorithms [27] and to rely/guarantee reasoning [28].

6. In the context of proof-carrying code, separation logic has inspired work on proving run-time library code for dynamic allocation [29].

7. A decision procedure has been devised for a restricted form of the logic that is capable of shape analysis of lists [30].

8. Fractional permissions (in the sense of Boyland [31]) and counting permissions have been introduced so that one can permit several concurrent processes to have read-only access to an area of the heap [32]. This approach has also been applied to program variables [33].

9. Separation logic itself has been extended to a higher-order logic [34].

10. Separation logic has been implemented in Isabelle/HOL [15].

It should also be mentioned that separation logic is related to other recent logics that embody a notion of separation, such as spatial logics or ambient logic [35, 36, 37, 38, 39, 40].

1.3 The Programming Language

The programming language we will use is a low-level imperative language — specifically, the simple imperative language originally axiomatized by Hoare [3, 4], extended with new commands for the manipulation of mutable shared data structures:

$$
\begin{array}{lr}
\langle \text{comm} \rangle ::= \cdots & \\
\quad | \ \langle \text{var} \rangle := \mathbf{cons}(\langle \text{exp} \rangle, \ldots, \langle \text{exp} \rangle) & \text{allocation} \\
\quad | \ \langle \text{var} \rangle := [\langle \text{exp} \rangle] & \text{lookup} \\
\quad | \ [\langle \text{exp} \rangle] := \langle \text{exp} \rangle & \text{mutation} \\
\quad | \ \mathbf{dispose}\ \langle \text{exp} \rangle & \text{deallocation}
\end{array}
$$

Memory managenent is explicit; there is no garbage collection. As we will see, any dereferencing of dangling addresses will cause a fault.

Semantically, we extend computational states to contain two components: a store (sometimes called a stack), mapping variables into values (as in the semantics of the unextended simple imperative language), and a heap, mapping addresses into values (and representing the mutable structures).

In the early versions of separation logic, integers, atoms, and addresses were regarded as distinct kinds of value, and heaps were mappings from finite sets of addresses to nonempty tuples of values:

$$
\begin{aligned}
\text{Values} &= \text{Integers} \cup \text{Atoms} \cup \text{Addresses} \\
&\quad \text{where Integers, Atoms, and Addresses are disjoint} \\
&\quad \mathbf{nil} \in \text{Atoms} \\
\text{Stores}_V &= V \to \text{Values} \\
\text{Heaps} &= \bigcup_{A \subseteq_{\text{fin}} \text{Addresses}} (A \to \text{Values}^+) \\
\text{States}_V &= \text{Stores}_V \times \text{Heaps} \\
&\quad \text{where } V \text{ is a finite set of variables.}
\end{aligned}
$$

(Actually, in most work using this kind of state, authors have imposed restricted formats on the records in the heap, to reflect the specific usage of the program they are specifying.)

To permit unrestricted address arithmetic, however, in the version of the logic used in most of this paper we will assume that all values are integers, an infinite number of which are addresses; we also assume that atoms are integers that are not addresses, and that heaps map addresses into single values:

$$\begin{aligned}
\text{Values} &= \text{Integers} \\
\text{Atoms} \cup \text{Addresses} &\subseteq \text{Integers} \\
&\quad \text{where Atoms and Addresses are disjoint} \\
\mathbf{nil} &\in \text{Atoms} \\
\text{Stores}_V &= V \to \text{Values} \\
\text{Heaps} &= \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \to \text{Values}) \\
\text{States}_V &= \text{Stores}_V \times \text{Heaps} \\
&\quad \text{where } V \text{ is a finite set of variables.}
\end{aligned}$$

(To permit unlimited allocation of records of arbitrary size, we require that, for all $n \geq 0$, the set of addresses must contain infinitely many consecutive sequences of length n. For instance, this will occur if only a finite number of positive integers are not addresses.)

Our intent is to capture the low-level character of machine language. One can think of the store as describing the contents of registers, and the heap as describing the contents of an addressable memory. This view is enhanced by assuming that each address is equipped with an "activity bit"; then the domain of the heap is the finite set of *active* addresses.

The semantics of ordinary and boolean expressions is the same as in the simple imperative language:

$$\begin{aligned}
[\![e \in \langle \text{exp} \rangle]\!]_{\text{exp}} &\in (\bigcup_{\substack{\text{fin} \\ V \supseteq \text{FV}(e)}} \text{Stores}_V) \to \text{Values} \\
[\![b \in \langle \text{boolexp} \rangle]\!]_{\text{bexp}} &\in \\
&\quad (\bigcup_{\substack{\text{fin} \\ V \supseteq \text{FV}(b)}} \text{Stores}_V) \to \{\mathbf{true}, \mathbf{false}\}
\end{aligned}$$

(where $\text{FV}(p)$ is the set of variables occurring free in the phrase p). In

particular, expressions do not depend upon the heap, so that they are always well-defined and never cause side-effects.

Thus expressions do not contain notations, such as **cons** or $[-]$, that refer to the heap; instead these notations are part of the structure of commands (and thus cannot be nested). It follows that none of the new heap-manipulating commands are instances of the simple assignment command $\langle \text{var} \rangle := \langle \text{exp} \rangle$ (even though we write the allocation, lookup, and mutation commands with the familiar operator $:=$). In fact, these commands will not obey Hoare's inference rule for assignment. However, since they alter the store at the variable v, we will say that the commands $v := \mathbf{cons}(\cdots)$ and $v := [e]$, as well as $v := e$ (but not $[v] := e$ or **dispose** v) *modify* v.

Our strict avoidance of side-effects in expressions will allow us to use them in assertions with the same freedom as in ordinary mathematics. This will substantially simplify the logic, at the expense of occasional complications in programs.

The semantics of the new commands is simple enough to be conveyed by example. If we begin with a state where the store maps the variables x and y into three and four, and the heap is empty, then the typical effect of each kind of heap-manipulating command is:

		Store :	$\mathsf{x}\colon 3,\ \mathsf{y}\colon 4$
		Heap :	empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;		$\Downarrow$
		Store :	$\mathsf{x}\colon 37,\ \mathsf{y}\colon 4$
		Heap :	$37\colon 1,\ 38\colon 2$
Lookup	$\mathsf{y} := [\mathsf{x}]$;		$\Downarrow$
		Store :	$\mathsf{x}\colon 37,\ \mathsf{y}\colon 1$
		Heap :	$37\colon 1,\ 38\colon 2$
Mutation	$[\mathsf{x} + 1] := 3$;		$\Downarrow$
		Store :	$\mathsf{x}\colon 37,\ \mathsf{y}\colon 1$
		Heap :	$37\colon 1,\ 38\colon 3$
Deallocation	$\mathbf{dispose}(\mathsf{x} + 1)$		$\Downarrow$
		Store :	$\mathsf{x}\colon 37,\ \mathsf{y}\colon 1$
		Heap :	$37\colon 1$

The allocation operation $\mathbf{cons}(e_1, \ldots, e_n)$ activates and initializes n cells in the heap. It is important to notice that, aside from the requirement that the addresses of these cells be consecutive and previously inactive, the choice of addresses is indeterminate.

The remaining operations, for mutation, lookup, and deallocation, all cause memory faults (denoted by the terminal configuration **abort**) if an inactive address is dereferenced or deallocated. For example:

		Store : $\mathsf{x}: 3,\ \mathsf{y}: 4$
		Heap : empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;	$\Downarrow$
		Store : $\mathsf{x}: 37,\ \mathsf{y}: 4$
		Heap : $37: 1,\ 38: 2$
Lookup	$\mathsf{y} := [\mathsf{x}]$;	$\Downarrow$
		Store : $\mathsf{x}: 37,\ \mathsf{y}: 1$
		Heap : $37: 1,\ 38: 2$
Mutation	$[\mathsf{x} + 2] := 3$;	$\Downarrow$
		abort

1.4 Assertions

As in Hoare logic, assertions describe states, but now states contain heaps as well as stores. Thus, in addition to the usual operations and quantifiers of predicate logic, we have four new forms of assertion that describe the heap:

- **emp** (empty heap)
  The heap is empty.

- $e \mapsto e'$ (singleton heap)
  The heap contains one cell, at address e with contents e'.

- $p_1 \ * \ p_2$ (separating conjunction)
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.

- $p_1 \mathrel{-\!\!*} p_2$ (separating implication)
  If the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

It is also useful to introduce abbreviations for asserting that an address e is active:

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'.\ e \mapsto x' \quad \text{where } x' \text{ not free in } e,$$

that e points to e' somewhere in the heap:

$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' \ * \ \mathbf{true},$$

and that e points to a record with several fields:

$$e \mapsto e_1, \ldots, e_n \stackrel{\text{def}}{=} e \mapsto e_1 \ * \ \cdots \ * \ e + n - 1 \mapsto e_n$$

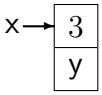
$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 \ * \ \cdots \ * \ e + n - 1 \hookrightarrow e_n$$

$$\text{iff } e \mapsto e_1, \ldots, e_n \ * \ \mathbf{true}.$$

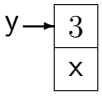
Notice that assertions of the form $e \mapsto e'$, $e \mapsto -$, and $e \mapsto e_1, \ldots, e_n$ determine the extent (i.e., domain) of the heap they describe, while those of the form $e \hookrightarrow e'$ and $e \hookrightarrow e_1, \ldots, e_n$ do not. (Technically, the former are said to be *precise* assertions. A precise definition of precise assertions will be given in Section 2.3.3.)

By using $\mapsto$, $\hookrightarrow$, and both separating and ordinary conjunction, it is easy to describe simple sharing patterns concisely. For instance:

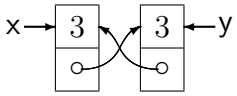
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β, α is an address, and the heap maps α into 3 and $\alpha + 1$ into β).



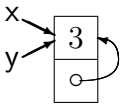
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x.



3. $\mathsf{x} \mapsto 3, \mathsf{y} \ * \ \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for separate parts of the heap.



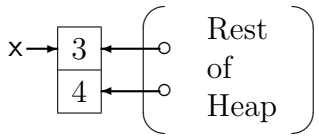
4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

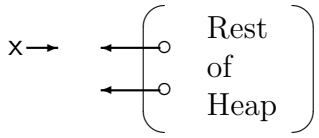
Separating implication is somewhat more subtle, but is illustrated by the following example (due to O'Hearn): Suppose the assertion p asserts various conditions about the store and heap, including that the store maps x into the address of a record containing 3 and 4:

Store : $\mathsf{x}: \alpha, \ldots$
Heap : $\alpha: 3, \alpha + 1: 4$, Rest of Heap



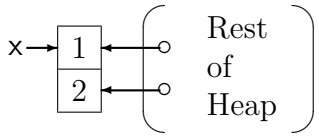
Then $(\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p$ asserts that, if one were to add to the current heap a disjoint heap consisting of a record at address x containing 3 and 4, then the resulting heap would satisfy p. In other words, the current heap is like that described by p, except that the record is missing:

Store : $\mathsf{x}: \alpha, \ldots$
Heap : Rest of Heap, as above



Moreover, $\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)$ asserts that the heap consists of a record at x containing 1 and 2, plus a separate part as above:

Store : $\mathsf{x}: \alpha, \ldots$
Heap : $\alpha: 1, \alpha + 1: 2,$
Rest of Heap, as above



This example suggests that $\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)$ describes a state that would be changed by the mutation operations $[\mathsf{x}] := 3$ and $[\mathsf{x} + 1] := 4$ into a state satisfying p. In fact, we will find that

$$\{\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\} \ [\mathsf{x}] := 3 \ ; [\mathsf{x} + 1] := 4 \ \{p\}$$

is a valid specification (i.e., Hoare triple) in separation logic — as is the more general specification

$$\{\mathsf{x} \mapsto -, - \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\} \ [\mathsf{x}] := 3 \ ; [\mathsf{x} + 1] := 4 \ \{p\}.$$

The inference rules for predicate calculus (not involving the new operators we have introduced) remain sound in this enriched setting. Additional axiom schemata for separating conjunction include commutative and associative

laws, the fact that **emp** is a neutral element, and various distributive and semidistributive laws:

$$\begin{aligned}
p_1 \ * \ p_2 &\Leftrightarrow p_2 \ * \ p_1 \\
(p_1 \ * \ p_2) \ * \ p_3 &\Leftrightarrow p_1 \ * \ (p_2 \ * \ p_3) \\
p \ * \ \mathbf{emp} &\Leftrightarrow p \\
(p_1 \vee p_2) \ * \ q &\Leftrightarrow (p_1 \ * \ q) \vee (p_2 \ * \ q) \\
(p_1 \wedge p_2) \ * \ q &\Rightarrow (p_1 \ * \ q) \wedge (p_2 \ * \ q) \\
(\exists x.\ p_1) \ * \ p_2 &\Leftrightarrow \exists x.\ (p_1 \ * \ p_2) \quad \text{when } x \text{ not free in } p_2 \\
(\forall x.\ p_1) \ * \ p_2 &\Rightarrow \forall x.\ (p_1 \ * \ p_2) \quad \text{when } x \text{ not free in } p_2
\end{aligned}$$

There is also an inference rule showing that separating conjunction is monotone with respect to implication:

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 \ * \ q_1 \Rightarrow p_2 \ * \ q_2} \quad \text{(monotonicity)}$$

and two further rules capturing the adjunctive relationship between separating conjunction and separating implication:

$$\frac{p_1 \ * \ p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 \mathrel{-\!\!*} p_3)} \quad \text{(currying)} \qquad \frac{p_1 \Rightarrow (p_2 \mathrel{-\!\!*} p_3)}{p_1 \ * \ p_2 \Rightarrow p_3.} \quad \text{(decurrying)}$$

On the other hand, there are two rules that one might expect to hold for an operation called "conjunction" that in fact fail:

$$\begin{aligned}
p &\Rightarrow p \ * \ p && \text{(Contraction — unsound)} \\
p \ * \ q &\Rightarrow p && \text{(Weakening — unsound)}
\end{aligned}$$

A counterexample to both of these axiom schemata is provided by taking p to be $\mathsf{x} \mapsto 1$ and q to be $\mathsf{y} \mapsto 2$; then p holds for a certain single-field heap while $p \ * \ p$ holds for no heap, and $p \ * \ q$ holds for a certain two-field heap while p holds for no two-field heap. (Thus separation logic is a substructural logic.)

Finally, we give axiom schemata for the predicate $\mapsto$. (Regrettably, these are far from complete.)

$$e_1 \mapsto e_1' \wedge e_2 \mapsto e_2' \Leftrightarrow e_1 \mapsto e_1' \wedge e_1 = e_2 \wedge e_1' = e_2'$$
$$e_1 \hookrightarrow e_1' \ * \ e_2 \hookrightarrow e_2' \Rightarrow e_1 \neq e_2$$
$$\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -)$$
$$(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).$$

1.5 Specifications and their Inference Rules

While assertions describe states, specifications describe commands. In specification logic, specifications are Hoare triples, which come in two flavors:

⟨specification⟩ ::=

{⟨assertion⟩} ⟨command⟩ {⟨assertion⟩}	(partial correctness)
| [⟨assertion⟩] ⟨command⟩ [⟨assertion⟩]	(total correctness)

In both flavors, the initial assertion is called the *precondition* (or sometimes the *precedent*), and the final assertion is called the *postcondition* (or sometimes the *consequent*).

The *partial correctness specification* $\{p\}\ c\ \{q\}$ is true iff, starting in any state in which p holds,

- No execution of c aborts, and
- When some execution of c terminates in a final state, then q holds in the final state.

The *total correctness specification* $[\,p\,]\ c\ [\,q\,]$ (which we will use much less often) is true iff, starting in any state in which p holds,

- No execution of c aborts, and
- Every execution of c terminates, and
- When some execution of c terminates in a final state, then q holds in the final state.

These forms of specification are so similar to those of Hoare logic that it is important to note the differences. Our specifications are implicitly quantified over both stores and heaps, and also (since allocation is indeterminate) over all possible executions. Moreover, any execution (starting in a state satisfying p) that gives a memory fault falsifies both partial and total specifications.

The last point goes to the heart of separation logic. As O'Hearn [5] paraphrased Milner, "Well-specified programs don't go wrong." As a consequence, during the execution of a program that has been proved to meet some specification (assuming that the program is only executed in initial states satisfying the precondition of the specification), it is unnecessary to check for memory faults, or even to equip heap cells with activity bits.

In fact, it is not the implementor's responsibility to detect memory faults. It is the programmer's responsibility to avoid them — and separation logic is a tool for this purpose. Indeed, according to the logic, the implementor is free to implement memory faults however he wishes, since nothing can be proved that might gainsay him.

Roughly speaking, the fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. For example, during an execution of

$$c_0 \,;\, \mathsf{x} := \mathbf{cons}(1,2) \,;\, c_1 \,;\, [\mathsf{x}+2] := 7,$$

no allocation performed by the subcommand c_0 or c_1 can be guaranteed to allocate the location $\mathsf{x}+2$; thus as long as c_0 and c_1 terminate and c_1 does not modify x, there is a possibility that the execution will abort. It follows that there is no postcondition that makes the specification

$$\{\mathbf{true}\}\ c_0 \,;\, \mathsf{x} := \mathbf{cons}(1,2) \,;\, c_1 \,;\, [\mathsf{x}+2] := 7\ \{?\}$$

valid.

Sometimes, however, the notion of record boundaries dissolves, as in the following valid (and provable) specification of a program that tries to form a two-field record by gluing together two one-field records:

$$\begin{array}{l}
\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\mathbf{if}\ \mathsf{y} = \mathsf{x}+1\ \mathbf{then\ skip\ else} \\
\quad \mathbf{if}\ \mathsf{x} = \mathsf{y}+1\ \mathbf{then}\ \mathsf{x} := \mathsf{y}\ \mathbf{else} \\
\qquad (\mathbf{dispose}\ \mathsf{x} \,;\, \mathbf{dispose}\ \mathsf{y} \,;\, \mathsf{x} := \mathbf{cons}(1,2)) \\
\{\mathsf{x} \mapsto -, -\}.
\end{array} \tag{1.5}$$

It is evident that such a program goes well beyond the discipline imposed by type systems for mutable data structures.

In our new setting, the command-specific inference rules of Hoare logic remain sound, as do such structural rules as

- Strengthening Precedent

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

- Weakening Consequent

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

- Existential Quantification (Ghost Variable Elimination)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

  where v is not free in c.

- Conjunction

$$\frac{\{p\}\ c\ \{q_1\} \qquad \{p\}\ c\ \{q_2\}}{\{p\}\ c\ \{q_1 \wedge q_2\}.}$$

- Substitution

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

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

(All of the inference rules presented in this section are the same for partial and total correctness.)

An exception is what is sometimes called the "rule of constancy" [27, Section 3.3.5; 28, Section 3.5]:

$$\frac{\{p\}\ c\ \{q\}}{\{p \wedge r\}\ c\ \{q \wedge r\},} \quad \text{(unsound)}$$

where no variable occurring free in r is modified by c. It has long been understood that this rule is vital for scalability, since it permits one to extend a "local" specification of c, involving only the variables actually used by that command, by adding arbitrary predicates about variables that are not modified by c and will therefore be preserved by its execution.

Surprisingly, however, the rule of constancy becomes unsound when one moves from traditional Hoare logic to separation logic. For example, the conclusion of the instance

$$\frac{\{\mathsf{x} \mapsto -\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4\}}{\{\mathsf{x} \mapsto - \wedge \mathsf{y} \mapsto 3\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4 \wedge \mathsf{y} \mapsto 3\}}$$

is not valid, since its precondition does not preclude the case $\mathsf{x} = \mathsf{y}$, where aliasing will falsify $\mathsf{y} \mapsto 3$ when the mutation command is executed.

O'Hearn realized, however, that the ability to extend local specifications can be regained at a deeper level by using separating conjunction. In place of the rule of constancy, he proposed the *frame rule*:

- Frame Rule
$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$
where no variable occurring free in r is modified by c.

By using the frame rule, one can extend a local specification, involving only the variables *and heap cells* that may actually be used by c (which O'Hearn calls the *footprint* of c), by adding arbitrary predicates about variables and heap cells that are not modified or mutated by c. Thus, the frame rule is the key to "local reasoning" about the heap:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged [8].

In any valid specification $\{p\}\ c\ \{q\}$, p must assert that the heap contains every cell in the footprint of c (except for cells that are freshly allocated by c); "locality" is the converse implication that every cell asserted to be contained in the heap belongs to the footprint. The role of the frame rule is to infer from a local specification of a command the more global specification appropriate to the possibly larger footprint of an enclosing command.

Beyond the rules of Hoare logic and the frame rule, there are inference rules for each of the new heap-manipulating commands. Indeed, for each of these commands, we can give three kinds of rules: local, global, and backward-reasoning.

For mutation, for example, the simplest rule is the local rule:

- Mutation (local)
$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}},$$

which specifies the effect of mutation on the single cell being mutated. From this, one can use the frame rule to derive a global rule:

- Mutation (global)
$$\frac{}{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}},$$

which also specifies that anything in the heap beyond the cell being mutated is left unchanged by the mutation. (One can rederive the local rule from the global one by taking r to be **emp**.)

Beyond these forms, there is also:

- Mutation (backwards reasoning)
$$\frac{}{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}},$$

which is called a *backward reasoning* rule since, by substituting for p, one can find a precondition for any postcondition. [5].

A similar development works for deallocation, except that the global form is itself suitable for backward reasoning:

- Deallocation (local)
$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}.$$

- Deallocation (global, backwards reasoning)
$$\frac{}{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}}.$$

In the same way, one can give equivalent local and global rules for allocation commands in the *nonoverwriting* case where the old value of the variable being modified plays no role. Here we abbreviate $e_1, \ldots, e_n$ by $\overline{e}$.

- Allocation (nonoverwriting, local)

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

  where v is not free in $\overline{e}$.

- Allocation (nonoverwriting, global)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\}},$$

  where v is not free in $\overline{e}$ or r.

Of course, we also need more general rules for allocation commands $v := \mathbf{cons}(\overline{e})$, where v occurs in $\overline{e}$ or the precondition, as well as a backward-reasoning rule for allocation, and rules for lookup. Since all of these rules are more complicated than those given above (largely because most of them contain quantifiers), we postpone them to Section 3.7 (where we will also show that the different forms of rules for each command are interderivable).

As a simple illustration of separation logic, the following is an annotated specification of the command (1.5) that tries to glue together adjacent records:

$$\begin{array}{l}
\{\mathsf{x} \mapsto - \ *\ \mathsf{y} \mapsto -\} \\
\mathbf{if}\ \mathsf{y} = \mathsf{x} + 1\ \mathbf{then} \\
\quad \{\mathsf{x} \mapsto -, -\} \\
\quad \mathbf{skip} \\
\mathbf{else\ if}\ \mathsf{x} = \mathsf{y} + 1\ \mathbf{then} \\
\quad \{\mathsf{y} \mapsto -, -\} \\
\quad \mathsf{x} := \mathsf{y} \\
\mathbf{else} \\
\quad \big(\{\mathsf{x} \mapsto - \ *\ \mathsf{y} \mapsto -\} \\
\quad \mathbf{dispose}\ \mathsf{x}\,; \\
\quad \{\mathsf{y} \mapsto -\} \\
\quad \mathbf{dispose}\ \mathsf{y}\,; \\
\quad \{\mathbf{emp}\} \\
\quad \mathsf{x} := \mathbf{cons}(1, 2)\big) \\
\{\mathsf{x} \mapsto -, -\}.
\end{array}$$

We will make the concept of an annotated specification — as a user-friendly form for presenting proof of specifications — rigorous in Sections 3.3 and 3.6. For the present, one can think of the intermediate assertions as comments that must be true whenever control passes through them (assuming the initial assertion is true when the program begins execution), and that must also ensure the correct functioning of the rest of the program being executed.

A second example describes a command that uses allocation and mutation to construct a two-element cyclic structure containing relative addresses:

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a}) \; ; \\
\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\} \\
\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b}) \; ; \\
\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\} \\
\{(\mathsf{x} \mapsto \mathsf{a}, -) \; * \; (\mathsf{y} \mapsto \mathsf{b}, -)\} \\
[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x} \; ; \\
\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, -)\} \\
[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y} \; ; \\
\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\} \\
\{\exists \mathsf{o}. \; (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \; * \; (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b}, \; -\mathsf{o})\}.
\end{array}
$$

1.6 Lists

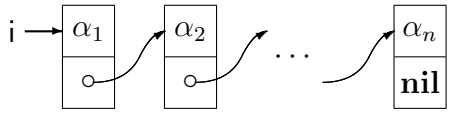
To specify a program adequately, it is usually necessary to describe more than the form of its structures or the sharing patterns between them; one must relate the states of the program to the abstract values that they denote. For instance, to specify the list-reversal program in Section 1.1, it would hardly be enough to say that "If i is a list before execution, then j will be a list afterwards". One needs to say that "If i is a list representing the sequence α before execution, then afterwards j will be a list representing the sequence that is the reflection of α."

To do so in general, it is necessary to define the set of abstract values (sequences, in this case), along with their primitive operations, and then to define predicates on the abstract values by structural induction. Since these

kinds of definition are standard, we will treat them less formally than the novel aspects of our logic.

Sequences and their primitive operations are an easy first example since they are a standard and well-understood mathematical concept, so that we can omit their definition. To denote the primitive operations, we write ϵ for the empty sequence, $\alpha \cdot \beta$ for the composition of α followed by β, $\alpha^\dagger$ for the reflection of α, and α_i for the ith component of α.

The simplest list structure for representing sequences is the *singly-linked* list. To describe this representation, we write $\mathsf{list}\,\alpha\,\mathsf{i}$ when i is a list representing the sequence α:



It is straightforward to define this predicate by induction on the structure of α:

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

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

(where ϵ denotes the empty sequence and $\alpha{\cdot}\beta$ denotes the concatenation of α followed by β), and to derive a test whether the list represents an empty sequence:

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

Then the following is an annotated specification of the program for reversing

a list:

$$
\begin{array}{l}
\{\mathsf{list}\ \alpha_0\ \mathsf{i}\} \\
\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathbf{nil} = \mathbf{nil})\} \\
\mathsf{j} := \mathbf{nil}\ ; \\
\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ (\mathbf{emp} \wedge \mathsf{j} = \mathbf{nil})\} \\
\{\mathsf{list}\ \alpha_0\ \mathsf{i}\ *\ \mathsf{list}\ \epsilon\ \mathsf{j}\} \\
\{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\} \\
\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do} \\
\quad \big(\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ (\mathsf{a}\cdot\alpha)\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\} \\
\quad\ \{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\} \\
\quad\ \mathsf{k} := [\mathsf{i}+1]\ ; \\
\quad\ \{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\} \\
\quad\ [\mathsf{i}+1] := \mathsf{j}\ ; \\
\quad\ \{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\} \\
\quad\ \{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ (\mathsf{a}\cdot\beta)\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \mathsf{a} \cdot \beta\} \\
\quad\ \{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\} \\
\quad\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k} \\
\quad\ \{\exists \alpha, \beta.\ (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}\big) \\
\{\exists \alpha, \beta.\ \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon\} \\
\{\mathsf{list}\ \alpha_0^\dagger\ \mathsf{j}\}
\end{array}
$$

Within the assertions here, Greek letters are used as variables denoting sequences. More formally, we have extended the state to map Greek variables into sequences as well as sans serif variables into integers.

1.7 Trees and Dags

When we move from list to tree structures, the possible patterns of sharing within the structures become richer.

At the outset, we face a problem of nomenclature: Words such as "tree" and "graph" are often used to describe a variety of abstract structures, as

well as particular ways of representing such structures. Here we will focus on a particular kind of abstract value called an "S-expression" in the LISP community. The set S-exps of these values is the least set such that

$$\tau \in \text{S-exps iff } \tau \in \text{Atoms}$$
$$\text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps.}$$

(Of course, this is just a particular, and very simple, initial algebra — as is "sequence". We could take carriers of any lawless many-sorted initial algebra to be our abstract data, but this would complicate our exposition while adding little of interest.)

For clarity, it is vital to maintain the distinction between abstract values and their representations. Thus, we will call abstract values "S-expressions", while calling representations without sharing "trees", and representations with sharing but no cycles "dags" (for "directed acyclic graphs").

We write $\mathsf{tree}\,\tau\,(i)$ (or $\mathsf{dag}\,\tau\,(i)$) to indicate that i is the root of a tree (or dag) representing the S-expression τ. Both predicates are defined by induction on the structure of τ:

$$\mathsf{tree}\,a\,(i) \text{ iff } \mathbf{emp} \wedge i = a$$
$$\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff } \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ *\ \mathsf{tree}\,\tau_1\,(i_1) \ *\ \mathsf{tree}\,\tau_2\,(i_2)$$

$$\mathsf{dag}\,a\,(i) \text{ iff } i = a$$
$$\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff } k\exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ *\ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)).$$

(In Sections 5.1 and 5.2, we will see that $\mathsf{tree}\,\tau\,(i)$ is a precise assertion, so that it describes a heap containing a tree-representation of τ and nothing else, while $\mathsf{dag}\,\tau\,(i)$ is an *intuitionistic* assertion, describing sa heap that may contain extra space as well as a tree-representation of τ.)

1.8 Arrays and the Iterated Separating Conjunction

It is straightforward to extend our programming language to include heap-allocated one-dimensional arrays, by introducing an allocation command where the number of consecutive heap cells to be allocated is specified by an operand. It is simplest to leave the initial values of these cells indeterminate.

We will use the syntax

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

where v:=**allocate** e will be a command that allocates e consecutive locations and makes the first of these locations the value of the variable v. For instance:

$$\mathsf{x} := \textbf{allocate}\ \mathsf{y} \qquad \begin{array}{ll} \text{Store}: & \mathsf{x}: 3,\ \mathsf{y}: 4 \\ \text{Heap}: & \text{empty} \\ & \Downarrow \\ \text{Store}: & \mathsf{x}: 37,\ \mathsf{y}: 4 \\ \text{Heap}: & 37: -,\ 38: -, 39: -,\ 40: - \end{array}$$

To describe such arrays, it is helpful to extend the concept of separating conjunction to a construct that iterates over a finite contiguous set of integers. We use the syntax

$$\langle \text{assert} \rangle ::= \cdots \mid \bigodot_{\langle \text{var} \rangle = \langle \text{exp} \rangle}^{\langle \text{exp} \rangle} \langle \text{assert} \rangle$$

Roughly speaking, $\bigodot_{v=e}^{e'} p$ bears the same relation to $*$ that $\forall_{v=e}^{e'} p$ bears to $\wedge$. More precisely, let I be the contiguous set $\{\, v \mid e \le v \le e' \,\}$ of integers between the values of e and e'. Then $\bigodot_{v=e}^{e'} p(v)$ is true iff the heap can be partitioned into a family of disjoint subheaps, indexed by I, such that $p(v)$ is true for the vth subheap.

Then array allocation is described by the following inference rule:

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

where v does not occur free in r or e.

A simple illustration of the iterated separating conjunction is the use of an array as a cyclic buffer. We assume that an n-element array has been allocated at address l, e.g., by $\mathsf{l} :=$ **allocate** n, and we use the variables

m	number of active elements
i	address of first active element
j	address of first inactive element.

Then when the buffer contains a sequence α, it should satisfy the assertion

$$\begin{array}{l} 0 \le \mathsf{m} \le \mathsf{n}\ \wedge\ \mathsf{l} \le \mathsf{i} < \mathsf{l} + \mathsf{n}\ \wedge\ \mathsf{l} \le \mathsf{j} < \mathsf{l} + \mathsf{n}\ \wedge \\ \mathsf{j} = \mathsf{i} \oplus \mathsf{m}\ \wedge\ \mathsf{m} = \#\alpha\ \wedge \\ ((\bigodot_{\mathsf{k}=0}^{\mathsf{m}-1} \mathsf{i} \oplus \mathsf{k} \mapsto \alpha_{\mathsf{k}+1})\ *\ (\bigodot_{\mathsf{k}=0}^{\mathsf{n}-\mathsf{m}-1} \mathsf{j} \oplus \mathsf{k} \mapsto -)), \end{array}$$

where $x \oplus y = x + y$ modulo n, and $\mathsf{l} \le x \oplus y < \mathsf{l} + \mathsf{n}$.

1.9 Proving the Schorr-Waite Algorithm

One of the most ambitious applications of separation logic has been Yang's proof of the Schorr-Waite algorithm for marking structures that contain sharing and cycles [12, 10]. This proof uses the older form of classical separation logic [5] in which address arithmetic is forbidden and the heap maps addresses into multifield records — each containing, in this case, two address fields and two boolean fields.

Since addresses refer to entire records with identical number and types of fields, it is easy to assert that the record at x has been allocated:

$$\mathsf{allocated}(\mathsf{x}) \stackrel{\text{def}}{=} \mathsf{x} \hookrightarrow -, -, -, -,$$

that all records in the heap are marked:

$$\mathsf{markedR} \stackrel{\text{def}}{=} \forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \mathsf{x} \hookrightarrow -, -, -, \mathbf{true},$$

that x is not a dangling address:

$$\mathsf{noDangling}(\mathsf{x}) \stackrel{\text{def}}{=} (\mathsf{x} = \mathbf{nil}) \vee \mathsf{allocated}(\mathsf{x}),$$

or that no record in the heap contains a dangling address:

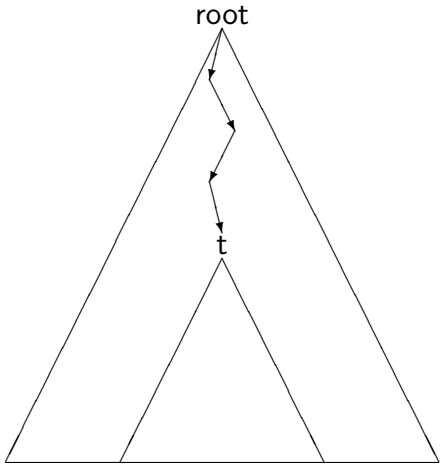
$$\begin{aligned}\mathsf{noDanglingR} \stackrel{\text{def}}{=} \forall \mathsf{x}, \mathsf{l}, \mathsf{r}.\ (\mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, -, -) \Rightarrow \\ \mathsf{noDangling}(\mathsf{l}) \wedge \mathsf{noDangling}(\mathsf{r}).\end{aligned}$$

The heap described by the main invariant of the program is the footprint of the entire algorithm, which is exactly the structure that is reachable from the address root. The invariant itself is:

$$\begin{aligned}&\mathsf{noDanglingR} \wedge \mathsf{noDangling}(\mathsf{t}) \wedge \mathsf{noDangling}(\mathsf{p}) \wedge \\ &\big(\mathsf{listMarkedNodesR}(\mathsf{stack}, \mathsf{p})\ * \\ &\qquad (\mathsf{restoredListR}(\mathsf{stack}, \mathsf{t}) \mathrel{-\!\!*} \mathsf{spansR}(\mathsf{STree}, \mathsf{root}))\big) \wedge \\ &\big(\mathsf{markedR}\ *\ \big(\mathsf{unmarkedR} \wedge \big(\forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \\ &\qquad (\mathsf{reach}(\mathsf{t}, \mathsf{x}) \vee \mathsf{reachRightChildInList}(\mathsf{stack}, \mathsf{x}))\big)\big)\big).\end{aligned}$$

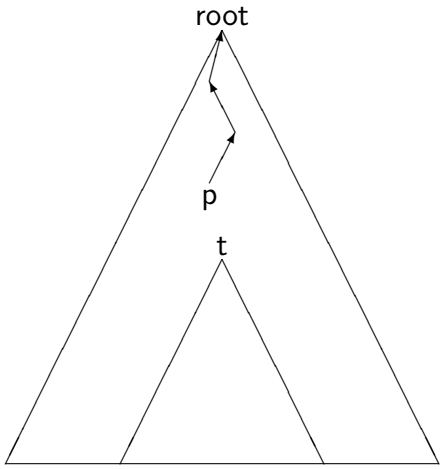
At the point in the computation described by this invariant, the value of the variable t indicates the current subheap that is about to be scanned. At the

beginning of the computation, there is a path called the *spine* from root to this value:



The assertion restoredListR(stack, t) describes this state of the spine; the abstract variable stack encodes the information contained in the spine.

At the instant described by the invariant, however, the links in the spine are reversed:



This reversed state of the spine, again containing the information encoded by stack, is described by the assertion listMarkedNodesR(stack, p).

The assertion spansR(STree, root), which also occurs in the precondition of the algorithm, asserts that the abstract structure STree is a spanning tree of the heap. Thus, the second and third lines of the invariant use separating implication elegantly to assert that, if the spine is correctly restored, then the heap will have the same spanning tree as it had initially. (In fact, the proof goes through if spansR(STree, root) is any predicate about the heap

that is independent of the boolean fields in the records; spanning trees are used only because they are sufficent to determine the heap, except for the boolean fields.) To the author's knowledge, this part of the invariant is the earliest conceptual use of separating implication in a real proof of a program (as opposed to its formal use in expressing backward-reasoning rules and weakest preconditions).

In the rest of the invariant, the heap is partitioned into marked and unmarked records, and it is asserted that every active unmarked record can be reached from the variable `t` or from certain fields in the spine. However, since this assertion lies within the right operand of the separating conjunction that separates marked and unmarked notes, the paths by which the unmarked records are reached must consist of unmarked records. Anyone (such as the author [41, Section 5.1]) who has tried to verify this kind of graph traversal, even informally, will appreciate the extraordinary succinctness of the last two lines of Yang's invariant.

1.10 Shared-Variable Concurrency

O'Hearn has extended separation logic to reason about shared-variable concurrency, drawing upon early ideas of Hoare [42] and Owicki and Gries [43].

For the simplest case, where concurrency is unconstrained by any kind of synchronization mechanism, Hoare had given the straightforward rule:

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ c_1 \parallel c_2\ \{q_1 \wedge q_2\},}$$

when the free variables of p_1, c_1, and q_1 are not modified by c_2, and vice-versa.

Unfortunately, this rule fails in separation logic since, even though the side condition prohibits the processes from interfering via assignments to variables, they permit interference via mutations in the heap. O'Hearn realized that the rule could be saved by replacing the ordinary conjunctions by separating conjunctions, which separated the heap into parts that can only be mutated by a single process:

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1\ *\ p_2\}\ c_1 \parallel c_2\ \{q_1\ *\ q_2\}}$$

(with the same side condition as above).

Things became far less straightforward, however, when synchronization was introduced. Hoare had investigated conditional critical regions, keyed to "resources", which were disjoint collections of variables. His crucial idea was that there should be an invariant associated with each resource, such that when one entered a critical region keyed to a resource, one could assume that the invariant was true, but when one left the region, the invariant must be restored.

O'Hearn was able to generalize these ideas so that both processes and resources could "own" portions of the heap, and this ownership could move among processes and resources dynamically as the processes entered and left critical regions.

As a simple example, consider two processes that share a buffer consisting of a single **cons**-cell. At the level of the processes, there are simply two procedures: $\mathsf{put}(\mathsf{x})$, which accepts a **cons**-cell and makes it disappear, and $\mathsf{get}(\mathsf{y})$, which makes a **cons**-cell appear. The first process allocates a **cons**-cell and gives it to $\mathsf{put}(\mathsf{x})$; the second process obtains a **cons**-cell from $\mathsf{get}(\mathsf{y})$, uses it, and deallocates it:

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\{\mathbf{emp} * \mathbf{emp}\} \\
\begin{array}{lcl}
\{\mathbf{emp}\} & & \{\mathbf{emp}\} \\
\mathsf{x} := \mathbf{cons}(\ldots,\ldots)\,; & & \mathsf{get}(\mathsf{y})\,; \\
\{\mathsf{x} \mapsto -,-\} & \| & \{\mathsf{y} \mapsto -,-\} \\
\mathsf{put}(\mathsf{x})\,; & & \text{“Use } \mathsf{y}\text{”}\,; \\
\{\mathbf{emp}\} & & \{\mathsf{y} \mapsto -,-\} \\
 & & \mathbf{dispose}\,\mathsf{y}\,; \\
 & & \{\mathbf{emp}\}
\end{array} \\
\{\mathbf{emp} * \mathbf{emp}\} \\
\{\mathbf{emp}\}
\end{array}
$$

Behind the scenes, however, there is a resource buf that implements a small buffer that can hold a single **cons**-cell. Associated with this resource are a boolean variable full, which indicates whether the buffer currently holds a cell, and an integer variable c that points to the cell when full is true. Then $\mathsf{put}(\mathsf{x})$ is implemented by a critical region that checks the buffer is empty and then fills it with x, and $\mathsf{get}(\mathsf{y})$ is implemented by a conditional critical

regions that checks the buffer is full and then empties it into y:

$$\mathsf{put}(\mathsf{x}) = \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \neg\mathsf{full}\ \mathbf{do}\,(\mathsf{c} := \mathsf{x}\,;\mathsf{full} := \mathbf{true})$$

$$\mathsf{get}(\mathsf{y}) = \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \mathsf{full}\ \mathbf{do}\,(\mathsf{y} := \mathsf{c}\,;\mathsf{full} := \mathbf{false})$$

Associated with the resource buf is an invariant:

$$\mathsf{R} \stackrel{\text{def}}{=} (\mathsf{full} \wedge \mathsf{c} \mapsto -,-) \vee (\neg\mathsf{full} \wedge \mathbf{emp}).$$

The effect of O'Hearn's inference rule for critical regions is that the resource invariant is used explicitly to reason about the body of a critical region, but is hidden outside of the critical region:

$$\begin{array}{rl}
 & \{\mathsf{x} \mapsto -,-\} \\
\mathsf{put}(\mathsf{x}) = & \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \neg\mathsf{full}\ \mathbf{do}\,\big(\\
 & \quad \{(R \ast \mathsf{x} \mapsto -,-) \wedge \neg\mathsf{full}\} \\
 & \quad \{\mathbf{emp} \ast \mathsf{x} \mapsto -,-\} \\
 & \quad \{\mathsf{x} \mapsto -,-\} \\
 & \quad \mathsf{c} := \mathsf{x}\,;\mathsf{full} := \mathbf{true} \\
 & \quad \{\mathsf{full} \wedge \mathsf{c} \mapsto -,-\} \\
 & \quad \{R\} \\
 & \quad \{R \ast \mathbf{emp}\}\big) \\
 & \{\mathbf{emp}\} \\
 & \\
 & \{\mathbf{emp}\} \\
\mathsf{get}(\mathsf{y}) = & \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \mathsf{full}\ \mathbf{do}\,\big(\\
 & \quad \{(R \ast \mathbf{emp}) \wedge \mathsf{full}\} \\
 & \quad \{\mathsf{c} \mapsto -,- \ast \mathbf{emp}\} \\
 & \quad \{\mathsf{c} \mapsto -,-\} \\
 & \quad \mathsf{y} := \mathsf{c}\,;\mathsf{full} := \mathbf{false} \\
 & \quad \{\neg\mathsf{full} \wedge \mathsf{y} \mapsto -,-\} \\
 & \quad \{(\neg\mathsf{full} \wedge \mathbf{emp}) \ast \mathsf{y} \mapsto -,-\} \\
 & \quad \{R \ast \mathsf{y} \mapsto -,-\}\big) \\
 & \{\mathsf{y} \mapsto -,-\}
\end{array}$$

On the other hand, the resource invariant reappears outside the declaration of the resource, indicating that it must be initialized beforehand, and will remain true afterwards:

$$
\begin{array}{l}
\{R * \mathbf{emp}\} \\
\textbf{resource } \mathsf{buf} \textbf{ in} \\
\qquad\qquad \{\mathbf{emp}\} \\
\qquad\quad \{\mathbf{emp} * \mathbf{emp}\} \\
\qquad\quad \vdots \quad \| \quad \vdots \\
\qquad\quad \{\mathbf{emp} * \mathbf{emp}\} \\
\qquad\qquad \{\mathbf{emp}\} \\
\{R * \mathbf{emp}\}
\end{array}
$$

1.11 Fractional Permissions

Especially in concurrent programming, one would like to extend separation logic to permit *passivity*, i.e., to allow processes or other subprograms that are otherwise restricted to separate storage to share access to read-only variables. R. Bornat [32] has opened this possibility by introducing the concept of permissions, originally devised by John Boyland [31].

The basic idea is to associate a fractional real number called a *permission* with the $\mapsto$ relation. We write $e \overset{z}{\mapsto} e'$, where z is a real number such that $0 < z \leq 1$, to indicate that e points to e' with permission z. Then $e \overset{1}{\mapsto} e'$ has the same meaning as $e \mapsto e'$, so that a permission of one allows all operations, but when $z < 1$ only lookup operations are allowed.

This idea is formalized by a conservation law for permissions:

$$e \overset{z}{\mapsto} e' \ * \ e \overset{z'}{\mapsto} e' \text{ iff } e \overset{z+z'}{\longmapsto} e',$$

along with local axiom schemata for each of the heap-manipulating operations:

$$
\begin{array}{c}
\{\mathbf{emp}\} v := \mathbf{cons}(e_1, \ldots, e_n) \{e \overset{1}{\mapsto} e_1, \ldots, e_n\} \\
\{e \overset{1}{\mapsto} -\} \mathbf{dispose}(e) \{\mathbf{emp}\} \\
\{e \overset{1}{\mapsto} -\}[e] := e' \{e \overset{1}{\mapsto} e'\} \\
\{e \overset{z}{\mapsto} e'\} v := [e] \{e \overset{z}{\mapsto} e' \wedge v = e\},
\end{array}
$$

with appropriate restrictions on variable occurrences.

Chapter 2

Assertions

In this chapter, we give a more detailed exposition of the assertions of separation logic: their meaning, illustrations of their usage, inference rules, and several classes of assertions with special properties.

First, we discuss some general properties that are shared among assertions and other phrases occurring in both our programming language and its logic. In all cases, variable binding behaves in the standard manner. In expressions there are no binding constructions, in assertions quantifiers are binding constructions, and in commands declarations are binding constructions. In each case, the scope of the variable being bound is the immediately following subphrase, except that in declarations of the form **newvar** $v = e$ **in** c, or iterated separating conjunctions of the form $\bigodot_{v=e_0}^{e_1} p$ (to be introduced in Section 6.1), the initialization e and the bounds e_0 and e_1 are not in the scope of the binder v.

We write $\mathrm{FV}(p)$ for the set of variables occurring free in p, which is defined in the standard way. The meaning of any phrase is independent of the value of those variables that do not occur free in the phrase.

Substitution is also defined in the standard way. We begin by considering *total* substitutions that act upon all the free variables of a phrase: For any phrase p such that $\mathrm{FV}(p) \subseteq \{v_1, \ldots, v_n\}$, we write

$$p/v_1 \to e_1, \ldots, v_n \to e_n$$

to denote the phrase obtained from p by simultaneously substituting each expression e_i for the variable v_i, (When there are bound variables in p, they will be renamed to avoid capture.)

When expressions are substituted for variables in an expression or assertion p, the effect mimics a change of the store. In the specific case where p is an expression:

Proposition 1 *(Total Substitution Law for Expressions) Let δ abbreviate the substitution*

$$v_1 \to e_1, \ldots, v_n \to e_n,$$

let s be a store such that $\mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \subseteq \mathrm{dom}\, s$*, and let*

$$\widehat{s} = [\, v_1 \colon [\![e_1]\!]_{\mathrm{exp}} s \mid \ldots \mid v_n \colon [\![e_n]\!]_{\mathrm{exp}} s \,].$$

If e is an expression (or boolean expression) such that $\mathrm{FV}(e) \subseteq \{v_1, \ldots, v_n\}$*, then*

$$[\![e/\delta]\!]_{\mathrm{exp}} s = [\![e]\!]_{\mathrm{exp}} \widehat{s}.$$

Here we have introduced a notation for describing stores (and more generally, functions with finite domains) by enumeration: We write $[\, x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$ (where $x_1, \ldots, x_n$ are distinct) for the function with domain $\{x_1, \ldots, x_n\}$ that maps each x_i into y_i.

Next, we generalize this result to *partial* substitutions that need not act upon all the free variables of a phrase: When $\mathrm{FV}(p)$ is not a subset of $\{v_1, \ldots, v_n\}$,

$$p/v_1 \to e_1, \ldots, v_n \to e_n$$

abbreviates

$$p/v_1 \to e_1, \ldots, v_n \to e_n, v'_1 \to v'_1, \ldots, v'_k \to v'_k,$$

where $\{v'_1, \ldots, v'_k\} = \mathrm{FV}(p) - \{v_1, \ldots, v_n\}$. Then the above proposition can be generalized to

Proposition 2 *(Partial Substitution Law for Expressions) Suppose e is an expression (or boolean expression), and let δ abbreviate the substitution*

$$v_1 \to e_1, \ldots, v_n \to e_n,$$

Then let s be a store such that $(\mathrm{FV}(e) - \{v_1, \ldots, v_n\}) \cup \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \subseteq \mathrm{dom}\, s$*, and let*

$$\widehat{s} = [\, s \mid v_1 \colon [\![e_1]\!]_{\mathrm{exp}} s \mid \ldots \mid v_n \colon [\![e_n]\!]_{\mathrm{exp}} s \,].$$

Then

$$[\![e/\delta]\!]_{\mathrm{exp}} s = [\![e]\!]_{\mathrm{exp}} \widehat{s}.$$

Here we have introduced a notation for describing the extension or variation of a function. We write $[\, f \mid x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$ (where $x_1, \ldots, x_n$ are distinct) for the function whose domain is the union of the domain of f with $\{x_1, \ldots, x_n\}$, that maps each x_i into y_i and all other members x of the domain of f into $f\, x$.

A similar result for assertions will be given in the next section. As we will see in the next chapter, however, the situation for commands is more subtle.

2.1 The Meaning of Assertions

When s is a store, h is a heap, and p is an assertion whose free variables all belong to the domain of s, we write

$$s, h \models p$$

to indicate that the state s, h *satisfies* p, or p *is true in* s, h, or p *holds in* s, h. Then the following formulas define this relation by induction on the structure of p. (Here we write $h_0 \perp h_1$ when h_0 and h_1 are heaps with disjoint domains, and $h_0 \cdot h_1$ to denote the union of heaps with disjoint domains.)

$$\begin{aligned}
s, h \models b \ &\text{iff}\ [\![b]\!]_{\text{bexp}} s = \mathbf{true}, \\
s, h \models \neg\, p \ &\text{iff}\ s, h \models p \text{ is false}, \\
s, h \models p_0 \wedge p_1 \ &\text{iff}\ s, h \models p_0 \text{ and } s, h \models p_1 \\
&\qquad\qquad \text{(and similarly for } \vee, \Rightarrow, \Leftrightarrow), \\
s, h \models \forall v.\ p \ &\text{iff}\ \forall x \in \mathbf{Z}.\ [\, s \mid v \colon x \,], h \models p, \\
s, h \models \exists v.\ p \ &\text{iff}\ \exists x \in \mathbf{Z}.\ [\, s \mid v \colon x \,], h \models p, \\
s, h \models \mathbf{emp} \ &\text{iff}\ \operatorname{dom} h = \{\}, \\
s, h \models e \mapsto e' \ &\text{iff}\ \operatorname{dom} h = \{[\![e]\!]_{\text{exp}} s\} \text{ and } h([\![e]\!]_{\text{exp}} s) = [\![e']\!]_{\text{exp}} s, \\
s, h \models p_0 \ * \ p_1 \ &\text{iff}\ \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\
&\qquad\qquad s, h_0 \models p_0 \text{ and } s, h_1 \models p_1, \\
s, h \models p_0 \mathrel{-\!\!*} p_1 \ &\text{iff}\ \forall h'.\ (h' \perp h \text{ and } s, h' \models p_0) \text{ implies} \\
&\qquad\qquad s, h \cdot h' \models p_1.
\end{aligned}$$

All but the last four formulas coincide with the standard interpretation of predicate logic, with the heap h being carried along without change.

When $s, h \models p$ holds for all states s, h (such that the domain of s contains the free variables of p), we say that p is *valid*. When $s, h \models p$ holds for some state s, h, we say that p is *satisfiable*.

The following illustrates the use of these formulas to determine the meaning of an assertion:

$$\begin{aligned}
s, h \models \mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1 \text{ iff } & \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
& \text{and } s, h_0 \models \mathsf{x} \mapsto 0 \\
& \text{and } s, h_1 \models \mathsf{y} \mapsto 1 \\
\text{iff } & \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
& \text{and } \operatorname{dom} h_0 = \{s\,\mathsf{x}\} \text{ and } h_0(s\,\mathsf{x}) = 0 \\
& \text{and } \operatorname{dom} h_1 = \{s\,\mathsf{y}\} \text{ and } h_1(s\,\mathsf{y}) = 1 \\
\text{iff } & s\,\mathsf{x} \neq s\,\mathsf{y} \\
& \text{and } \operatorname{dom} h = \{s\,\mathsf{x}, s\,\mathsf{y}\} \\
& \text{and } h(s\,\mathsf{x}) = 0 \text{ and } h(s\,\mathsf{y}) = 1 \\
\text{iff } & s\,\mathsf{x} \neq s\,\mathsf{y} \text{ and } h = [\,s\,\mathsf{x}\colon 0 \mid s\,\mathsf{y}\colon 1\,].
\end{aligned}$$

The following illustrate the meaning of $\mapsto$ and $\hookrightarrow$ (including the abbreviations defined in Section 1.4):

$$\begin{aligned}
s, h \models \mathsf{x} \mapsto \mathsf{y} &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \models \mathsf{x} \mapsto - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \\
s, h \models \mathsf{x} \hookrightarrow \mathsf{y} &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \models \mathsf{x} \hookrightarrow - &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \\
s, h \models \mathsf{x} \mapsto \mathsf{y}, \mathsf{z} &\text{ iff } h = [\,s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z}\,] \\
s, h \models \mathsf{x} \mapsto -, - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}, s\,\mathsf{x} + 1\} \\
s, h \models \mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{z} &\text{ iff } h \supseteq [\,s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z}\,] \\
s, h \models \mathsf{x} \hookrightarrow -, - &\text{ iff } \operatorname{dom} h \supseteq \{s\,\mathsf{x}, s\,\mathsf{x} + 1\}.
\end{aligned}$$

To illustrate the meaning of the separating conjunction, suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct addresses, so that

$$h_0 = [\, s\,\mathsf{x}\colon 0\,] \qquad \text{and} \qquad h_1 = [\, s\,\mathsf{y}\colon 1\,]$$

are heaps with disjoint domains. Then

If p is:	then $s, h \models p$ iff:
$\mathsf{x} \mapsto 0$	$h = h_0$
$\mathsf{y} \mapsto 1$	$h = h_1$
$\mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \ * \ \mathsf{x} \mapsto 0$	**false**
$\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1$	$h = h_0$ or $h = h_1$
$\mathsf{x} \mapsto 0 \ * \ (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$(\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1) \ * \ (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1 \ * \ (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	**false**
$\mathsf{x} \mapsto 0 \ * \ \mathbf{true}$	$h_0 \subseteq h$
$\mathsf{x} \mapsto 0 \ * \ \neg\, \mathsf{x} \mapsto 0$	$h_0 \subseteq h$.

Here the behavior of disjunction is slightly surprising.

The effect of substitution on the meaning of assertions is similar to that on expressions. We consider only the more general partial case:

Proposition 3 *(Partial Substitution Law for Assertions) Suppose p is an assertion, and let δ abbreviate the substitution*

$$v_1 \to e_1, \ldots, v_n \to e_n,$$

Then let s be a store such that $(\mathrm{FV}(p) - \{v_1, \ldots, v_n\}) \cup \mathrm{FV}(e_1) \cup \cdots \cup \mathrm{FV}(e_n) \subseteq \mathrm{dom}\, s$*, and let*

$$\widehat{s} = [\, s \mid v_1\colon [\![e_1]\!]_{\mathrm{exp}} s \mid \ldots \mid v_n\colon [\![e_n]\!]_{\mathrm{exp}} s\,].$$

Then

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

2.2 Inference

We will reason about assertions using inference rules of the form

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\mathcal{C},}$$

where the zero or more $\mathcal{P}_i$ are called the *premisses* and the $\mathcal{C}$ is called the *conclusion*.

The premisses and conclusion are schemata, i.e., they may contain *metavariables*, each of which ranges over some set of phrases, such as expressions, variables, or assertions. To avoid confusion, we will use italic (or occasionally Greek) characters for metavariables, but sans serif characters for the *object* variables of the logic and programming lanugage.

An instance of an inference rule is obtained by replacing each metavariable by a phrase in its range. These replacements must satisfy the *side conditions* (if any) of the rule. (Since this is replacement of metavariables rather than substitution for variables, there is never any renaming.) For instance,

Inference Rules	Instances
$\dfrac{p_0 \qquad p_0 \Rightarrow p_1}{p_1}$	$\dfrac{\mathsf{x}+0=\mathsf{x} \qquad \mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0}{\mathsf{x}=\mathsf{x}+0}$
$\dfrac{}{e_1 = e_0 \Rightarrow e_0 = e_1}$	$\dfrac{}{\mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0}$
$\mathsf{x}+0=\mathsf{x}$	$\mathsf{x}+0=\mathsf{x}$

An inference rule is *sound* iff, for all instances, if the premisses of the instance are all valid, then the conclusion is valid.

A *formal proof* (for assertions) is a sequence of assertions, each of which is the conclusion of some instance of a sound inference rule whose premisses occur earlier in the sequence. For example,

$$\begin{gathered}\mathsf{x}+0=\mathsf{x}\\ \mathsf{x}+0=\mathsf{x} \Rightarrow \mathsf{x}=\mathsf{x}+0\\ \mathsf{x}=\mathsf{x}+0.\end{gathered}$$

Since we require the inference rules used in the proof to be sound, it follows that the assertions in a formal proof must all be valid.

Notice the distinction between formal proofs, whose constituents are assertions written in separation logic (or, in the next chapter, specifications containing assertions and commands), and *meta*-proofs, which are ordinary mathematical proofs using the semantics of assertions (and, in the next chapter, of commands).

An inference rule with zero premisses is called an *axiom schema*. The overbar is often omitted. (Notice that the first assertion in a proof must be an instance of an axiom schema.) An axiom schema containing no metavariables (so that it is its own unique instance) is called an *axiom*. The overbar is usually omitted.

The following are inference rules that are sound for predicate calculus, and remain sound for the extension to assertions in separation logic:

$$\frac{p \qquad p \Rightarrow q}{q} \quad \text{(modus ponens)}$$

$$\frac{p \Rightarrow q}{p \Rightarrow (\forall v.\ q)} \quad \text{when } v \notin \mathrm{FV}(p) \tag{2.1}$$

$$\frac{p \Rightarrow q}{(\exists v.\ p) \Rightarrow q} \quad \text{when } v \notin \mathrm{FV}(q).$$

In addition, the following axiom schemas are sound for predicate calculus

and assertions in separation logic:

$$
\begin{aligned}
p &\Rightarrow (q \Rightarrow p)\\
(p \Rightarrow (q \Rightarrow r)) &\Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))\\
(p \wedge q) &\Rightarrow p\\
(p \wedge q) &\Rightarrow q\\
p &\Rightarrow (q \Rightarrow (p \wedge q))\\
p &\Rightarrow (p \vee q)\\
q &\Rightarrow (p \vee q)\\
(p \Rightarrow r) &\Rightarrow ((q \Rightarrow r) \Rightarrow ((p \vee q) \Rightarrow r))\\
(p \Rightarrow q) &\Rightarrow ((p \Rightarrow \neg\, q) \Rightarrow \neg\, p)\\
\neg(\neg\, p) &\Rightarrow p\\
(p \Leftrightarrow q) &\Rightarrow ((p \Rightarrow q) \wedge (q \Rightarrow p))\\
((p \Rightarrow q) \wedge (q \Rightarrow p)) &\Rightarrow (p \Leftrightarrow q)\\
(\forall v.\ p) &\Rightarrow (p/v \rightarrow e)\\
(p/v \rightarrow e) &\Rightarrow (\exists v.\ p).
\end{aligned}
\tag{2.2}
$$

The rules in (2.1) and (2.2) (with the exception of the rules involving $\Leftrightarrow$) are taken from Kleene [44, page 82]. They are (one of many possible) complete sets of *logical* inference rules (i.e., rules that are sound for any interpretation of the domain of discourse or the function and predicate symbols).

It is important to notice the difference between

$$\frac{p}{q} \qquad \text{and} \qquad \overline{p \Rightarrow q.}$$

A rule of the first form will be sound providing, for all instances, the validity of p implies the validity of q, i.e., whenever p holds for all states, q holds for all states. But a rule of the second form will be sound providing, for all instances, $p \Rightarrow q$ is valid, i.e., for every state, if p holds in that state then q holds in the same state. Consider the rule of generalization (which can be derived from the rules above),

$$\frac{p}{\forall v.\ p}.$$

This rule is sound; for example, the instance

$$\frac{\mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x}}{\forall \mathsf{x}.\ \mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x}}$$

is sound because its conclusion is valid, while the instance

$$\frac{\mathsf{x} = 0}{\forall \mathsf{x}.\ \mathsf{x} = 0}$$

is sound because its premiss is not valid.

On the other hand, the corresponding implication

$$p \Rightarrow \forall v.\ p \qquad \text{(unsound)}$$

is not sound, since, for instance, there is a state in which $\mathsf{x} = 0$ holds but $\forall \mathsf{x}.\ \mathsf{x} = 0$ does not.

We have already seen the following general inference rules for assertions:

$$\begin{aligned}
p_0 \ * \ p_1 &\Leftrightarrow p_1 \ * \ p_0 \\
(p_0 \ * \ p_1) \ * \ p_2 &\Leftrightarrow p_0 \ * \ (p_1 \ * \ p_2) \\
p \ * \ \mathbf{emp} &\Leftrightarrow p \\
(p_0 \vee p_1) \ * \ q &\Leftrightarrow (p_0 \ * \ q) \vee (p_1 \ * \ q) \\
(p_0 \wedge p_1) \ * \ q &\Rightarrow (p_0 \ * \ q) \wedge (p_1 \ * \ q) \\
(\exists x.\ p_0) \ * \ p_1 &\Leftrightarrow \exists x.\ (p_0 \ * \ p_1) \quad \text{when } x \text{ not free in } p_1 \\
(\forall x.\ p_0) \ * \ p_1 &\Rightarrow \forall x.\ (p_0 \ * \ p_1) \quad \text{when } x \text{ not free in } p_1
\end{aligned} \tag{2.3}$$

$$\frac{p_0 \Rightarrow p_1 \qquad q_0 \Rightarrow q_1}{p_0 \ * \ q_0 \Rightarrow p_1 \ * \ q_1} \quad \text{(monotonicity)}$$

$$\frac{p_0 \ * \ p_1 \Rightarrow p_2}{p_0 \Rightarrow (p_1 \mathrel{-\!\!*} p_2)} \quad \text{(currying)} \qquad \frac{p_0 \Rightarrow (p_1 \mathrel{-\!\!*} p_2)}{p_0 \ * \ p_1 \Rightarrow p_2.} \quad \text{(decurrying)}$$

as well as specific rules for $\mapsto$ and $\hookrightarrow$:

$$\begin{aligned}
e_0 \mapsto e_0' \wedge e_1 \mapsto e_1' &\Leftrightarrow e_0 \mapsto e_0' \wedge e_0 = e_1 \wedge e_0' = e_1' \\
e_0 \hookrightarrow e_0' \ * \ e_1 \hookrightarrow e_1' &\Rightarrow e_0 \neq e_1 \\
\mathbf{emp} &\Leftrightarrow \forall x.\ \neg(x \hookrightarrow -) \\
(e \hookrightarrow e') \wedge p &\Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).
\end{aligned} \tag{2.4}$$

2.3 Special Classes of Assertions

There are several classes of assertions that play an important role in separation logic. In each case, the class has a semantic definition that implies the soundness of additional axiom schemata; often there are also useful syntactic criteria that imply membership in the class.

2.3.1 Pure Assertions

The simplest such class is that of pure assertions, which are independent of the heap. More precisely, an assertion p is *pure* iff, for all stores s and all heaps h and h',

$$s, h \models p \text{ iff } s, h' \models p.$$

A sufficient syntactic criteria is that an assertion is pure if it does not contain **emp**, $\mapsto$, or $\hookrightarrow$.

When all of the subphrases of an assertion are pure, the distinction between separating and ordinary operations collapses, so that $*$ and $\wedge$ are interchangeable, as are $-\!\!*$ and $\Rightarrow$. The following axiom schemata delineate the consequences when some subassertions are pure:

$$\begin{array}{rl}
p_0 \wedge p_1 \Rightarrow p_0 * p_1 & \text{when } p_0 \text{ or } p_1 \text{ is pure} \\
p_0 * p_1 \Rightarrow p_0 \wedge p_1 & \text{when } p_0 \text{ and } p_1 \text{ are pure} \\
(p \wedge q) * r \Leftrightarrow (p * r) \wedge q & \text{when } q \text{ is pure} \\
(p_0 -\!\!* p_1) \Rightarrow (p_0 \Rightarrow p_1) & \text{when } p_0 \text{ is pure} \\
(p_0 \Rightarrow p_1) \Rightarrow (p_0 -\!\!* p_1) & \text{when } p_0 \text{ and } p_1 \text{ are pure.}
\end{array}$$

(The third of these schemata is ubiquitous in proofs of programs.)

2.3.2 Strictly Exact Assertions

At the opposite extreme from pure assertions are strictly exact assertions, which uniquely determine the heap. An assertion is *strictly exact* iff, for all stores s and all heaps h and h',

$$s, h \models p \text{ and } s, h' \models p \text{ implies } h = h'.$$

(This classification of assertions was introduced by Yang [10].)

Examples of strictly exact assertions include:

- **emp**.
- $e \mapsto e'$.
- $p * q$, when p and q are strictly exact.
- $p \wedge q$, when p or q is strictly exact.
- p, when $p \Rightarrow q$ is valid and q is strictly exact.

Proposition 4 *When q is strictly exact,*

$$((q * \mathbf{true}) \wedge p) \Rightarrow (q * (q \mathrel{-\!\!*} p))$$

is valid.

PROOF Suppose $s, h \models (q * \mathbf{true}) \wedge p$, so that $s, h \models q * \mathbf{true}$ and $s, h \models p$. Then there are heaps h_0 and h_1 such that $h_0 \perp h_1$, $h_0 \cdot h_1 = h$, and $s, h_0 \models q$.

To see that $s, h_1 \models q \mathrel{-\!\!*} p$, let h' be any heap such that $h' \perp h_1$ and $s, h' \models q$. Since q is strictly exact, $h' = h_0$, so that $h' \cdot h_1 = h_0 \cdot h_1 = h$, and thus $s, h' \cdot h_1 \models p$.

Then $s, h_0 \cdot h_1 \models q * (q \mathrel{-\!\!*} p)$, so that $s, h \models q * ((q \mathrel{-\!\!*} p)$.

END OF PROOF

For example, taking q to be the strictly exact assertion $e \mapsto e'$ gives the final axiom schema in (2.4).

2.3.3 Precise Assertions

Given a heap, if a precise assertion holds for any subheap, then it holds for a unique subheap. In other words, an assertion q is *precise* iff, for all s and h, there is at most one $h' \subseteq h$ such that

$$s, h' \models q.$$

Examples of precise assertions include:

- Strictly exact assertions
- $e \mapsto -$
- $p * q$, when p and q are precise

- $p \wedge q$, when p or q is precise
- p, when $p \Rightarrow q$ is valid and q is precise
- $\mathsf{list}\ \alpha\ e$ and $\exists \alpha.\ \mathsf{list}\ \alpha\ e$
- $\mathsf{tree}\ \tau\ (e)$ and $\exists \tau.\ \mathsf{tree}\ \tau\ (e)$,

where list is defined in Section 1.6, and tree is defined in Section 1.7.

On the other hand, the following are instances of imprecise assertions:

$$\mathbf{true} \qquad \mathbf{emp} \vee \mathsf{x} \mapsto 10 \qquad \mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 10 \qquad \exists \mathsf{x}.\ \mathsf{x} \mapsto 10$$
$$\mathsf{dag}\ \tau\ (\mathsf{i}) \qquad \exists \tau.\ \mathsf{dag}\ \tau\ (\mathsf{i}),$$

where dag is defined as in Section 1.7.

There is a close connection between preciseness and distributivity. The semi-distributive laws

$$(p_0 \wedge p_1)\ *\ q \Rightarrow (p_0\ *\ q) \wedge (p_1\ *\ q)$$
$$(\forall x.\ p)\ *\ q \Rightarrow \forall x.\ (p\ *\ q) \quad \text{when } x \text{ not free in } q$$

are valid for all assertions. But their converses

$$(p_0\ *\ q) \wedge (p_1\ *\ q) \Rightarrow (p_0 \wedge p_1)\ *\ q$$
$$\forall x.\ (p\ *\ q) \Rightarrow (\forall x.\ p)\ *\ q \quad \text{when } x \text{ not free in } q$$

are not. For example, when

$$s(\mathsf{x}) = 1 \quad s(\mathsf{y}) = 2 \quad h = [\,1\colon 10 \mid 2\colon 20\,],$$

the assertion

$$(\mathsf{x} \mapsto 10\ *\ (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20)) \wedge (\mathsf{y} \mapsto 20\ *\ (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20))$$

is true, but

$$((\mathsf{x} \mapsto 10 \wedge \mathsf{y} \mapsto 20)\ *\ (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20))$$

is false.

However, the converses are valid when q is precise:

Proposition 5 *When q is precise,*

$$(p_0 \ * \ q) \wedge (p_1 \ * \ q) \Rightarrow (p_0 \wedge p_1) \ * \ q$$

is valid. When q is precise and x is not free in q,

$$\forall x.\ (p \ * \ q) \Rightarrow (\forall x.\ p) \ * \ q$$

is valid.

PROOF (of the first law) Suppose $s, h \models (p_0 \ * \ q) \wedge (p_1 \ * \ q)$. Then there are:

- An $h_0 \subseteq h$ such that $s, h - h_0 \models p_0$ and $s, h_0 \models q$, and
- An $h_1 \subseteq h$ such that $s, h - h_1 \models p_1$ and $s, h_1 \models q$.

Thus, since q is precise, $h_0 = h_1$, $h - h_0 = h - h_1$, $s, h - h_0 \models p_0 \wedge p_1$, and $s, h \models (p_0 \wedge p_1) \ * \ q$. END OF PROOF

2.3.4 Intuitionistic Assertions

Intuitionistic assertions are monotone with respect to the extension of heaps. An assertion i is *intuitionistic* iff, for all stores s and heaps h and h':

$$(h \subseteq h' \text{ and } s, h \models i) \text{ implies } s, h' \models i.$$

Assume i and i' are intuitionistic assertions, p is any assertion, e and e' are expressions, and τ denotes an S-expression. Then the following assertions are intuitionistic:

Any pure assertion	$p \ * \ i$
$p \mathrel{-\!\!*} i$	$i \mathrel{-\!\!*} p$
$i \wedge i'$	$i \vee i'$
$\forall v.\ i$	$\exists v.\ i$
$\mathsf{dag}\ \tau\ (e)$	$\exists \tau.\ \mathsf{dag}\ \tau\ (e)$,

and as special cases:

$$p \ * \ \mathbf{true} \qquad \mathbf{true} \mathrel{-\!\!*} p \qquad e \hookrightarrow e'.$$

The following inference rules are sound when i and i' are intuitionistic:

$$(i \ * \ i') \Rightarrow (i \wedge i')$$

$$(i \ * \ p) \Rightarrow i \qquad i \Rightarrow (p \mathrel{-\!\!*} i)$$

$$\frac{p \Rightarrow i}{(p \ * \ \mathbf{true}) \Rightarrow i} \qquad \frac{i \Rightarrow p}{i \Rightarrow (\mathbf{true} \mathrel{-\!\!*} p).}$$

The last two of these rules, in conjunction with the rules

$$p \Rightarrow (p \ * \ \mathbf{true}) \qquad (\mathbf{true} \mathrel{-\!\!*} p) \Rightarrow p,$$

which hold for all assertions, implies that $p \ * \ \mathbf{true}$ is the strongest intuitionistic assertion weaker than p, and $\mathbf{true} \mathrel{-\!\!*} p$ is the weakest intuitionistic assertion that is stronger than p. In turn this implies, when i is intuitionistic,

$$i \Leftrightarrow (i \ * \ \mathbf{true}) \qquad (\mathbf{true} \mathrel{-\!\!*} i) \Leftrightarrow i.$$

If we define the operations

$$\overset{\text{i}}{\neg} p \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (\neg p)$$

$$p \overset{\text{i}}{\Rightarrow} q \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (p \Rightarrow q)$$

$$p \overset{\text{i}}{\Leftrightarrow} q \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (p \Leftrightarrow q),$$

then the assertions built from pure assertions and $e \hookrightarrow e'$, using these operations and $\wedge$, $\vee$, $\forall$, $\exists$, $*$, and $\mathrel{-\!\!*}$ form an intuitionistic version of separation logic, which can be translated into the classical version by replacing the left sides of the above definitions by their right sides.

This is the modal translation from intuitionistic to classical separation logic given by Ishtiaq and O'Hearn [5]. It allows us to reason intuitionistically within the classical logic rather than using the intuitionistic logic.

2.3.5 Supported Assertions

It is easily seen that no assertion that is true in any state can be both precise and intuitionistic.

Thus satisfiable precise assertions do not inhabit the intuitionistic world. This raises the question of whether there is a class of assertions that bears

the same relationship to intuitionistic assertions that precise assertions bear to arbitrary (i.e., classical) assertions. In this section, we will see that this role is filled by supported assertions.

An assertion q is *supported* iff, for all s, h_0, and h_1, if $h_0 \cup h_1$ is a function, and $s, h_0 \models q$ and $s, h_1 \models q$ are true, then there is an h' such that $h' \subseteq h_0$, $h' \subseteq h_1$, and $s, h' \models q$ is true. Equivalently,

Proposition 6 *An assertion q is* supported *iff, for all s and h, if the set*

$$H = \{\, h' \mid h' \subseteq h \text{ and } s, h' \models q \,\}$$

is nonempty, then it has a least element.

PROOF Suppose that q is supported, fix s and h, and let h_0 be a member of H with minimum domain size, and h_1 be any member of H. Since h_0 and h_1 are both subsets of h, $h_0 \cup h_1$ must be a function. Then the first definition guarantees that there is an $h' \in H$ that is a subset of both h_0 and h_1. But h' must be equal to h_0, since otherwise it would have a smaller domain size. Thus $h_0 \subseteq h_1$ for every $h_1 \in H$.

On the other hand, suppose that q meets the conditions of the proposition, $h_0 \cup h_1$ is a function, $s, h_0 \models q$ and $s, h_1 \models q$ are true. Take h to be $h_0 \cup h_1$, so that $h_0, h_1 \in H$. Then take h' to be the least element of H. END OF PROOF

For example, the following assertions are imprecise, intuitionistic, and supported:

$$\mathbf{true} \qquad \mathsf{x} \hookrightarrow 10 \qquad \mathsf{x} \hookrightarrow 10 \wedge \mathsf{y} \hookrightarrow 10 \qquad \mathsf{dag}\,\tau\,(\mathsf{i}) \qquad \exists\tau.\, \mathsf{dag}\,\tau\,(\mathsf{i}),$$

imprecise, intuitionistic, and unsupported:

$$\mathsf{x} \hookrightarrow 10 \vee \mathsf{y} \hookrightarrow 10 \qquad \exists \mathsf{x}.\, \mathsf{x} \hookrightarrow 10 \qquad \neg\, \mathbf{emp},$$

imprecise, nonintuitionistic, and supported:

$$\mathbf{emp} \vee \mathsf{x} \mapsto 10,$$

and imprecise, nonintuitionistic, and unsupported:

$$\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 10 \qquad \exists \mathsf{x}.\, \mathsf{x} \mapsto 10.$$

When q is supported and the remaining assertions are intuitionistic, the semidistributive laws given earlier become full distributive laws:

Proposition 7 *When p_0 and p_1 are intuitionistic and q is supported,*

$$(p_0 \ * \ q) \wedge (p_1 \ * \ q) \Rightarrow (p_0 \wedge p_1) \ * \ q$$

is valid. When p is intuitionistic, q is supported, and x is not free in q,

$$\forall x.\ (p \ * \ q) \Rightarrow (\forall x.\ p) \ * \ q$$

is valid.

PROOF (of the first law): Suppose $s, h \models (p_0 \ * \ q) \wedge (p_1 \ * \ q)$. Then there are:

$$\text{An } h_0 \subseteq h \text{ such that } s, h - h_0 \models p_0 \text{ and } s, h_0 \models q,$$

$$\text{An } h_1 \subseteq h \text{ such that } s, h - h_1 \models p_1 \text{ and } s, h_1 \models q.$$

Then, since q is supported and $h_0 \cup h_1$ is a function, there is an $h' \subseteq h_0, h_1$ such that $s, h' \models q$. Moreover, since $h - h_0, h - h_1 \subseteq h - h'$, and p_0 and p_1 are intuitionistic, $s, h - h' \models p_0 \wedge p_1$, and therefore $s, h \models (p_0 \wedge p_1) \ * \ q$.

END OF PROOF

We have already seen that, if p is any assertion, then $p \ * \ \mathbf{true}$ is intuitionistic, and if i is intuitionistic, then $i \Leftrightarrow (i \ * \ \mathbf{true})$. Thus, $- \ * \ \mathbf{true}$ maps arbitrary assertions into intuitionistic assertions, and acts as an identity (up to equivalence of assertions) on the latter.

In addition,

Proposition 8 *(1) If p is precise, then p is supported. (2) q is supported iff $q \ * \ \mathbf{true}$ is supported.*

PROOF (1) If p is precise, then, for any s and h, the set $H = \{\, h' \mid h' \subseteq h \text{ and } s, h' \models q \,\}$ in Proposition 6 contains at most one element.

(2) Suppose q is supported, $h_0 \cup h_1$ is a function, $s, h_0 \models q \ * \ \mathbf{true}$ and $s, h_1 \models q \ * \ \mathbf{true}$. Then there are $h_0' \subseteq h_0$ and $h_1' \subseteq h_1$ such that $s, h_0' \models q$ and $s, h_1' \models q$, and since q is supported, there is an h' that is a subset of h_0' and h_1', and therefore h_0 and h_1, such that $s, h' \models q$, and therefore $s, h' \models q \ * \ \mathbf{true}$.

Suppose $q \ * \ \mathbf{true}$ is supported, $h_0 \cup h_1$ is a function, $s, h_0 \models q$ and $s, h_1 \models q$. Then $s, h_0 \models q \ * \ \mathbf{true}$ and $s, h_1 \models q \ * \ \mathbf{true}$, and since $q \ * \ \mathbf{true}$ is supported, there is a common subset h' of h_0 and h_1 such that $s, h' \models q \ * \ \mathbf{true}$. But then there is a subset h'' of h', and therefore of h_0 and h_1, such that $s, h'' \models q$.

END OF PROOF

(The analogous conjecture, that q is supported iff **true** $-\!\!*\ q$ is supported, fails in both directions. Suppose q is $1 \hookrightarrow 1 \vee 2 \hookrightarrow 2 \vee \mathbf{emp}$. Then q is supported, since the empty heap is the least heap in which it holds, but **true** $-\!\!*\ q$ is not supported, since it holds for the heaps $[\,1{:}\,1\,]$ and $[\,2{:}\,2\,]$, but not for their only common subheap, which is the empty heap. On the other hand, suppose q is $1 \hookrightarrow 1 \vee 2 \mapsto 2$. Then **true** $-\!\!*\ q$ is supported, since $[\,1{:}\,1\,]$ is the least heap for which it holds, but q is not supported, since it holds for $[\,1{:}\,1\,]$ and $[\,2{:}\,2\,]$, but not for the empty heap.)

Thus $-\ *\ \mathbf{true}$ maps precise assertions into supported intuitionistic assertions and acts as an identity (up to equivalence of assertions) on the latter.

2.3.6 The Precising Operation

Having found an operation that maps precise assertions into supported intuitionistic assertions, we now introduce an operation, due to Yang, that moves in the opposite direction, which we call the *precising* operation:

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

For example,

$$\mathrm{Pr}\ \mathbf{true}\ \mathrm{iff}\ \mathbf{emp}$$

$$\mathrm{Pr}\ (\mathsf{x} \hookrightarrow 10)\ \mathrm{iff}\ \mathsf{x} \mapsto 10$$

$$\mathrm{Pr}\ (\mathbf{emp} \vee \mathsf{x} \mapsto 10)\ \mathrm{iff}\ \mathbf{emp}$$

$$\mathrm{Pr}\ (\mathsf{x} \hookrightarrow 10 \wedge \mathsf{y} \hookrightarrow 10)\ \mathrm{iff}\ \mathbf{if}\ \mathsf{x} = \mathsf{y}\ \mathbf{then}\ \mathsf{x} \mapsto 10\ \mathbf{else}\ (\mathsf{x} \mapsto 10\ *\ \mathsf{y} \mapsto 10)$$

Then

Proposition 9 *(1) If p is supported, then* Pr p *is precise. (2) If p is precise, then* Pr $p \Leftrightarrow p$.

Proof (1) Suppose p is supported, and $h_0, h_1 \subseteq h$ are such that $s, h_0 \models \mathrm{Pr}\ p$ and $s, h_1 \models \mathrm{Pr}\ p$.

We must show $h_0 = h_1$. Assume the contrary. Since Pr p is $p \wedge \neg(p\ *\ \neg\,\mathbf{emp})$,we have $s, h_0 \models p$ and $s, h_1 \models p$. Then since p is supported, there is a common subset h' of h_0 and h_1 such that $s, h' \models p$. Since $h_0 \neq h_1$, however,

h' must be a proper subset of h_i for $i = 0$ or $i = 1$. Thus $h_i = h' \cdot (h_i - h')$, where $s, h_i - h' \models \neg\,\mathbf{emp}$. Then $s, h_i \models p * \neg\,\mathbf{emp}$, which contradicts $s, h_i \models \mathrm{Pr}\ p$.

(2) Obviously, $\mathrm{Pr}\ p \Rightarrow p$. To show the opposite implication when p is precise, assume $s, h \models p$. If $s, h \models p * \neg\,\mathbf{emp}$ held, then there would be a proper subset h' of h such that $s, h' \models p$, which would contradict the preciseness of p. Thus $s, h \models \neg(p * \neg\,\mathbf{emp})$. END OF PROOF

Thus Pr maps supported operations into precise operations, and acts as an identity on the latter.

(The conjecture that, when p is supported, Pr p is the weakest precise assertion stronger than p, is false. Suppose p is the supported assertion $1 \mapsto 1 \vee \mathbf{emp}$. Then $1 \mapsto 1$ is a precise assertion stronger than p, but Pr p, which is equivalent to **emp**, is not weaker than $1 \mapsto 1$.)

Additional relationships between $- * \mathbf{true}$ and Pr are provided by

Proposition 10

(1) $\mathrm{Pr}\ (p * \mathbf{true}) \Rightarrow p$.

(2) $p \Rightarrow \mathrm{Pr}\ (p * \mathbf{true})$ *when* p *is precise.*

(3) $(\mathrm{Pr}\ q) * \mathbf{true} \Rightarrow q$ *when* q *is intuitionistic.*

(4) $q \Rightarrow (\mathrm{Pr}\ q) * \mathbf{true}$ *when* q *is supported.*

Therefore, $\mathrm{Pr}\ (p * \mathbf{true}) \Leftrightarrow p$ *when* p *is precise, and* $(\mathrm{Pr}\ q) * \mathbf{true} \Rightarrow q$ *when* q *is supported and intuitionistic.*

PROOF (1) Suppose $s, h \models \mathrm{Pr}\ (p * \mathbf{true})$. Then $s, h \models p * \mathbf{true}$ and $s, h \models \neg(p * \mathbf{true} * \neg\,\mathbf{emp})$, and there is an $h' \subseteq h$ such that $s, h' \models p$. If $h' = h$, we are done; otherwise, h' is a proper subset of h, so that $s, h \models p * \mathbf{true} * \neg\,\mathbf{emp}$, which contradicts $s, h \models \neg(p * \mathbf{true} * \neg\,\mathbf{emp})$.

(2) Suppose $s, h \models p$. Then $s, h \models p * \mathbf{true}$. Moreover, $s, h \models \neg(p * \mathbf{true} * \neg\,\mathbf{emp})$, for otherwise $s, h \models p * \mathbf{true} * \neg\,\mathbf{emp}$ would imply that there is a proper subset h' of h such that $s, h' \models p$, which would contradict the preciseness of p.

(3) Suppose $s, h \models (\mathrm{Pr}\ q) * \mathbf{true}$. Then there is an $h' \subseteq h$ such that $s, h' \models \mathrm{Pr}\ q$. Then $s, h' \models q$, and since q is intuitionistic, $s, h \models q$.

(4) Suppose $s, h \models q$. Since q is supported, there is a least $h' \subseteq h$ such that $s, h' \models q$. Then $s, h' \models \mathrm{Pr}\ q$, since otherwise $s, h' \models q * \neg\,\mathbf{emp}$,

which would imply that a proper subset h'' of h' would satisfy $s, h'' \models q$, contradicting the leastness of h'. Thus $s, h \models (\mathrm{Pr}\ q) * \mathbf{true}$.

END OF PROOF

Thus $- * \mathbf{true}$ and Pr are isomorphisms between the set of precise assertions and the set of supported intuitionistic assertions, and act as identities on these sets:

2.4 Some Derived Inference Rules

We conclude this chapter with the deriviations of five inference rules, which are interesting in their own right and will be useful later.

An inference-rule derivation is a proof schema, such that appropriate replacements of the metavariables will yield a formal proof from assumptions (steps that do not follow from previous steps). At the schematic level the assumptions are the premisses of the rule to be proved, and the final step is the conclusion. Thus the derivation shows how any instance of the derived rule can be replaced by a sound proof fragment.

$$\text{To derive:} \quad \frac{}{q * (q \mathrel{-\!\!*} p) \Rightarrow p} \tag{2.5}$$

1. $q * (q \mathrel{-\!\!*} p) \Rightarrow (q \mathrel{-\!\!*} p) * q$ $\qquad (p_0 * p_1 \Rightarrow p_1 * p_0)$
2. $(q \mathrel{-\!\!*} p) \Rightarrow (q \mathrel{-\!\!*} p)$ $\qquad (p \Rightarrow p)$
3. $(q \mathrel{-\!\!*} p) * q \Rightarrow p$ $\qquad$ (decurrying, 2)
4. $q * (q \mathrel{-\!\!*} p) \Rightarrow p$ $\qquad$ (trans impl, 1, 3)

where *transitive implication* is the inference rule

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r}$$

(which can be derived from the rules in (2.1)).

Note that, from the rule derived above, it is easy to obtain

$$(q * (q \mathrel{-\!\!*} p)) \Rightarrow ((q * \mathbf{true}) \wedge p),$$

which is the converse of Proposition 4 (without the restriction that q must be strictly exact).

To derive: $\dfrac{}{r \Rightarrow (q \mathrel{-\!\!*} (q * r))}$ (2.6)

1. $(r * q) \Rightarrow (q * r)$ $(p_0 * p_1 \Rightarrow p_1 * p_0)$
2. $r \Rightarrow (q \mathrel{-\!\!*} (q * r))$ (currying, 1)

To derive: $\dfrac{}{(p * r) \Rightarrow (p * (q \mathrel{-\!\!*} (q * r)))}$ (2.7)

1. $p \Rightarrow p$ $(p \Rightarrow p)$
2. $r \Rightarrow (q \mathrel{-\!\!*} (q * r))$ (derived above)
3. $(p * r) \Rightarrow (p * (q \mathrel{-\!\!*} (q * r)))$ (monotonicity, 1, 2)

To derive: $\dfrac{p_0 \Rightarrow (q \mathrel{-\!\!*} r) \qquad p_1 \Rightarrow (r \mathrel{-\!\!*} s)}{p_1 * p_0 \Rightarrow (q \mathrel{-\!\!*} s)}$ (2.8)

1. $p_1 \Rightarrow p_1$ $(p \Rightarrow p)$
2. $p_0 \Rightarrow (q \mathrel{-\!\!*} r)$ (assumption)
3. $p_0 * q \Rightarrow r$ (decurrying, 2)
4. $p_1 * p_0 * q \Rightarrow p_1 * r$ (monotonicity, 1, 3)
5. $p_1 \Rightarrow (r \mathrel{-\!\!*} s)$ (assumption)
6. $p_1 * r \Rightarrow s$ (decurrying, 5)
7. $p_1 * p_0 * q \Rightarrow s$ (trans impl, 4, 6)
8. $p_1 * p_0 \Rightarrow (q \mathrel{-\!\!*} s)$ (currying, 7)

$$\text{To derive:} \qquad \frac{p' \Rightarrow p \qquad q \Rightarrow q'}{(p \mathrel{-\!\!*} q) \Rightarrow (p' \mathrel{-\!\!*} q').} \tag{2.9}$$

1. $(p \mathrel{-\!\!*} q) \Rightarrow (p \mathrel{-\!\!*} q)$ $(p \Rightarrow p)$
2. $p' \Rightarrow p$ (assumption)
3. $(p \mathrel{-\!\!*} q) * p' \Rightarrow (p \mathrel{-\!\!*} q) * p$ (monotonicity, 1, 2)
4. $(p \mathrel{-\!\!*} q) * p \Rightarrow q$ (decurrying, 1)
5. $(p \mathrel{-\!\!*} q) * p' \Rightarrow q$ (trans impl, 3, 4)
6. $q \Rightarrow q'$ (assumption)
7. $(p \mathrel{-\!\!*} q) * p' \Rightarrow q'$ (trans impl, 5, 6)
8. $(p \mathrel{-\!\!*} q) \Rightarrow (p' \mathrel{-\!\!*} q')$ (currying, 7)

Exercise 1

Give a formal proof of the valid assertion

$$(\mathsf{x} \mapsto \mathsf{y} \ * \ \mathsf{x}' \mapsto \mathsf{y}') \ * \ \mathbf{true} \Rightarrow \\ ((\mathsf{x} \mapsto \mathsf{y} \ * \ \mathbf{true}) \wedge (\mathsf{x}' \mapsto \mathsf{y}' \ * \ \mathbf{true})) \wedge \mathsf{x} \neq \mathsf{x}'$$

from the rules in (2.3) and (2.4), and (some of) the following inference rules for predicate calculus (which can be derived from the rules in (2.1) and (2.2)):

$$p \Rightarrow \mathbf{true} \qquad p \Rightarrow p \qquad p \wedge \mathbf{true} \Rightarrow p$$

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r} \quad \text{(trans impl)}$$

$$\frac{p \Rightarrow q \qquad p \Rightarrow r}{p \Rightarrow q \wedge r} \quad (\wedge\text{-introduction})$$

Your proof will be easier to read if you write it as a sequence of steps rather than a tree. In the inference rules, you should regard $*$ as left associative, e.g.,

$$e_0 \mapsto e_0' \ * \ e_1 \mapsto e_1' \ * \ \mathbf{true} \Rightarrow e_0 \neq e_1$$

stands for

$$(e_0 \mapsto e_0' \ * \ e_1 \mapsto e_1') \ * \ \mathbf{true} \Rightarrow e_0 \neq e_1.$$

For brevity, you may weaken $\Leftrightarrow$ to $\Rightarrow$ when it is the main operator of an axiom. You may also omit instances of the axiom schema $p \Rightarrow p$ when it is used as a premiss of the monotonicity rule.

Exercise 2

None of the following axiom schemata are sound. For each, given an instance which is not valid, along with a description of a state in which the instance is false.

$$p_0 \ * \ p_1 \Rightarrow p_0 \wedge p_1 \qquad \text{(unsound)}$$

$$p_0 \wedge p_1 \Rightarrow p_0 \ * \ p_1 \qquad \text{(unsound)}$$

$$(p_0 \ * \ p_1) \vee q \Rightarrow (p_0 \vee q) \ * \ (p_1 \vee q) \qquad \text{(unsound)}$$

$$(p_0 \vee q) \ * \ (p_1 \vee q) \Rightarrow (p_0 \ * \ p_1) \vee q \qquad \text{(unsound)}$$

$$(p_0 \ * \ q) \wedge (p_1 \ * \ q) \Rightarrow (p_0 \wedge p_1) \ * \ q \qquad \text{(unsound)}$$

$$(p_0 \ * \ p_1) \wedge q \Rightarrow (p_0 \wedge q) \ * \ (p_1 \wedge q) \qquad \text{(unsound)}$$

$$(p_0 \wedge q) \ * \ (p_1 \wedge q) \Rightarrow (p_0 \ * \ p_1) \wedge q \qquad \text{(unsound)}$$

$$\big(\forall x.\ (p_0 \ * \ p_1)\big) \Rightarrow (\forall x.\ p_0) \ * \ p_1 \quad \text{when } x \text{ not free in } p_1 \qquad \text{(unsound)}$$

$$(p_0 \Rightarrow p_1) \Rightarrow \big((p_0 \ * \ q) \Rightarrow (p_1 \ * \ q)\big) \qquad \text{(unsound)}$$

$$(p_0 \Rightarrow p_1) \Rightarrow (p_0 \mathrel{-\!\!*} p_1) \qquad \text{(unsound)}$$

$$(p_0 \mathrel{-\!\!*} p_1) \Rightarrow (p_0 \Rightarrow p_1) \qquad \text{(unsound)}$$

An Introduction
to Separation Logic

October 23, 2008

Chapter 3

Specifications

From assertions, we move on to specifications, which describe the behavior of commands. In this chapter, we will define the syntax and meaning of specifications, give and illustrate inference rules for proving valid specifications, and define a compact form of proof called an "annotated specification".

Since separation logic has been built upon it, we will review the basics of Hoare logic. Further descriptions of this logic, including many examples of proofs, have been given by the author [41, Chapters 1 and 2], [45, Chapters 3 and 4]. A more theoretical view appears in [46, Chapter 8].

The original papers by Hoare [3, 4], as well as earlier work by Naur [47] and Floyd [48], are still well worth reading.

3.1 Hoare Triples

For much of these notes, the only kind of specification will be the *Hoare triple*, which consists of two assertions surrounding a command. More precisely, there are two forms of Hoare triple.

A *partial correctness specification*, written

$$\{p\}\ c\ \{q\}$$

is *valid* iff, starting in any state in which the assertion p holds, no execution of the command c aborts and, for any execution of c that terminates in a final state, the assertion q holds in the final state.

A *total correctness specification*, written

$$[\,p\,]\ c\ [\,q\,]$$

is *valid* iff, starting in any state in which p holds, no execution of c aborts, every execution of c terminates, and, for any execution of c that terminates in a final state, q holds in the final state. (In these notes, we will consider total specifications infrequently.)

In both forms, p is called the *precondition* (or *precedent*) and q is called the *postcondition* (or *consequent*).

Notice that, in both forms, there is an implicit universal quantification over both initial and final states. Thus the meaning of a specification is simply **true** or **false**, and does not depend upon a state. So to say that a specification is true is the same as saying that it is valid. (This situation will change when we introduce procedures in Section 4.5.)

The following are examples of valid partial correctness specifications of simple commands:

$$\{\mathsf{x} - \mathsf{y} > 3\}\ \mathsf{x} := \mathsf{x} - \mathsf{y}\ \{\mathsf{x} > 3\}$$

$$\{\mathsf{x} + \mathsf{y} \geq 17\}\ \mathsf{x} := \mathsf{x} + 10\ \{\mathsf{x} + \mathsf{y} \geq 27\}$$

$$\{\mathbf{emp}\}\ \mathsf{x} := \mathbf{cons}(1, 2)\ \{\mathsf{x} \mapsto 1, 2\}$$

$$\{\mathsf{x} \mapsto 1, 2\}\ \mathsf{y} := [\mathsf{x}]\ \{\mathsf{x} \mapsto 1, 2 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1, 2 \wedge \mathsf{y} = 1\}\ [\mathsf{x} + 1] := 3\ \{\mathsf{x} \mapsto 1, 3 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1, 3 \wedge \mathsf{y} = 1\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x} + 1 \mapsto 3 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \leq 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\}$$

$$\{\mathbf{true}\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\} \quad (*)$$

$$\{\mathsf{x} > 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathbf{false}\} \quad (*)$$

A more elaborate example is a specification of the "record-gluing" program (1.5):

$$\begin{array}{l}\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\ \mathbf{if}\ \mathsf{y} = \mathsf{x} + 1\ \mathbf{then\ skip} \\ \quad \mathbf{else\ if}\ \mathsf{x} = \mathsf{y} + 1\ \mathbf{then}\ \mathsf{x} := \mathsf{y} \\ \quad \mathbf{else}\ (\mathbf{dispose}\ \mathsf{x}\ ;\ \mathbf{dispose}\ \mathsf{y}\ ;\ \mathsf{x} := \mathbf{cons}(1, 2)) \\ \{\mathsf{x} \mapsto -, -\}\end{array}$$

(All of the above examples, except those marked $(*)$, would also be valid as total specifications.)

3.2 Hoare's Inference Rules for Specifications

As we did with assertions in Section 2.2, we will reason about specifications using inference rules of the form

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\mathcal{C},}$$

where the premisses and conclusion may contain metavariables, each of which ranges over some set of phrases, such as expressions, variables, or assertions. Again, a rule is said to be *sound* iff, for every instance of the rule, the conclusion of the instance is valid whenever all of the premisses of the instance are valid. But now the premisses may be either specifications or assertions, and the conclusion must be a specification.

In this section, we give the original rules of Hoare logic [3] (along with a more recent rule for variable declarations and several alternative variants of the rules. All of the rules given here remain valid for separation logic. In each case, we give the rule and one or more of its instances:

- Assignment (AS)

$$\frac{}{\{q/v \to e\}\ v := e\ \{q\}}$$

Instances:

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

- Sequential Composition (SQ)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ c_2\ \{r\}}$$

An instance:

$$\frac{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \qquad \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

- Strengthening Precedent (SP)

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\ c\ \{r\}}$$

An instance:

$$\frac{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n} \qquad \{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

In contrast to rules such as Assignment and Sequential Composition, which are called *command-specific rules*, the rules for Strengthening Precedents and Weakening Consequents (to be introduced later) are applicable to arbitrary commands, and are therefore called *structural rules*.

These two rules are exceptional in having premisses, called *verification conditions*, that are assertions rather than specifications. The verification conditions are the mechanism used to introduce mathematical facts about kinds of data, such as $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}$, into proofs of specifications.

To be completely formal, our notion of proof should include formal subproofs of verification conditions, using the rules of predicate calculus as well as rules about integers and other kinds of data. In these notes, however, to avoid becoming mired in proofs of simple arithmetic facts, we will often omit the proofs of verification conditions. It must be emphasized, however, that the soundness of a formal proof can be destroyed by an invalid verification condition.

- Partial Correctness of **while** (WH)

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i \wedge \neg\, b\}}$$

An instance:

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\}\end{array}}$$

Here the assertion i is the *invariant* of the **while** command. This is the one inference rule in this chapter that does not extend to total correctness — reflecting that the **while** command is the one construct in our present programming language that can cause nontermination. (In Section 4.5, however, we will find a similar situation when we introduce recursive procedures.)

- Weakening Consequent (WC)

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\ c\ \{r\}}$$

An instance:

$$\frac{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \\ \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}\end{array}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Notice that $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}$ is another verification condition.

At this stage, we can give a slightly nontrivial example of a formal proof

(of the heart of a simple program for computing powers of two):

1. $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}$ (VC)
2. $\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (AS)
3. $\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (AS)
4. $\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (SQ 2,3)
5. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (SP 1,4)
6. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WH 5)
   while $\mathsf{k} \neq \mathsf{n}$ **do** $(\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})$
   $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\}$
7. $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}$ (VC)
8. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WC 6,7)
   while $\mathsf{k} \neq \mathsf{n}$ **do** $(\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})$
   $\{\mathsf{y} = 2^{\mathsf{n}}\}$

Additional rules describe additional forms of commands:

- **skip** (SK)

$$\frac{}{\{q\}\ \textbf{skip}\ \{q\}}$$

An instance:

$$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\, \mathrm{odd}(\mathsf{k})\}\ \textbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\, \mathrm{odd}(\mathsf{k})\}$$

- Conditional (CD)

$$\frac{\{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \neg\ b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

An instance:

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\} \quad \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\ \textbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}}{\{\mathsf{y} = 2^{\mathsf{k}}\}\ \textbf{if}\ \mathrm{odd}(\mathsf{k})\ \textbf{then}\ \mathsf{k} := \mathsf{k} + 1; \mathsf{y} := 2 \times \mathsf{y}\ \textbf{else skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}}$$

- Variable Declaration (DC)

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\}}$$

  when v does not occur free in p or q.

An instance:

$$\frac{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\ \textbf{newvar}\ \mathsf{k}\ \textbf{in} \\ \quad \big(\mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \quad \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\big) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Here the requirement on the declared variable v formalizes the concept of *locality*, i.e., that the value of v when c begins execution has no effect on this execution, and that the value of v when c finishes execution has no effect on the rest of the program.

Notice that the concept of locality is context-dependent: Whether a variable is local or not depends upon the requirements imposed by the surrounding program, which are described by the specification that reflects these requirements. For example, in the specification

$$\{\textbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2\},$$

t is local, and can be declared at the beginning of the command being specified, but in

$$\{\mathbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2 \wedge \mathsf{t} = (\mathsf{x} + \mathsf{y})\},$$

t is not local, and cannot be declared.

For several of the rules we have given, there are alternative versions. For instance:

- Alternative Rule for Assignment (ASalt)

$$\frac{}{\{p\}\ v := e\ \{\exists v'.\ v = e' \wedge p'\}}$$

  where $v' \notin \{v\} \cup \mathrm{FV}(e) \cup \mathrm{FV}(p)$, e' is $e/v \to v'$, and p' is $p/v \to v'$. The quantifier can be omitted when v does not occur in e or p.

The difficulty with this "forward"rule (due to Floyd [48]) is the accumulation of quantifiers, as in the verification condition in the following proof:

1. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n}\}$ (ASalt)
2. $\{\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}$ (ASalt)
   $\{\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})\}$
3. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}$ (SQ 1)
   $\{\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})\}$
4. $(\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})) \Rightarrow (\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n})$ (VC)
5. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WC 3,4)

(Compare Step 4 with the verification condition in the previous formal proof.)

- Alternative Rule for Conditionals (CDalt)

$$\frac{\{p_1\}\ c_1\ \{q\} \qquad \{p_2\}\ c_2\ \{q\}}{\{(b \Rightarrow p_1) \wedge (\neg\ b \Rightarrow p_2)\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \{q\}}$$

An instance:

$$\frac{\begin{array}{c}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\ \textbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\end{array}}{\begin{array}{l}\{(\mathrm{odd}(\mathsf{k}) \Rightarrow \mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})) \wedge (\neg\,\mathrm{odd}(\mathsf{k}) \Rightarrow \mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k}))\} \\ \textbf{if}\ \mathrm{odd}(\mathsf{k})\ \textbf{then}\ \mathsf{k} := \mathsf{k} + 1; \mathsf{y} := 2 \times \mathsf{y}\ \textbf{else skip} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\end{array}}$$

(A comparison with the instance of (CD) given earlier indicates why (CD) is usually preferable to (CDalt).)

There is also a rule that combines the rules for strengthening precedents and weakening consequences:

- Consequence (CONSEQ)

$$\frac{p \Rightarrow p' \qquad \{p'\}\ c\ \{q'\} \qquad q' \Rightarrow q}{\{p\}\ c\ \{q\}}$$

An instance:

$$\frac{\begin{array}{l}(\mathsf{n} \geq 0) \Rightarrow (1 = 2^0 \wedge 0 \leq \mathsf{n}) \\ \{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n}\} \\ (\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n}) \Rightarrow (\mathsf{y} = 2^{\mathsf{n}})\end{array}}{\begin{array}{l}\{\mathsf{n} \geq 0\} \\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

(One can use (CONSEQ) in place of (SP) and (WC), but the price is an abundance of vacuous verification conditions of the form $p \Rightarrow p$.)

3.3 Annotated Specifications

As was made evident in the previous section, full-blown formal proofs in Hoare logic, even of tiny programs, are long and tedious. Because of this, the usual way of presenting such proofs, at least to human readers, is by means of a specification that has been annotated with intermediate assertions. Several examples of such *annotated specifications* (also often called "proof outlines") have already appeared in Sections 1.5 and 1.10.

The purpose of an annotated specification is to provide enough information so that it would be straightforward for the reader to construct a full formal proof, or at least to determine the verification conditions occuring in the proof. In this section, we will formulate rigorous criteria for achieving this purpose.

In the first place, we note that without annotations, it is not straightforward to construct a proof of a specification from the specification itself. For example, if we try to use the rule for sequential composition,

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ c_2\ \{r\},}$$

to obtain the main step of a proof of the specification

$$\begin{array}{l}\{\mathsf{n} \geq 0\} \\ (\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ; \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\},\end{array}$$

there is no indication of what assertion should replace the metavariable q.

But if we were to change the rule to

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ \underline{\{q\}}\ c_2\ \{r\},}$$

then the new rule would require the annotation q (underlined here for emphasis) to occur in the conclusion:

$$
\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
(\mathsf{k} := 0 \,;\, \mathsf{y} := 1) \,; \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}) \\
\{\mathsf{y} = 2^{\mathsf{n}}\}.
\end{array}
$$

Then, once q has been determined, the premisses must be

$$
\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
(\mathsf{k} := 0 \,;\, \mathsf{y} := 1) \,; \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}
\end{array}
\qquad \text{and} \qquad
\begin{array}{l}
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do} \\
\quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}) \\
\{\mathsf{y} = 2^{\mathsf{n}}\}.
\end{array}
$$

The basic trick is to add annotations to the conclusions of the inference rules so that the conclusion of each rule completely determines its premisses. Fortunately, however, it is not necessary to annotate every semicolon in a command — indeed, in many cases one can even omit the precondition (or occasionally the postcondition) from an annotated specification.

The main reason for this state of affairs is that it is often the case that, for a command c and a postcondition q, one can calculate a *weakest precondition* p_w, which is an assertion such that $\{p\}\ c\ \{q\}$ holds just when $p \Rightarrow p_w$.

Thus, when we can calculate the weakest precondition p_w of c and q, we can regard $c\,\{q\}$ as an annotated specification proving $\{p_w\}\ c\ \{q\}$ — since $c\,\{q\}$ provides enough information to determine p_w.

(We are abusing terminology slightly here. For historical reasons, the term weakest precondition is reserved for total correctness, while the phrase *weakest liberal precondition* is used for partial correctness. In these notes, however, since we will rarely consider total correctness, we will drop the qualification "liberal".)

For instance, the weakest precondition of the assignment command $v := e$ and a postcondition q is $q/v \to e$. Thus we can regard $v := e\{q\}$ as an annotated specification for $\{q/v \to e\}\ v := e\ \{q\}$.

In general, we will write the *annotation description*

$$\mathcal{A} \gg \{p\}\ c\ \{q\},$$

and say that $\mathcal{A}$ *establishes* $\{p\}\ c\ \{q\}$, when $\mathcal{A}$ is an annotated specification that determines the specification $\{p\}\ c\ \{q\}$ and shows how to obtain a formal

proof of this specification. (The letter $\mathcal{A}$, with various decorations, will be a metavariable ranging over annotated specifications and their subphrases.) We will define the valid annotation descriptions by means of inference rules.

For assignment commands, the inference rule is

- Assignment (ASan)

$$\frac{}{v := e\,\{q\} \gg \{q/v \to e\}\; v := e\;\{q\}.}$$

For example, we have the instances

$$\left.\begin{array}{l} \mathsf{y} := 2 \times \mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}\right\} \gg \left\{\begin{array}{l} \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathsf{y} := 2 \times \mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}\right.$$

and

$$\left.\begin{array}{l} \mathsf{k} := \mathsf{k} + 1 \\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}\right\} \gg \left\{\begin{array}{l} \{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\} \\ \mathsf{k} := \mathsf{k} + 1 \\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}. \end{array}\right.$$

In general, we say that an annotated specification is *left-complete* if it begins with a precondition, *right-complete* if it ends with a postcondition, and *complete* if it is both left- and right-complete. Then

$$\left.\begin{array}{r} \mathcal{A} \\ \{p\}\mathcal{A} \\ \mathcal{A}\{q\} \\ \{p\}\mathcal{A}\{q\} \end{array}\right\} \text{ will match } \left\{\begin{array}{l} \text{any annotated specification.} \\ \text{any left-complete annotated specification.} \\ \text{any right-complete annotated specification.} \\ \text{any complete annotated specification.} \end{array}\right.$$

For the sequential composition of commands, we have the rule:

- Sequential Composition (SQan)

$$\frac{\mathcal{A}_1\,\{q\} \gg \{p\}\; c_1\;\{q\} \qquad \mathcal{A}_2 \gg \{q\}\; c_2\;\{r\}}{\mathcal{A}_1\,;\mathcal{A}_2 \gg \{p\}\; c_1\,; c_2\;\{r\}.}$$

Here the right-complete annotated specification $\mathcal{A}_1\,\{q\}$ in the first premiss must end in the postcondition $\{q\}$, which is stripped from this specification

when it occurs within the conclusion. This prevents $\{q\}$ from being duplicated in the conclusion $\mathcal{A}_1 \,;\, \mathcal{A}_2$, or even from occurring there if $\mathcal{A}_2$ is not left-complete.

For example, if we take the above conclusions inferred from (ASan) as premisses, we can infer

$$\left.\begin{array}{l} \mathsf{k} := \mathsf{k}+1 \,;\, \mathsf{y} := 2\times\mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}\right\} \gg \left\{\begin{array}{l} \{2\times\mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\} \\ \mathsf{k} := \mathsf{k}+1 \,;\, \mathsf{y} := 2\times\mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}. \end{array}\right.$$

Next, there is the rule

- Strengthening Precedent (SPan)

$$\frac{p \Rightarrow q \qquad \mathcal{A} \gg \{q\}\ c\ \{r\}}{\{p\}\,\mathcal{A} \gg \{p\}\ c\ \{r\}.}$$

Notice that this rule can be used to make any assertion left-complete (trivially by using the implication $p \Rightarrow p$). As a nontrivial example, if we take the above conclusion inferred from (SQan) as a premiss, along with the valid verification condition

$$(\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}) \Rightarrow (2\times\mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}),$$

we can infer

$$\left.\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\} \\ \mathsf{k} := \mathsf{k}+1 \,;\, \mathsf{y} := 2\times\mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}\right\} \gg \left\{\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\} \\ \mathsf{k} := \mathsf{k}+1 \,;\, \mathsf{y} := 2\times\mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}. \end{array}\right.$$

At this point, the reader may wonder whether intermediate assertions are ever actually needed in annotated specifications. In fact, intermediate assertions, and occasionally other annotations are needed for three reasons:

1. **while** commands and calls of recursive procedures do not always have weakest preconditions that can be expressed in our assertion language.

2. Certain structural inference rules, such as the existential quantification rule (EQ) or the frame rule (FR), do not fit well into the framework of weakest assertions.

3. Intermediate assertions are often needed to simplify verification conditions.

The first of these reasons is illustrated by the rule for the **while** command, whose annotated specifications are required to contain their invariant, immediately before the symbol **while**:

- Partial Correctness of **while** (WHan)

$$\frac{\{i \wedge b\}\ \mathcal{A}\ \{i\} \gg \{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A}) \gg \{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg\, b\}.}$$

(Here, the parentheses in the annotated specification $\{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A})$ are needed to separate postconditions of the body of the **while** command from postconditions of the **while** command itself. A similar usage of parentheses will occur in some other rules for annotation descriptions.)

For example, if we take the above conclusion inferred from (SPan) as a premiss, we can infer

$$\left.\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\\ \quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\end{array}\right\} \gg \left\{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\\ \quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}.\end{array}\right.$$

The annotated specifications that can be concluded from the rule (WHan) are not right-complete. However, one can add the postcondition $i \wedge \neg\, b$, or any assertion implied by $i \wedge \neg\, b$, by using the rule

- Weakening Consequent (WCan)

$$\frac{\mathcal{A} \gg \{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\mathcal{A}\,\{r\} \gg \{p\}\ c\ \{r\},}$$

which will make any annotated specification right-complete.

For example, if we take the above conclusion inferred from (WHan) as a premiss, along with the valid verification condition

$$\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}},$$

we obtain

$$\left.\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do} \\ \quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\} \end{array}\right\} \gg \left\{\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do} \\ \quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}. \end{array}\right.$$

The reader may verify that further applications of the rules (ASan), (SQan), and (SPan) lead to the annotated specification given at the beginning of this section.

There are additional rules for **skip**, conditional commands, and variable declarations:

- Skip (SKan)
$$\frac{}{\textbf{skip}\,\{q\} \gg \{q\}\ \textbf{skip}\ \{q\}.}$$
- Conditional (CDan)
$$\frac{\{p \wedge b\}\ \mathcal{A}_1\ \{q\} \gg \{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \neg\, b\}\ \mathcal{A}_2\ \{q\} \gg \{p \wedge \neg\, b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if } b \textbf{ then } \mathcal{A}_1 \textbf{ else } (\mathcal{A}_2)\ \{q\} \gg \{p\}\ \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\ \{q\}.}$$
- Variable Declaration (DCan)
$$\frac{\{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\}}{\{p\}\ \textbf{newvar } v \textbf{ in } (\mathcal{A})\ \{q\} \gg \{p\}\ \textbf{newvar } v \textbf{ in } c\ \{q\},}$$
when v does not occur free in p or q.

It should be clear that the rules for annotated specifications given in this section allow considerable flexibility in the choice of intermediate assertions. To make proofs clearer, we will often provide more annotations than necessary.

3.4 More Structural Inference Rules

In addition to Strengthening Precedent and Weakening Consequent, there are a number of other structural inference rules of Hoare logic that remain sound for separation logic. In this section and the next, we give a variety of these rules. For the moment, we ignore annotated specifications, and give the simpler rules that suffice for formal proofs.

- Vacuity (VAC)

$$\frac{}{\{\mathbf{false}\}\ c\ \{q\}}$$

This rule is sound because the definition of both partial and total correctness specifications begin with a universal quantification over states satisfying the precondition — and there are no states satisfying **false**. It is useful for characterizing commands that are not executed (often the bodies of **while** commands). For example, one can prove

1. $(\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \mathsf{k} < \mathsf{b}) \Rightarrow \mathbf{false}$ (VC)
2. $\{\mathbf{false}\}$ (VAC)
   $\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k}$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$
3. $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \mathsf{k} < \mathsf{b}\}$ (SP)
   $\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k}$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$
4. $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$ (WH)
   while $\mathsf{k} < \mathsf{b}$ **do** $(\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k})$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \neg\, \mathsf{k} < \mathsf{b}\}$.

- Disjunction (DISJ)

$$\frac{\{p_1\}\ c\ \{q\} \qquad \{p_2\}\ c\ \{q\}}{\{p_1 \vee p_2\}\ c\ \{q\}}$$

For example, consider two (annotated) specifications of the same command:

$\{\mathsf{a} - 1 \leq \mathsf{b}\}$
$\mathsf{s} := 0\ ;\ \mathsf{k} := \mathsf{a} - 1\ ;$
$\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{k}} \mathsf{i} \wedge \mathsf{k} \leq \mathsf{b}\}$
while $\mathsf{k} < \mathsf{b}$ **do**
  $(\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k})$
$\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}$

$\{\mathsf{a} - 1 \geq \mathsf{b}\}$
$\mathsf{s} := 0\ ;\ \mathsf{k} := \mathsf{a} - 1\ ;$
$\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$
while $\mathsf{k} < \mathsf{b}$ **do**
  $(\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k})$
$\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b}\}$
$\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}$.

(Here, the second specification describes the situation where the body of the **while** command is never executed, so that the subspecification of the **while** command can be obtain by using (VAC).) Using these specifications as premisses to (DISJ), we can obtain the main step in

$$
\begin{array}{l}
\{\mathbf{true}\} \\
\{\mathsf{a} - 1 \leq \mathsf{b} \vee \mathsf{a} - 1 \geq \mathsf{b}\} \\
\mathsf{s} := 0 \,;\, \mathsf{k} := \mathsf{a} - 1 \,; \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{\mathsf{s} = \textstyle\sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}.
\end{array}
$$

- Conjunction (CONJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}}$$

  (It should be noted that, in some extensions of separation logic, this rule becomes unsound.)

- Existential Quantification (EQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

  where v is not free in c.

- Universal Quantification (UQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\forall v.\ p\}\ c\ \{\forall v.\ q\},}$$

  where v is not free in c.

When a variable v does not occur free in the command in a specification (as is required of the premisses of the above two rules), it is said to be a *ghost variable* of the specification.

- Substitution (SUB)

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

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

The restrictions on this rule are needed to avoid aliasing. For example, in

$$\{\mathsf{x} = \mathsf{y}\}\ \mathsf{x} := \mathsf{x} + \mathsf{y}\ \{\mathsf{x} = 2 \times \mathsf{y}\},$$

one can substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{w} - 1$ to infer

$$\{\mathsf{z} = 2 \times \mathsf{w} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{w} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{w} - 1)\}.$$

But one cannot substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{z} - 1$ to infer the invalid

$$\{\mathsf{z} = 2 \times \mathsf{z} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{z} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{z} - 1)\}.$$

This rule for substitution will become important when we consider procedures.

- Renaming (RN)

$$\frac{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\}}{\{p\}\ \textbf{newvar}\ v'\ \textbf{in}\ (c/v \to v')\ \{q\},}$$

  where v' does not occur free in c.

Actually, this is not a structural rule, since it is only applicable to variable declarations. On the other hand, unlike the other nonstructural rules, it is not syntax-directed.

The only time it is necessary to use this rule is when one must prove a specification of a variable declaration that violates the proviso that the variable being declared must not occur free in the pre- or postcondition. For example,

1. $\{\mathsf{x} = 0\}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}$ (AS)
2. $\{\mathsf{x} = 0\}\ \textbf{newvar}\ \mathsf{y}\ \textbf{in}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}$ (DC 1)
3. $\{\mathsf{x} = 0\}\ \textbf{newvar}\ \mathsf{x}\ \textbf{in}\ \mathsf{x} := 1\ \{\mathsf{x} = 0\}.$ (RN 2)

In practice, the rule is rarely used. It can usually be avoided by renaming local variables in the program before proving it.

Renaming of bound variables in pre- and postconditions can be treated by using verification conditions such as $(\forall \mathsf{x}.\ \mathsf{z} \neq 2 \times \mathsf{x}) \Rightarrow (\forall \mathsf{y}.\ \mathsf{z} \neq 2 \times \mathsf{y})$ as premisses in the rules (SP) and (WC).

3.5 The Frame Rule

In Section 1.5, we saw that the Hoare-logic Rule of Constancy fails for separation logic, but is replaced by the more general

- Frame Rule (FR)
$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$
where no variable occurring free in r is modified by c.

In fact, the frame rule lies at the heart of separation logic, and provides the key to local reasoning. An instance is

$$\frac{\{\mathbf{list}\ \alpha\ \mathsf{i}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j}\}}{\{\mathbf{list}\ \alpha\ \mathsf{i}\ *\ \mathbf{list}\ \gamma\ \mathsf{x}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j}\ *\ \mathbf{list}\ \gamma\ \mathsf{x}\}},$$

(assuming “Reverse List” does not modify x or γ).

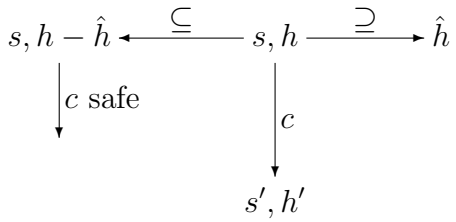
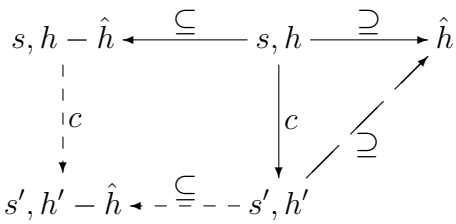
The soundness of the frame rule is surprisingly sensitive to the semantics of our programming language. Suppose, for example, we changed the behavior of deallocation, so that, instead of causing a memory fault, **dispose** x behaved like **skip** when the value of x was not in the domain of the heap. Then $\{\mathbf{emp}\}$ **dispose** x $\{\mathbf{emp}\}$ would be valid, and the frame rule could be used to infer $\{\mathbf{emp}\ *\ \mathsf{x} \mapsto 10\}$ **dispose** x $\{\mathbf{emp}\ *\ \mathsf{x} \mapsto 10\}$. Then, since **emp** is a neutral element for $*$, we would have $\{\mathsf{x} \mapsto 10\}$ **dispose** x $\{\mathsf{x} \mapsto 10\}$, which is patently false.

To reveal the programming-language properties that the frame rule depends upon, we begin with some definitions:

> If, starting in the state s, h, no execution of a command c aborts, then c is *safe at* s, h. If, starting in the state s, h, every execution of c terminates without aborting, then c *must terminate normally* at s, h.

Then the frame rule depends upon two properties of the programming language, which capture the idea that a program will only depend upon or change the part of the initial heap within its footprint, and will abort if any of that part of the heap is missing:

> *Safety Monotonicity* If $\hat{h} \subseteq h$ and c is safe at $s, h - \hat{h}$, then c is safe at s, h. If $\hat{h} \subseteq h$ and c must terminate normally at $s, h - \hat{h}$, then c must terminate normally at s, h.

> *The Frame Property* If $\hat{h} \subseteq h$, c is safe at $s, h - \hat{h}$, and some execution of c starting at s, h terminates normally in the state s', h',
>
> 
>
>
> then $\hat{h} \subseteq h'$ and some execution of c starting at $s, h - \hat{h}$, terminates normally in the state $s', h' - \hat{h}$:
>
> 
>

Then:

Proposition 11 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

PROOF Assume $\{p\}\ c\ \{q\}$ is valid, and $s, h \models p * r$. Then there is an $\hat{h} \subseteq h$ such that $s, h - \hat{h} \models p$ and $s, \hat{h} \models r$.

From $\{p\}\ c\ \{q\}$, we know that c is safe at $s, h - \hat{h}$ (and in the total-correctness case it must terminate normally). Then, by safety monotonicity, we know that c is safe at s, h (and in the total-correctness case it must terminate normally).

Suppose that, starting in the state s, h, there is an execution of c that terminates in the state s', h'. Since c is safe at $s, h - \hat{h}$, we know by the frame property that $\hat{h} \subseteq h'$ and that, starting in the state $s, h - \hat{h}$, there is some execution of c that terminates in the state $s', h' - \hat{h}$. Then $\{p\}\ c\ \{q\}$ and $s, h - \hat{h} \models p$ imply that $s', h' - \hat{h} \models q$.

Since an execution of c carries s, h into s', h', we know that $s\,v = s'v$ for all v that are not modified by c. Then, since these include the free variables of r, $s, \hat{h} \models r$ implies that $s', \hat{h} \models r$. Thus $s', h' \models q * r$. END OF PROOF

3.6 More Rules for Annotated Specifications

We now consider how to enlarge the concept of annotated specifications to encompass the structural rules given in the two preceding sections. Since these rules are applicable to arbitrary commands, their annotated versions will indicate explicitly which rule is being applied.

In all of these rules, we assume that the annotated specifications in the premisses will often be sequences of several lines. In the unary rules (VACan), (EQan), (UQan), (FRan), and (SUBan), braces are used to indicate the vertical extent of the single operand. In the binary rules (DISJan), and (CONJan), the two operands are placed symmetrically around the indicator DISJ or CONJ.

For the vacuity rule, we have

- Vacuity (VACan)

$$\frac{}{\Big\{c\Big\}\mathrm{VAC}\,\{q\} \gg \{\mathbf{false}\}\ c\ \{q\}.}$$

Here c contains no annotations, since no reasoning about its subcommands is used. For example, using (VACan), (SPan), (WHan), and (WCan):

$$\begin{array}{l}\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}\\ \mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do}\\ \quad \left.\begin{array}{l}(\mathsf{k} := \mathsf{k} + 1\,;\\ \mathsf{s} := \mathsf{s} + \mathsf{k})\end{array}\right\}\mathrm{VAC}\\ \{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b}\}.\end{array}$$

Notice that, here and with later unary structural rules, when the braced sequence contains several lines, we will omit the left brace.

For the disjunction rule, we have

- Disjunction (DISJan)

$$\frac{\mathcal{A}_1\,\{q\} \gg \{p_1\}\ c\ \{q\} \qquad \mathcal{A}_2\,\{q\} \gg \{p_2\}\ c\ \{q\}}{(\mathcal{A}_1 \text{ DISJ } \mathcal{A}_2)\,\{q\} \gg \{p_1 \vee p_2\}\ c\ \{q\}.}$$

For example,

$$\{\mathbf{true}\}$$

$$\begin{array}{l}
\{\mathsf{a}-1 \le \mathsf{b}\} \\
\mathsf{s} := 0\,;\, \mathsf{k} := \mathsf{a}-1\,; \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{k}} \mathsf{i} \wedge \mathsf{k} \le \mathsf{b}\} \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k}+1\,;\, \mathsf{s} := \mathsf{s}+\mathsf{k})
\end{array}
\quad \text{DISJ} \quad
\begin{array}{l}
\{\mathsf{a}-1 \ge \mathsf{b}\} \\
\mathsf{s} := 0\,;\, \mathsf{k} := \mathsf{a}-1\,; \\
\{\mathsf{s} = 0 \wedge \mathsf{a}-1 \ge \mathsf{b} \wedge \mathsf{k} \ge \mathsf{b}\} \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k}+1\,;\, \mathsf{s} := \mathsf{s}+\mathsf{k})\Big\}\text{VAC} \\
\{\mathsf{s} = 0 \wedge \mathsf{a}-1 \ge \mathsf{b}\}.
\end{array}$$

$$\{\mathsf{s} = \textstyle\sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}.$$

In the remaining structural rules, the annotated specification in the conclusion need not be left- or right-complete; it simply contains the annotated specifications of the premisses.

- Conjunction (CONJan)

$$\frac{\mathcal{A}_1 \gg \{p_1\}\ c\ \{q_1\} \qquad \mathcal{A}_2 \gg \{p_2\}\ c\ \{q_2\}}{(\mathcal{A}_1 \text{ CONJ } \mathcal{A}_2) \gg \{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}.}$$

- Existential Quantification (EQan)

$$\frac{\mathcal{A} \gg \{p\}\ c\ \{q\}}{\Big\{\mathcal{A}\Big\}\,\exists v\ \gg \{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

  where v is not free in c.

- Universal Quantification (UQan)

$$\frac{\mathcal{A} \gg \{p\}\ c\ \{q\}}{\Big\{\mathcal{A}\Big\}\,\forall v\ \gg \{\forall v.\ p\}\ c\ \{\forall v.\ q\},}$$

  where v is not free in c.

(In using the two rules above, we will often abbreviate $\{\cdots\{\mathcal{A}\}\exists v_1\cdots\}\exists v_n$ by $\{\mathcal{A}\}\exists v_1\ldots,v_n$, and $\{\cdots\{\mathcal{A}\}\forall v_1\cdots\}\forall v_n$ by $\{\mathcal{A}\}\forall v_1\ldots,v_n$.)

- Frame (FRan)

$$\frac{\mathcal{A}\gg\{p\}\ c\ \{q\}}{\{\mathcal{A}\}\ *\ r\ \gg\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

  where no variable occurring free in r is modified by c.

For example,

$$\begin{array}{l}\{\exists \mathsf{j}.\ \mathsf{x}\mapsto -,\mathsf{j}\ *\ \mathsf{list}\,\alpha\,\mathsf{j}\}\\ \left.\begin{array}{l}\{\mathsf{x}\mapsto -\}\\ [\mathsf{x}]:=\mathsf{a}\\ \{\mathsf{x}\mapsto \mathsf{a}\}\end{array}\right\}\ *\ \mathsf{x}+1\mapsto \mathsf{j}\ *\ \mathsf{list}\,\alpha\,\mathsf{j}\Bigg\}\exists\mathsf{j}\\ \{\exists \mathsf{j}.\ \mathsf{x}\mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{list}\,\alpha\,\mathsf{j}\}\end{array}$$

- Substitution (SUBan)

$$\frac{\mathcal{A}\gg\{p\}\ c\ \{q\}}{\{\mathcal{A}\}/\delta\ \gg\{p/\delta\}\ (c/\delta)\ \{q/\delta\}},$$

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

In the conclusion of this rule, $\{\mathcal{A}\}/\delta$ denotes an annotated specification in which "/" and the substitution denoted by δ occur literally, i.e., the substitution is not carried out on $\mathcal{A}$. The total substitution δ may be abbreviated by a partial substitution, but the conditions on what is substituted for the v_i must hold for all $v_1,\ldots,v_n$ occurring free in p, c, or q.

We omit any discussion of the renaming rule (RN), since it is rarely used, and does not lend itself to annotated specifications.

3.7 Inference Rules for Mutation and Disposal

Finally, we come to the new commands for manipulating the heap, which give rise to a surprising variety of inference rules. For each of these four com-

mands, we can give three kinds of inference rule: local, global, and backward-reasoning.

In this and the next two section, we will present these rules and show that, for each type of command, the different rules are all derivable from one another (except for specialized rules that are only applicable in the "nonoverwriting" case). (Some of the deriviations will make use of the inference rules for assertions derived in Section 2.4.)

For mutation commands, we have

- The local form (MUL)

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

- The global form (MUG)

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}}.$$

- The backward-reasoning form (MUBR)

$$\frac{}{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}}.$$

One can derive (MUG) from (MUL) by using the frame rule:

$$\begin{array}{l} \{(e \mapsto -)\ *\ r\} \\ \left.\begin{array}{l} \quad \{e \mapsto -\} \\ \quad [e] := e' \\ \quad \{e \mapsto e'\} \end{array}\right\} *\ r \\ \{(e \mapsto e')\ *\ r\}, \end{array}$$

while to go in the opposite direction it is only necessary to take r to be **emp**:

$$\begin{array}{l} \{e \mapsto -\} \\ \{(e \mapsto -)\ *\ \mathbf{emp}\} \\ [e] := e' \\ \{(e \mapsto e')\ *\ \mathbf{emp}\} \\ \{e \mapsto e'\}. \end{array}$$

To derive (MUBR) from (MUG), we take r to be $(e \mapsto e') \mathrel{-\!\!*} p$ and use the derived axiom schema (2.5): $q \ast (q \mathrel{-\!\!*} p) \Rightarrow p$.

$$\begin{array}{l}\{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!*} p)\}\\ [e] := e'\\ \{(e \mapsto e') \ast ((e \mapsto e') \mathrel{-\!\!*} p)\}\\ \{p\},\end{array}$$

while to go in the opposite direction we take p to be $(e \mapsto e') \ast r$ and use the derived axiom schema (2.7): $(p \ast r) \Rightarrow (p \ast (q \mathrel{-\!\!*} (q \ast r)))$.

$$\begin{array}{l}\{(e \mapsto -) \ast r\}\\ \{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!*} ((e \mapsto e') \ast r))\}\\ [e] := e'\\ \{(e \mapsto e') \ast r\}.\end{array}$$

For deallocation, there are only two rules, since the global form is also suitable for backward reasoning:

- The local form (DISL)

$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}.$$

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

$$\frac{}{\{(e \mapsto -) \ast r\}\ \mathbf{dispose}\ e\ \{r\}}.$$

As with the mutation rules, one can derive (DISBR) from (DISL) by using the frame rule, and go in the opposite direction by taking r to be **emp**.

3.8 Rules for Allocation

When we turn to the inference rules for allocation and lookup, our story becomes more complicated, since these commands modify variables. More precisely, they are what we will call *generalized assignment commands*, i.e., commands that first perform a computation that does not alter the store

(though it may affect the heap), and then after this computation has finished, change the value of the store for a single variable.

However, neither allocation nor lookup are assignment commands in the sense we will use in these notes, since they do not obey the rule (AS) for assignment. For example, if we tried to apply this rule to an allocation command, we could obtain

$$\{\mathbf{cons}(1,2) = \mathbf{cons}(1,2)\}\ \mathsf{x} := \mathbf{cons}(1,2)\ \{\mathsf{x} = \mathsf{x}\}, \qquad \text{(syntactically illegal)}$$

where the precondition is not a syntactically well-formed assertion, since $\mathbf{cons}(1,2)$ is not an expression — for the compelling reason that it has a side effect.

Even in the analogous case for lookup,

$$\{[\mathsf{y}] = [\mathsf{y}]\}\ \mathsf{x} := [\mathsf{y}]\ \{\mathsf{x} = \mathsf{x}\} \qquad \text{(syntactically illegal)}$$

is prohibited since [y] can have the side effect of aborting.

For allocation (and lookup) it is simplest to begin with local and global rules for the *nonoverwriting* case, where the old value of the variable being modified plays no role. For brevity, we abbreviate the sequence $e_1, \ldots, e_n$ of expressions by $\overline{e}$:

- The local nonoverwriting form (CONSNOL)

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

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

- The global nonoverwriting form (CONSNOG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\}},$$

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

As with mutation and deallocation, one can derive the global form from the local by using the frame rule, and the local from the global by taking r to be **emp**.

The price for the simplicity of the above rules is the prohibition of overwriting, which is expressed by the conditions $v \notin \mathrm{FV}(\overline{e})$ and $v \notin \mathrm{FV}(\overline{e}, r)$. Turning to the more complex and general rules that permit overwriting, we have three forms for allocation:

- The local form (CONSL)

$$\overline{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\}},$$

  where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$ (i.e., each e'_i denotes $e_i/v \to v'$).

- The global form (CONSG)

$$\overline{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

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

- The backward-reasoning form (CONSBR)

$$\overline{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\}},$$

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

To explain these rules, we begin with (CONSG). Here the existentially quantified variable v' denotes the old value of v, which has been overwritten by the allocation command (and may possibly no longer be determined by the store), much as in the alternative assignment rule (ASalt) in Section 3.2. A typical instance is

$$\{\mathsf{list}\ \alpha\ \mathsf{i}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}\}.$$

One can derive (CONSG) from the nonoverwriting rule (CONSNOG) by using a plausible equivalence that captures the essence of generalized assignment:

$$v := \mathbf{cons}(\overline{e}) \cong \mathbf{newvar}\ \hat{v}\ \mathbf{in}\ (\hat{v} := \mathbf{cons}(\overline{e})\ ;\ v := \hat{v}), \tag{3.1}$$

where $\hat{v}$ does not occur in $\overline{e}$ — and can be chosen not to occur in other specified phrases. (Here $\cong$ denotes an equivalence of meaning between two commands.)

We can regard this equivalence as defining the possibly overwriting case of allocation on the left by the nonoverwriting case on the right. Then we can derive (CONSG) from (CONSNOG) by existentially quantifying v and

renaming it to v' (using the last axiom schema displayed in 2.2 in Section 2.2: $(p/v \to e) \Rightarrow (\exists v.\ p)$).

$$
\begin{array}{l}
\{r\} \\
\textbf{newvar } \hat{v} \textbf{ in} \\
\quad \big(\hat{v} := \textbf{cons}(\overline{e})\ ; \\
\quad \{(\hat{v} \mapsto \overline{e})\ *\ r\} \\
\quad \{\exists v.\ (\hat{v} \mapsto \overline{e})\ *\ r\} \\
\quad \{\exists v'.\ (\hat{v} \mapsto \overline{e}')\ *\ r'\} \\
\quad v := \hat{v}\big) \\
\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}
\end{array}
$$

One might expect the local rule to be

$$\frac{}{\{\textbf{emp}\}\ v := \textbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\}},$$

(where v' is distinct from v and $v' \notin \mathrm{FV}(\overline{e})$), which can be derived from (CONSG) by taking r to be **emp**. But this rule, though sound, is too weak. For example, the postcondition of the instance

$$\{\textbf{emp}\}\ \mathsf{i} := \textbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\}$$

gives no information about the second component of the new record.

In the stronger local rule (CONSL), the existential quantifier is dropped and v' becomes a variable that is not modified by $v := \textbf{cons}(\overline{e})$, so that its occurrences in the postcondition denote the same value as in the precondition.

For example, the following instance of (CONSL)

$$\{\mathsf{i} = \mathsf{j} \wedge \textbf{emp}\}\ \mathsf{i} := \textbf{cons}(3, \mathsf{i})\ \{\mathsf{i} \mapsto 3, \mathsf{j}\}$$

shows that the value of j in the postcondition is the value of i before the assignment.

We can derive (CONSL) from (CONSG) by replacing r by $v = v' \wedge \textbf{emp}$ and v' by v'', and using the fact that $v'' = v'$ is pure, plus simple properties

of equality and the existential quantifier:

$$\begin{array}{l}
\{v = v' \wedge \mathbf{emp}\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v''.\ (v \mapsto \overline{e}'')\ *\ (v'' = v' \wedge \mathbf{emp})\} \\
\{\exists v''.\ ((v \mapsto \overline{e}'') \wedge v'' = v')\ *\ \mathbf{emp}\} \\
\{\exists v''.\ (v \mapsto \overline{e}'') \wedge v'' = v'\} \\
\{\exists v''.\ (v \mapsto \overline{e}') \wedge v'' = v'\} \\
\{v \mapsto \overline{e}'\}.
\end{array}$$

(Here v'' is chosen to be distinct from v, v', and the free variables of $\overline{e}$.)

Then to complete the circle (and show the adequacy of (CONSL)), we derive (CONSG) from (CONSL) by using the frame rule and (EQ):

$$\begin{array}{l}
\{r\} \\
\{\exists v'.\ v = v' \wedge r\} \\
\quad \left.\begin{array}{l}
\{v = v' \wedge r\} \\
\{v = v' \wedge r'\} \\
\{v = v' \wedge (\mathbf{emp}\ *\ r')\} \\
\{(v = v' \wedge \mathbf{emp})\ *\ r'\} \\
\quad \left.\begin{array}{l}
\{(v = v' \wedge \mathbf{emp})\} \\
v := \mathbf{cons}(\overline{e}) \\
\{(v \mapsto \overline{e}')\}
\end{array}\right\}\ *\ r' \\
\{(v \mapsto \overline{e}')\ *\ r'\}
\end{array}\right\}\ \exists v' \\
\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}
\end{array}$$

In the backward-reasoning rule (CONSBR), the universal quantifier $\forall v''$ in the precondition expresses the nondeterminacy of allocation. (We have chosen the metavariable v'' rather than v' to simplify some deriviations.) The most direct way to see this is by a semantic proof that the rule is sound:

Suppose the precondition holds in the state s, h, i.e., that

$$s, h \models \forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''.$$

Then the semantics of universal quantification gives

$$\forall \ell.\ [\, s \mid v'' \colon \ell \,], h \models (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p'',$$

and the semantics of separating implication gives

$$\forall \ell, h'.\ h \perp h' \text{ and } \underline{[\, s \mid v'' \colon \ell \,], h' \models (v'' \mapsto \overline{e})} \text{ implies } [\, s \mid v'' \colon \ell \,], h \cdot h' \models p'',$$

where the underlined formula is equivalent to

$$h' = [\, \ell \colon [\![e_1]\!]_{\text{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\text{exp}} s \,].$$

Thus

$$\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h \text{ implies } [\, s \mid v'' \colon \ell \,], [\, h \mid \ell \colon [\![e_1]\!]_{\text{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\text{exp}} s \,] \models p''\big).$$

Then, by Proposition 3 in Chapter 2, since p'' denotes $p/v \to v''$, we have $[\, s \mid v'' \colon \ell \,], h' \models p''$ iff $\hat{s}, h' \models p$, where

$$\hat{s} = [\, s \mid v'' \colon \ell \mid v \colon [\![v'']\!]_{\text{exp}} [\, s \mid v'' \colon \ell \,] \,] = [\, s \mid v'' \colon \ell \mid v \colon \ell \,].$$

Moreover, since v'' does not occur free in p, we can simplify $\hat{s}, h' \models p$ to $[\, s \mid v \colon \ell \,], h' \models p$. Thus

$$\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h \text{ implies } [\, s \mid v \colon \ell \,], [\, h \mid \ell \colon [\![e_1]\!]_{\text{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\text{exp}} s \,] \models p\big).$$

Now execution of the allocation command $v := \mathbf{cons}(\overline{e})$, starting in the state s, h, will never abort, and will always termininate in a state $[\, s \mid v \colon \ell \,], [\, h \mid \ell \colon [\![e_1]\!]_{\text{exp}} s \mid \ldots \mid \ell + n - 1 \colon [\![e_n]\!]_{\text{exp}} s \,]$ for some ℓ such that $\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h$. Thus the condition displayed above insures that all possible terminating states satisfy the postcondition p.

We also show that (CONSBR) and (CONSG) are interderivable. To derive (CONSBR) from (CONSG), we choose $v' \notin \text{FV}(\overline{e}, p)$ to be distinct from v and v'', take r to be $\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''$, and use predicate-calculus properties of quantifiers, as well as (2.5): $q * (q \mathrel{-\!\!*} p) \Rightarrow p$.

$$\begin{array}{l}
\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v'.\ (v \mapsto \overline{e}') * (\forall v''.\ (v'' \mapsto \overline{e}') \mathrel{-\!\!*} p'')\} \\
\{\exists v'.\ (v \mapsto \overline{e}') * ((v \mapsto \overline{e}') \mathrel{-\!\!*} p)\} \\
\{\exists v'.\ p\} \\
\{p\}.
\end{array}$$

To go in the other direction, we choose $v'' \notin \mathrm{FV}(\overline{e}, r)$ to be distinct from v and v', take p to be $\exists v'.\ (v \mapsto \overline{e}')\ *\ r'$, and use properties of quantifiers, as well as (2.6): $r \Rightarrow (q \mathrel{-\!\!*} (q\ *\ r))$.

$$
\begin{array}{l}
\{r\} \\
\{\forall v''.\ r\} \\
\{\forall v''.\ (v'' \mapsto \overline{e})\ \mathrel{-\!\!*}\ ((v'' \mapsto \overline{e})\ *\ r)\} \\
\{\forall v''.\ (v'' \mapsto \overline{e})\ \mathrel{-\!\!*}\ (((v'' \mapsto \overline{e}')\ *\ r')/v' \to v)\} \\
\{\forall v''.\ (v'' \mapsto \overline{e})\ \mathrel{-\!\!*}\ (\exists v'.\ (v'' \mapsto \overline{e}')\ *\ r')\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}.
\end{array}
$$

3.9 Rules for Lookup

Finally, we come to the lookup command, which — for no obvious reason — has the richest variety of inference rules. We begin with the nonoverwriting rules:

- The local nonoverwriting form (LKNOL)

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

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

- The global nonoverwriting form (LKNOG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\ v := [e]\ \{(e \mapsto v)\ *\ p\}},$$

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

In (LKNOG), there is no restriction preventing v'' from being the same variable as v. Thus, as a special case,

$$\frac{}{\{\exists v.\ (e \mapsto v)\ *\ p\}\ v := [e]\ \{(e \mapsto v)\ *\ p\}},$$

where $v \notin \mathrm{FV}(e)$. For example, if we take

$$
\begin{array}{llll}
v & \text{to be} & \mathsf{j} & \\
e & \text{to be} & \mathsf{i} + 1 & \qquad p \quad \text{to be} \quad \mathsf{i} \mapsto 3\ *\ \mathbf{list}\ \alpha\ \mathsf{j},
\end{array}
$$

(and remember that $\mathsf{i} \mapsto 3, \mathsf{j}$ abbreviates $(\mathsf{i} \mapsto 3) \ * \ (\mathsf{i}+1 \mapsto \mathsf{j})$), then we obtain the instance

$$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j} \ * \ \mathbf{list}\ \alpha\ \mathsf{j}\}$$
$$\mathsf{j} := [\mathsf{i}+1]$$
$$\{\mathsf{i} \mapsto 3, \mathsf{j} \ * \ \mathbf{list}\ \alpha\ \mathsf{j}\}.$$

In effect, the action of the lookup command is to erase an existential quantifier. In practice, if one chooses the names of quantified variables with foresight, most lookup commands can be described by this simple special case.

Turning to the rules for the general case of lookup, we have

- The local form (LKL)

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

  where v, v', and v'' are distinct, and e' denotes $e/v \rightarrow v'$.

- The global form (LKG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'') \ * \ (r/v' \rightarrow v)\}\ v := [e]\ \{\exists v'.\ (e' \mapsto v) \ * \ (r/v'' \rightarrow v)\}},$$

  where v, v', and v'' are distinct, $v', v'' \notin \mathrm{FV}(e)$, $v \notin \mathrm{FV}(r)$, and e' denotes $e/v \rightarrow v'$.

- The first backward-reasoning form (LKBR1)

$$\frac{}{\{\exists v''.\ (e \mapsto v'') \ * \ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\}},$$

  where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \rightarrow v''$.

- The second backward-reasoning form (LKBR2)

$$\frac{}{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\}},$$

  where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \rightarrow v''$.

In each of these rules, one can think of v' as denoting the value of v before execution of the lookup command, and v'' as denoting the value of v after execution.

We begin with a semantic proof of the soundness of the local rule (LKL). Suppose that the precondition holds in the state s_0, h, i.e., that

$$s_0, h \models v = v' \wedge (e \mapsto v'').$$

Then $s_0\, v = s_0\, v'$ and $h = [\, [\![e]\!]_{\mathrm{exp}} s_0 \colon s_0\, v'' \,]$.

Starting in the state s_0, h, the execution of $v := [e]$ will not abort (since $[\![e]\!]_{\mathrm{exp}} s_0 \in \mathrm{dom}\, h$), and will terminate with the store $s_1 = [\, s_0 \mid v \colon s_0\, v'' \,]$ and the unchanged heap h. To see that this state satisfies the postcondition, we note that $s_1\, v = s_0\, v'' = s_1\, v''$ and, since e' does not contain v, $[\![e']\!]_{\mathrm{exp}} s_1 = [\![e']\!]_{\mathrm{exp}} s_0$. Then by applying Proposition 3 in Chapter 2, with $\hat{s} = [\, s_0 \mid v \colon s_0\, v' \,] = [\, s_0 \mid v \colon s_0\, v \,] = s_0$, we obtain $[\![e']\!]_{\mathrm{exp}} s_0 = [\![e]\!]_{\mathrm{exp}} s_0$. Thus $h = [\, [\![e']\!]_{\mathrm{exp}} s_1 \colon s_1\, v \,]$, and $s_1, h \models v = v'' \wedge (e' \mapsto v)$.

To derive (LKG) from (LKL), we use the frame rule and two applications of (EQ):

$$\begin{array}{l}
\{\exists v''.\ (e \mapsto v'') \ *\ (r / v' \to v)\} \\
\{\exists v', v''.\ (v = v' \wedge (e \mapsto v'')) \ *\ (r / v' \to v)\} \\
\quad \left.\begin{array}{l}
\{\exists v''.\ (v = v' \wedge (e \mapsto v'')) \ *\ (r / v' \to v)\} \\
\quad \left.\begin{array}{l}
\{(v = v' \wedge (e \mapsto v'')) \ *\ (r / v' \to v)\} \\
\quad \left.\begin{array}{l}
\{v = v' \wedge (e \mapsto v'')\} \\
v := [e] \\
\{v = v'' \wedge (e' \mapsto v)\}
\end{array}\right\} *\ r \\
\quad \{(v = v'' \wedge (e' \mapsto v)) \ *\ (r / v'' \to v)\}
\end{array}\right\} \exists v'' \\
\quad \{\exists v''.\ (v = v'' \wedge (e' \mapsto v)) \ *\ (r / v'' \to v)\}
\end{array}\right\} \exists v' \\
\{\exists v', v''.\ (v = v'' \wedge (e' \mapsto v)) \ *\ (r / v'' \to v)\} \\
\{\exists v'.\ (e' \mapsto v) \ *\ (r / v'' \to v)\}.
\end{array}$$

The global form (LKG) is the most commonly used of the rules that allow overwriting. As an example of an instance, if we take

$$\begin{array}{rcl} v & \text{to be} & \mathsf{j} \\ v' & \text{to be} & \mathsf{m} \\ v'' & \text{to be} & \mathsf{k} \end{array} \qquad \begin{array}{rcl} e & \text{to be} & \mathsf{j} + 1 \\ r & \text{to be} & \mathsf{i} + 1 \mapsto \mathsf{m} \ *\ \mathsf{k} + 1 \mapsto \mathbf{nil}, \end{array}$$

then we obtain (with a little use of the commutivity of $*$)

$$\begin{array}{l}\{\exists \mathsf{k}.\ \mathsf{i}+1 \mapsto \mathsf{j} \ *\ \mathsf{j}+1 \mapsto \mathsf{k} \ *\ \mathsf{k}+1 \mapsto \mathbf{nil}\}\\ \mathsf{j} := [\mathsf{j}+1]\\ \{\exists \mathsf{m}.\ \mathsf{i}+1 \mapsto \mathsf{m} \ *\ \mathsf{m}+1 \mapsto \mathsf{j} \ *\ \mathsf{j}+1 \mapsto \mathbf{nil}\}.\end{array}$$

To derive (LKL) from (LKG), we first rename the variables v' and v'' in (LKG) to be $\hat{v}'$ and $\hat{v}''$, chosen not to occur free in e, and then replace r in (LKG) by $\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp}$. Then we use the purity of equality, and predicate-calculus properties of equality and the existential quantifier, to obtain

$$\begin{array}{l}\{v = v' \wedge (e \mapsto v'')\}\\ \{\exists \hat{v}''.\ v = v' \wedge \hat{v}'' = v'' \wedge (e \mapsto \hat{v}'')\}\\ \{\exists \hat{v}''.\ (e \mapsto \hat{v}'') \ *\ (v = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})\}\\ \{\exists \hat{v}''.\ (e \mapsto \hat{v}'') \ *\ ((\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})/\hat{v}' \to v)\}\\ v := [e]\\ \{\exists \hat{v}'.\ (\hat{e}' \mapsto v) \ *\ ((\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})/\hat{v}'' \to v)\}\\ \{\exists \hat{v}'.\ (\hat{e}' \mapsto v) \ *\ (\hat{v}' = v' \wedge v = v'' \wedge \mathbf{emp})\}\\ \{\exists \hat{v}'.\ \hat{v}' = v' \wedge v = v'' \wedge (\hat{e}' \mapsto v)\}\\ \{v = v'' \wedge (e' \mapsto v)\}\end{array}$$

(where $\hat{e}'$ denotes $e/v \to \hat{v}'$).

Turning to the backward-reasoning rules, we will derive (LKBR1) from (LKG). The reasoning here is tricky. We first derive a variant of (LKBR1) in which the variable v'' is renamed to a fresh variable $\hat{v}''$ that is distinct from v. Specifically, we assume $\hat{v}'' \neq v$, $\hat{v}'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and $\hat{p}'' = p/v \to \hat{v}''$. We also take v' to be a fresh variable, and take r to be $(e' \mapsto \hat{v}'') \mathrel{-\!\!*} \hat{p}''$, where $e' = e/v \to v'$. Then, using (LKG) and the axiom schema (2.5) $q \ *\ (q \mathrel{-\!\!*} p) \Rightarrow p$, we obtain

$$\begin{array}{l}\{\exists \hat{v}''.\ (e \mapsto \hat{v}'') \ *\ ((e \mapsto \hat{v}'') \mathrel{-\!\!*} \hat{p}'')\}\\ v := [e]\\ \{\exists v'.\ (e' \mapsto v) \ *\ ((e' \mapsto v) \mathrel{-\!\!*} p)\}\\ \{\exists v'.\ p\}\\ \{p\},\end{array}$$

Now we consider (LKBR1) itself. The side condition $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$ implies $v'' \notin \mathrm{FV}(e)$ and, since $\hat{v}''$ is fresh, $v'' \notin \hat{p}''$. This allows us to rename $\hat{v}''$ to v'' in the first line of the proof, to obtain a proof of (LKBR1).

To derive (LKBR2) from (LKBR1), we use the last axiom schema in (2.4): $(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)$.

$$
\begin{array}{l}
\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\} \\
\{\exists v''.\ (e \mapsto v'') \ * \ ((e \mapsto v'') \mathrel{-\!\!*} p'')\} \\
v := [e] \\
\{p\}.
\end{array}
$$

Then to derive (LKL) from (LKBR2), we rename v'' to $\hat{v}$ in the precondition of (LKBR2), take p to be $v = v'' \wedge (e' \mapsto v)$, and use properties of $\hookrightarrow$, equality, and the existential quantifier:

$$
\begin{array}{l}
\{v = v' \wedge (e \mapsto v'')\} \\
\{v = v' \wedge (e \hookrightarrow v'') \wedge (e' \mapsto v'')\} \\
\{(e \hookrightarrow v'') \wedge (e' \mapsto v'')\} \\
\{\exists \hat{v}.\ (e \hookrightarrow v'') \wedge \hat{v} = v'' \wedge (e' \mapsto v'')\} \\
\{\exists \hat{v}.\ (e \hookrightarrow \hat{v}) \wedge \hat{v} = v'' \wedge (e' \mapsto \hat{v})\} \\
v := [e] \\
\{v = v'' \wedge (e' \mapsto v)\}.
\end{array}
$$

The reader may verify that we have given more than enough derivations to establish that all of the rules for lookup that permit overwriting are interderivable.

For good measure, we also derive the global nonoverwriting rule (LKNOG) from (LKG): Suppose v and v'' are distinct variables, $v \notin \mathrm{FV}(e)$, and $v'' \notin \mathrm{FV}(e) \cup \mathrm{FV}(p)$. We take v' to be a variable distinct from v and v'' that does not occur free in e or p, and r to be $p'' = p/v \to v''$. (Note that $v \notin \mathrm{FV}(e)$

implies that $e' = e$.) Then

$$\begin{array}{l}\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\\ \{\exists v''.\ (e \mapsto v'')\ *\ (p''/v' \to v)\}\\ v := [e]\\ \{\exists v'.\ (e' \mapsto v)\ *\ (p''/v'' \to v)\}\\ \{\exists v'.\ (e \mapsto v)\ *\ p\}\\ \{(e \mapsto v)\ *\ p\}.\end{array}$$

In the first line, we can rename the quantified variable v'' to be any variable not in $\mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, so that v'' does not need to be distinct from v.

3.10 Annotated Specifications for the New Rules

It is straightforward to give annotated specifications for the backward-reasoning rules for mutation, disposal, allocation, and lookup; as with all backward-reasoning rules, these annotated specifications do not have explicit preconditions. Thus (MUBR), (DISBR), (CONSBR), and (LKBR1) lead to:

- Mutation (MUBRan)

$$\frac{}{[e] := e'\ \{p\} \gg \{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}}.$$

- Disposal (DISBRan)

$$\frac{}{\mathbf{dispose}\ e\ \{r\} \gg \{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}}.$$

- Allocation (CONSBRan)

$$\frac{}{v := \mathbf{cons}(\overline{e})\ \{p\} \gg \{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\}},$$

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

- Lookup (LKBR1an)

$$\frac{}{v := [e]\ \{p\} \gg \{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\}},$$

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

Moreover, since the implicit preconditions of these rules are weakest preconditions, the rules can be used to derive annotations (with explicit preconditions) for the other forms of the heap-manipulating rules. For example, by taking p in (MUBRan) to be $e \mapsto e'$, and using the valid verification condition

$$\mathrm{VC} = (e \mapsto -) \Rightarrow (e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} (e \mapsto e')),$$

we may use (SPan) to obtain a proof

$$\dfrac{\mathrm{VC} \quad \overline{[e] := e' \, \{e \mapsto e'\} \gg \{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} (e \mapsto e'))\} \, [e] := e' \, \{e \mapsto e'\}}}{\{e \mapsto -\} \, [e] := e' \, \{e \mapsto e'\} \gg \{e \mapsto -\} \, [e] := e' \, \{e \mapsto e'\}}$$

of an annotation description corresponding to the local form (MUL).

In such a manner, one may derive rules of the form

$$\dfrac{}{\{p\} \ c \ \{q\} \gg \{p\} \ c \ \{q\}}$$

corresponding to each of rules (MUL), (MUG), (DISL), (CONSNOL), (CONSNOG), (CONSL), (CONSG), (LKNOL), (LKNOG), (LKL), (LKG), and (LKBR2).

3.11 A Final Example

In conclusion, we reprise the annotated specification given at the end of Section 1.5, this time indicating the particular inference rules and verification conditions that are used. (To make the action of the inference rules clear, we have also given the unabbreviated form of each assertion.)

$\{\mathbf{emp}\}$

$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a})$; (CONSNOL)

$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\}$ i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{a}\}$

$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b})$; (CONSNOG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) * (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{a} * \mathsf{y} \mapsto \mathsf{b} * \mathsf{y}+1 \mapsto \mathsf{b}\}$

$(p/v \to e \Rightarrow \exists v.\ p)$

$\{(\mathsf{x} \mapsto \mathsf{a}, -) * (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * (\exists \mathsf{a}.\ \mathsf{x}+1 \mapsto \mathsf{a}) * \mathsf{y} \mapsto \mathsf{b} * \mathsf{y}+1 \mapsto \mathsf{b}\}$

$[\mathsf{x}+1] := \mathsf{y} - \mathsf{x}$; (MUG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y}-\mathsf{x}) * (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{y}-\mathsf{x} * \mathsf{y} \mapsto \mathsf{b} * \mathsf{y}+1 \mapsto \mathsf{b}\}$

$(p/v \to e \Rightarrow \exists v.\ p)$

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y}-\mathsf{x}) * (\mathsf{y} \mapsto \mathsf{b}, -)\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{y}-\mathsf{x} * \mathsf{y} \mapsto \mathsf{b} * (\exists \mathsf{b}.\ \mathsf{y}+1 \mapsto \mathsf{b})\}$

$[\mathsf{y}+1] := \mathsf{x} - \mathsf{y}$; (MUG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y}-\mathsf{x}) * (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x}-\mathsf{y})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{y}-\mathsf{x} * \mathsf{y} \mapsto \mathsf{b} * \mathsf{y}+1 \mapsto \mathsf{x}-\mathsf{y}\}$

$(x - y = -(y - x))$

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y}-\mathsf{x}) * (\mathsf{y} \mapsto \mathsf{b}, -(\mathsf{y}-\mathsf{x}))\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{y}-\mathsf{x} * \mathsf{y} \mapsto \mathsf{b} * \mathsf{y}+1 \mapsto -(\mathsf{y}-\mathsf{x})\}$

$(p/v \to e \Rightarrow \exists v.\ p)$

$\{\exists \mathsf{o}.\ (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) * (\mathsf{x}+\mathsf{o} \mapsto \mathsf{b}, -\mathsf{o})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} * \mathsf{x}+1 \mapsto \mathsf{o} * \mathsf{x}+\mathsf{o} \mapsto \mathsf{b} * \mathsf{x}+\mathsf{o}+1 \mapsto -\mathsf{o}\}$

3.12 More about Annotated Specifications

In this section, we will show the essential property of annotated specifications: From a formal proof of a specification one can derive an annotated version of the specification, from which one can reconstruct a similar (though perhaps not identical) formal proof.

We begin by defining functions that map annotation descriptions into their underlying specifications and annotated specifications:

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

$$\text{erase-spec}(\mathcal{A} \gg \{p\}\ c\ \{q\}) \stackrel{\text{def}}{=} \mathcal{A}.$$

We extend these functions to inference rules and proofs of annotation descriptions, in which case they are applied to all of the annotation descriptions in the inference rule or proof (while leaving verification conditions unchanged).

We also define a function "concl" that maps proofs (of either specifications or annotation descriptions) into their conclusions.

Then we define a function cd that acts on an annotated specification $\mathcal{A}$ by deleting annotations to produce the command imbedded within $\mathcal{A}$. For most forms of annotation, the definition is obvious, but in the case of DISJ and CONJ, it relies on the fact that the commands imbedded in the subannotations must be identical:

$$\text{cd}(\mathcal{A}_1\,\text{DISJ}\,\mathcal{A}_2) = \text{cd}(\mathcal{A}_1) = \text{cd}(\mathcal{A}_2)$$

$$\text{cd}(\mathcal{A}_1\,\text{CONJ}\,\mathcal{A}_2) = \text{cd}(\mathcal{A}_1) = \text{cd}(\mathcal{A}_2).$$

In the case of an annotation for the substitution rule, the substitution is carried out:

$$\text{cd}(\{\!|\mathcal{A}|\!\}/\delta) = \text{cd}(\mathcal{A})/\delta.$$

Proposition 12 *1. The function* erase-annspec *maps each inference rule* (Xan) *into the rule* (X)*. (We are ignoring the alternative rules* (ASalt)*,* (CDalt)*, and* (CONSEQ)*.)*

2. The function erase-annspec *maps proofs of annotation descriptions into proofs of specifications.*

3. Suppose $\mathcal{A} \gg \{p\}\ c\ \{q\}$ is a provable annotation description. Then $\text{cd}(\mathcal{A}) = c$. Moreover, if $\mathcal{A}$ is left-complete, then it begins with $\{p\}$, and if $\mathcal{A}$ is right-complete then it ends with $\{q\}$.

PROOF The proof of (1) is by case analysis on the individual pairs of rules. The proofs of (2) and (3) are straightforward inductions on the structure of proofs of annotation descriptions. END OF PROOF

We use $\mathcal{P}$ or $\mathcal{Q}$ (with occasional decorations) as variables that range over proofs of specifications or of annotation descriptions. We also use the notations $\mathcal{P}[\{p\}\ c\ \{q\}]$ or $\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]$ as variables whose range is limited to proofs with a particular conclusion.

Next, we introduce three endofunctions on proofs of annotation descriptions that force annotations to be complete by introducing vacuous instances of the rules (SP) and (WC). If $\mathcal{A}$ is left-complete, then

$$\text{left-compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]) = \mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

otherwise,

$$\text{left-compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]) = \frac{p \Rightarrow p \quad \mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]}{\{p\}\mathcal{A} \gg \{p\}\ c\ \{q\}.}$$

Similarly, if $\mathcal{A}$ is right-complete, then

$$\text{right-compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]) = \mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

otherwise,

$$\text{right-compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]) = \frac{\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}] \quad q \Rightarrow q}{\mathcal{A}\{q\} \gg \{p\}\ c\ \{q\}.}$$

Then, combining these functions,

$$\text{compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}]) = \text{left-compl}(\text{right-compl}(\mathcal{Q}[\mathcal{A} \gg \{p\}\ c\ \{q\}])).$$

Now we can define a function Φ, mapping proofs of specifications into proofs of annotation descriptions, that satisfies

Proposition 13 *1. If $\mathcal{P}$ proves $\{p\}\ c\ \{q\}$, then $\Phi(\mathcal{P})$ proves $\mathcal{A} \gg \{p\}\ c\ \{q\}$ for some annotated specification $\mathcal{A}$.*

2. erase-annspec($\Phi(\mathcal{P})$) *is similar to $\mathcal{P}$, except for the possible insertion of instances of* (SP) *and* (WC) *in which the verification condition is a trivial implication of the form $p \Rightarrow p$.*

Note that Part 2 of this proposition implies that, except for extra implications of the form $p \Rightarrow p$, erase-annspec($\Phi(\mathcal{P})$) contains the same verification conditions as $\mathcal{P}$.

We define $\Phi(\mathcal{P})$ by induction on the structure of $\mathcal{P}$, with a case analysis on the final rule used to infer the conclusion of $\mathcal{P}$. The above proposition is proved concurrently (and straightforwardly) by the same induction and case analysis.

(AS) If $\mathcal{P}$ is

$$\frac{}{\{q/v \to e\}\ v := e\ \{q\},}$$

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

$$\frac{}{v := e\,\{q\} \gg \{q/v \to e\}\ v := e\ \{q\}.}$$

(SP) If $\mathcal{P}$ is

$$\frac{p \Rightarrow q \qquad \mathcal{P}'[\{q\}\ c\ \{r\}]}{\{p\}\ c\ \{r\},}$$

and

$$\Phi(\mathcal{P}'[\{q\}\ c\ \{r\}]) = \mathcal{Q}'[\mathcal{A} \gg \{q\}\ c\ \{r\}],$$

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

$$\frac{p \Rightarrow q \qquad \mathcal{Q}'[\mathcal{A} \gg \{q\}\ c\ \{r\}]}{\{p\}\,\mathcal{A} \gg \{p\}\ c\ \{r\}.}$$

(WC) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{p\}\ c\ \{q\}] \qquad q \Rightarrow r}{\{p\}\ c\ \{r\},}$$

and

$$\Phi(\mathcal{P}'[\{p\}\ c\ \{q\}]) = \mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}] \qquad q \Rightarrow r}{\mathcal{A}\,\{r\} \gg \{p\}\ c\ \{r\}.}$$

(SQ) If $\mathcal{P}$ is

$$\frac{\mathcal{P}_1[\{p\}\ c_1\ \{q\}] \qquad \mathcal{P}_2[\{q\}\ c_2\ \{r\}]}{\{p\}\ c_1\,;c_2\ \{r\},}$$

and

$$\text{right-compl}(\Phi(\mathcal{P}_1[\{p\}\ c_1\ \{q\}])) = \mathcal{Q}_1[\mathcal{A}_1\{q\} \gg \{p\}\ c_1\ \{q\}]$$
$$\Phi(\mathcal{P}_2[\{q\}\ c_2\ \{r\}]) = \mathcal{Q}_2[\mathcal{A}_2 \gg \{q\}\ c_2\ \{r\}],$$

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

$$\frac{\mathcal{Q}_1[\mathcal{A}_1\ \{q\} \gg \{p\}\ c_1\ \{q\}] \qquad \mathcal{Q}_2[\mathcal{A}_2 \gg \{q\}\ c_2\ \{r\}]}{\mathcal{A}_1\ ;\mathcal{A}_2 \gg \{p\}\ c_1\ ;c_2\ \{r\}.}$$

(WH) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{i \wedge b\}\ c\ \{i\}]}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg\, b\}},$$

and

$$\text{compl}(\Phi(\mathcal{P}'[\{i \wedge b\}\ c\ \{i\}])) = \mathcal{Q}'[\{i \wedge b\}\ \mathcal{A}\ \{i\} \gg \{i \wedge b\}\ c\ \{i\}],$$

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

$$\frac{\mathcal{Q}'[\{i \wedge b\}\ \mathcal{A}\ \{i\} \gg \{i \wedge b\}\ c\ \{i\}]}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A}) \gg \{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg\, b\}}.$$

(SK) If $\mathcal{P}$ is

$$\frac{}{\{q\}\ \textbf{skip}\ \{q\}},$$

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

$$\frac{}{\textbf{skip}\ \{q\} \gg \{q\}\ \textbf{skip}\ \{q\}}.$$

(CD) If $\mathcal{P}$ is

$$\frac{\mathcal{P}_1[\{p \wedge b\}\ c_1\ \{q\}] \qquad \mathcal{P}_2[\{p \wedge \neg\ b\}\ c_2\ \{q\}]}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}},$$

and

$$\text{compl}(\Phi(\mathcal{P}_1[\{p \wedge b\}\ c_1\ \{q\}])) = \mathcal{Q}_1[\{p \wedge b\}\ \mathcal{A}_1\ \{q\} \gg \{p \wedge b\}\ c_1\ \{q\}]$$
$$\text{compl}(\Phi(\mathcal{P}_2[\{p \wedge \neg\, b\}\ c_2\ \{q\}])) =$$
$$\mathcal{Q}_2[\{p \wedge \neg\, b\}\ \mathcal{A}_2\ \{q\} \gg \{p \wedge \neg\, b\}\ c_2\ \{q\}],$$

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

$$\frac{\begin{array}{c}\mathcal{Q}_1[\{p \wedge b\}\ \mathcal{A}_1\ \{q\} \gg \{p \wedge b\}\ c_1\ \{q\}] \\ \mathcal{Q}_2[\{p \wedge \neg b\}\ \mathcal{A}_2\ \{q\} \gg \{p \wedge \neg b\}\ c_2\ \{q\}]\end{array}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ \mathcal{A}_1\ \textbf{else}\ (\mathcal{A}_2)\ \{q\} \gg \{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}.}$$

(DC) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{p\}\ c\ \{q\}]}{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\},}$$

when v does not occur free in p or q, and

$$\text{compl}(\Phi(\mathcal{P}'[\{p\}\ c\ \{q\}])) = \mathcal{Q}'[\{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}'[\{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\}]}{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ (\mathcal{A})\ \{q\} \gg \{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\}.}$$

(VAC) If $\mathcal{P}$ is

$$\frac{}{\{\textbf{false}\}\ c\ \{q\},}$$

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

$$\frac{}{\Big\{c\Big\}\text{VAC}\ \{q\} \gg \{\textbf{false}\}\ c\ \{q\}.}$$

(DISJ) If $\mathcal{P}$ is

$$\frac{\mathcal{P}_1[\{p_1\}\ c\ \{q\}] \qquad \mathcal{P}_2[\{p_2\}\ c\ \{q\}]}{\{p_1 \vee p_2\}\ c\ \{q\},}$$

and

$$\text{right-compl}(\Phi(\mathcal{P}_1[\{p_1\}\ c\ \{q\}])) = \mathcal{Q}_1[\mathcal{A}_1\ \{q\} \gg \{p_1\}\ c\ \{q\}]$$

$$\text{right-compl}(\Phi(\mathcal{P}_2[\{p_2\}\ c\ \{q\}])) = \mathcal{Q}_2[\mathcal{A}_2\ \{q\} \gg \{p_2\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}_1[\mathcal{A}_1\ \{q\} \gg \{p_1\}\ c\ \{q\}] \qquad \mathcal{Q}_2[\mathcal{A}_2\ \{q\} \gg \{p_2\}\ c\ \{q\}]}{(\mathcal{A}_1\ DISJ\ \mathcal{A}_2)\ \{q\} \gg \{p_1 \vee p_2\}\ c\ \{q\}.}$$

(CONJ) If $\mathcal{P}$ is

$$\frac{\mathcal{P}_1[\{p_1\}\ c\ \{q_1\}] \qquad \mathcal{P}_2[\{p_2\}\ c\ \{q_2\}]}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\},}$$

and

$$\Phi(\mathcal{P}_1[\{p_1\}\ c\ \{q_1\}]) = \mathcal{Q}_1[\mathcal{A}_1 \gg \{p_1\}\ c\ \{q_1\}]$$

$$\Phi(\mathcal{P}_2[\{p_2\}\ c\ \{q_2\}]) = \mathcal{Q}_2[\mathcal{A}_2 \gg \{p_2\}\ c\ \{q_2\}],$$

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

$$\frac{\mathcal{Q}_1[\mathcal{A}_1 \gg \{p_1\}\ c\ \{q_1\}] \qquad \mathcal{Q}_2[\mathcal{A}_2 \gg \{p_2\}\ c\ \{q_2\}]}{(\mathcal{A}_1\ CONJ\ \mathcal{A}_2) \gg \{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}.}$$

(EQ) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{p\}\ c\ \{q\}]}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

where v is not free in c, and

$$\Phi(\mathcal{P}'[\{p\}\ c\ \{q\}]) = \mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}]}{\left\{\mathcal{A}\right\} \exists v\ \gg \{\exists v.\ p\}\ c\ \{\exists v.\ q\}.}$$

(UQ) is similar to (EQ).

(FR) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{p\}\ c\ \{q\}]}{\{p\ *\ r\}\ c\ \{q\ *\ r\},}$$

where no variable occurring free in r is modified by c, and

$$\Phi(\mathcal{P}'[\{p\}\ c\ \{q\}]) = \mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}]}{\left\{\mathcal{A}\right\}\ *\ r\ \gg \{p\ *\ r\}\ c\ \{q\ *\ r\}.}$$

(SUB) If $\mathcal{P}$ is

$$\frac{\mathcal{P}'[\{p\}\ c\ \{q\}]}{\{p/\delta\}\ (c/\delta)\ \{q/\delta\},}$$

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

$$\Phi(\mathcal{P}'[\{p\}\ c\ \{q\}]) = \mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}],$$

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

$$\frac{\mathcal{Q}'[\mathcal{A} \gg \{p\}\ c\ \{q\}]}{\big\{\mathcal{A}\big\}/\delta\ \gg \{p/\delta\}\ (c/\delta)\ \{q/\delta\}.}$$

(MUBR) If $\mathcal{P}$ is

$$\frac{}{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} q)\}\ [e] := e'\ \{q\},}$$

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

$$\frac{}{[e] := e'\ \{q\} \gg \{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} q)\}\ [e] := e'\ \{q\}.}$$

(DISBR) If $\mathcal{P}$ is

$$\frac{}{\{(e \mapsto -)\ *\ q\}\ \mathbf{dispose}\ e\ \{q\},}$$

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

$$\frac{}{\mathbf{dispose}\ e\ \{q\} \gg \{(e \mapsto -)\ *\ q\}\ \mathbf{dispose}\ e\ \{q\}.}$$

(CONSBR) If $\mathcal{P}$ is

$$\frac{}{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} q''\}\ v := \mathbf{cons}(\overline{e})\ \{q\},}$$

where v'' is distinct from v, $v'' \notin \mathrm{FV}(\overline{e}, q)$, and q'' denotes $q/v \to v''$, then $\Phi(\mathcal{P})$ is

$$\frac{}{v := \mathbf{cons}(\overline{e})\ \{q\} \gg \{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} q''\}\ v := \mathbf{cons}(\overline{e})\ \{q\}.}$$

(LKBR1) If $\mathcal{P}$ is

$$\frac{}{\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} q'')\}\ v := [e]\ \{q\},}$$

where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(q) - \{v\})$, and q'' denotes $q/v \to v''$, then $\Phi(\mathcal{P})$ is

$$\frac{}{v := [e]\ \{q\} \gg \{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} q'')\}\ v := [e]\ \{q\}.}$$

Finally, we will close the circle by defining a function Ψ that maps annotated specifications into proofs of the specifications that they annotate. But first, we must define certain sequences of premisses and proofs, and associated concepts:

- A *premiss from p to q* is either a specification $\{p\}\ c\ \{q\}$ or a verification condition $p \Rightarrow q$.
- A *coherent premiss sequence from p to q* is either a premiss from p to q, or a shorter coherent premiss sequence from p to some r, followed by a premiss from r to q. We use $\mathcal{S}$ (with occasional decorations) as a variable that ranges over coherent premiss sequences, and $\mathcal{S}[p, q]$ as a variable that ranges over coherent premiss sequences from p to q.
- A *proof from p to q* is either a proof of a specification $\{p\}\ c\ \{q\}$ or a verification condition $p \Rightarrow q$.
- A *coherent proof sequence from p to q* is either a proof from p to q, or a shorter coherent proof sequence from p to some r, followed by a premiss from r to q. We use $\mathcal{R}$ (with occasional decorations) as a variable that ranges over coherent proof sequences, and $\mathcal{R}[p, q]$ as a variable that ranges over coherent proof sequences from p to q.
- A coherent premiss (or proof) sequence is *proper* if it contains at least one specification (or proof of a specification).
- We extend the function "concl" to map coherent proof sequences into coherent premiss sequences by replacing each proof of a specification by its conclusion, and leaving verification conditions unchanged.
- The function "code" maps proper coherent premiss sequences into commands by sequentially composing the commands occurring within the premisses that are specifications. More precisely (where code$'$ is an auxilliary function mapping coherent premiss sequences or the empty sequence ϵ into commands):

$$\begin{aligned} \text{code}'(\mathcal{S}, \{p\}\ c\ \{q\}) &= \text{code}'(\mathcal{S})\,;c \\ \text{code}'(\mathcal{S}, p \Rightarrow q) &= \text{code}'(\mathcal{S}) \\ \text{code}'(\epsilon) &= \mathbf{skip} \\ \text{code}(\mathcal{S}) &= c \text{ when } \text{code}'(\mathcal{S}) = \mathbf{skip}\,;c. \end{aligned}$$

The utility of these concepts is that, for any proper coherent premiss sequence $\mathcal{S}$ from p to q, one can derive the inference rule

$$\frac{\mathcal{S}}{\{p\}\ \mathrm{code}(\mathcal{S})\ \{q\}.}$$

The derivation is obtained by using the rules (SQ), (SP), and (WC) to build up a proper coherent proof sequence $\mathcal{R}$ from p to q in which the components of $\mathcal{S}$ occur as assumptions. One begins by taking $\mathcal{R}$ to be $\mathcal{S}$, and then repeatedly applies the following nondeterminate step:

If $\mathcal{R}$ can be put in the form

$$\mathcal{R}_1, \mathcal{P}_1[\{p'\}\ c_1\ \{q'\}], \mathcal{P}_2[\{q'\}\ c_2\ \{r'\}], \mathcal{R}_2,$$

replace it by

$$\mathcal{R}_1, \frac{\mathcal{P}_1[\{p'\}\ c_1\ \{q'\}] \quad \mathcal{P}_2[\{q'\}\ c_2\ \{r'\}]}{\{p'\}\ c_1\ ;\ c_2\ \{r'\}}, \mathcal{R}_2,$$

or, if $\mathcal{R}$ can be put in the form

$$\mathcal{R}_1, p' \Rightarrow q', \mathcal{P}_2[\{q'\}\ c\ \{r'\}], \mathcal{R}_2,$$

replace it by

$$\mathcal{R}_1, \frac{p' \Rightarrow q' \quad \mathcal{P}_2[\{q'\}\ c\ \{r'\}]}{\{p'\}\ c\ \{r'\}}, \mathcal{R}_2,$$

or if $\mathcal{R}$ can be put in the form

$$\mathcal{R}_1, \mathcal{P}_1[\{p'\}\ c\ \{q'\}], q' \Rightarrow r', \mathcal{R}_2,$$

replace it by

$$\mathcal{R}_1, \frac{\mathcal{P}_1[\{p'\}\ c\ \{q'\}] \quad q' \Rightarrow r'}{\{p'\}\ c\ \{r'\}}, \mathcal{R}_2.$$

Each step reduces the length of $\mathcal{R}$ while preserving code(concl($\mathcal{R}$)), so that eventually one reaches the state where $\mathcal{R}$ is a single proof, of $\{p\}$ code($\mathcal{S}$) $\{q\}$.

We will define the function Ψ in terms of a function Ψ_0 that maps annotations into proper coherent proof sequences. Specifically, if $\Psi_0(\mathcal{A})$ is a proper coherent proof sequence from p to q, then

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

Strictly speaking, we should replace the final step of this proof by one of its derivations. The annotation $\mathcal{A}$ contains insufficient information to determine which of these derivations should be used; fortunately, the choice doesn't matter.

The function Ψ_0 is defined as follows:

$$\Psi_0(\{q\}) = \epsilon$$

$$\Psi_0(\mathcal{A}_0\ \{p\}\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{p\}), p \Rightarrow q$$

$$\Psi_0(\mathcal{A}_0\ v := e\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{q/v \rightarrow e\}),\ \overline{\{q/v \rightarrow e\}\ v := e\ \{q\}}$$

$$\Psi_0(\mathcal{A}_0\ ;\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{q\})$$

$$\Psi_0(\mathcal{A}_0\ \{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A})\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A}))\ \{i \wedge \neg\, b \Rightarrow q\}$$

$$\Psi_0(\mathcal{A}_0\ \{i\}\ \textbf{while}\ b\ \textbf{do}\ (\mathcal{A})) = \Psi_0(\mathcal{A}_0\ \{i\}),\ \frac{\Psi(\{i \wedge b\}\ \mathcal{A}\ \{i\})}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ \mathrm{cd}(\mathcal{A})\ \{i \wedge \neg\, b\}}$$

$$\Psi_0(\mathcal{A}_0\ \textbf{skip}\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{q\}),\ \overline{\{q\}\ \textbf{skip}\ \{q\}}$$

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

$$\Psi_0(\mathcal{A}_0\ \{p\}\ \textbf{newvar}\ v\ \textbf{in}\ (\mathcal{A})\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{p\}),\ \frac{\Psi(\{p\}\ \mathcal{A}\ \{q\})}{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ \mathrm{cd}(\mathcal{A})\ \{q\}}$$

$$\Psi_0(\mathcal{A}_0\ \big\{c\big\}\mathrm{VAC}\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{\textbf{false}\}),\ \overline{\{\textbf{false}\}\ c\ \{q\}}$$

$$\Psi_0(\mathcal{A}_0\ (\mathcal{A}_1\ \mathrm{DISJ}\ \mathcal{A}_2)\ \{q\}) = \Psi_0(\mathcal{A}_0\ \{p_1 \vee p_2\}),\ \frac{\Psi(\mathcal{A}_1\ \{q\}) \quad \Psi(\mathcal{A}_2\ \{q\})}{\{p_1 \vee p_2\}\ c\ \{q\}}$$

where $\Psi(\mathcal{A}_1\ \{q\})$ proves $\{p_1\}\ c\ \{q\}$
and $\Psi(\mathcal{A}_2\ \{q\})$ proves $\{p_2\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\ (\mathcal{A}_1\ \mathrm{CONJ}\ \mathcal{A}_2)\ \{q\}) = \Psi_0(\mathcal{A}_0\ (\mathcal{A}_1\ \mathrm{CONJ}\ \mathcal{A}_2), q_1 \wedge q_2 \Rightarrow q$$

where $\Psi(\mathcal{A}_1)$ proves $\{p_1\}\ c\ \{q_1\}$
and $\Psi(\mathcal{A}_2)$ proves $\{p_2\}\ c\ \{q_2\}$

$$\Psi_0(\mathcal{A}_0\,(\mathcal{A}_1\,\mathrm{CONJ}\,\mathcal{A}_2)) = \Psi_0(\mathcal{A}_0\,\{p_1 \wedge p_2\}),\ \frac{\Psi(\mathcal{A}_1) \quad \Psi(\mathcal{A}_2)}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}}$$

where $\Psi(\mathcal{A}_1)$ proves $\{p_1\}\ c\ \{q_1\}$
and $\Psi(\mathcal{A}_2)$ proves $\{p_2\}\ c\ \{q_2\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\,\exists v\,\{r\}) = \Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\,\exists v), q \Rightarrow r$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

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

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\,\forall v\,\{r\}) = \Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\,\forall v), q \Rightarrow r$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\,\forall v) = \Psi_0(\mathcal{A}_0\,\{\forall v.\ p\}),\ \frac{\Psi(\mathcal{A})}{\{\forall v.\ p\}\ c\ \{\forall v.\ q\}}$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\ *\ r\,\{s\}) = \Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\ *\ r), q\ *\ r \Rightarrow s$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}\ *\ r) = \Psi_0(\mathcal{A}_0\,\{p\ *\ r\}),\ \frac{\Psi(\mathcal{A})}{\{p\ *\ r\}\ c\ \{q\ *\ r\}}$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}/\delta\,\{r\}) = \Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}/\delta), q/\delta \Rightarrow r$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,\{\mathcal{A}\}/\delta) = \Psi_0(\mathcal{A}_0\,\{p/\delta\}),\ \frac{\Psi(\mathcal{A})}{\{p/\delta\}\ c\ \{q/\delta\}}$$

where $\Psi(\mathcal{A})$ proves $\{p\}\ c\ \{q\}$

$$\Psi_0(\mathcal{A}_0\,[e] := e'\,\{q\}) = \Psi_0(\mathcal{A}_0\,\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} q)\}),$$

$$\overline{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} q)\}\,[e] := e'\,\{q\}}$$

$$\Psi_0(\mathcal{A}_0\,\mathbf{dispose}\,e\,\{q\}) = \Psi_0(\mathcal{A}_0\,\{(e \mapsto -)\ *\ q\}),$$

$$\overline{\{(e \mapsto -)\ *\ q\}\,\mathbf{dispose}\,e\,\{q\}}$$

$$\Psi_0(\mathcal{A}_0\,v := \mathbf{cons}(\overline{e})\,\{q\}) = \Psi_0(\mathcal{A}_0\,\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} q''\}),$$

$$\overline{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} q''\}\ v := \mathbf{cons}(\overline{e})\ \{q\}}$$

$$\Psi_0(\mathcal{A}_0\,v := [e]\,\{q\}) = \Psi_0(\mathcal{A}_0\,\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} q'')\}),$$

$$\overline{\{\exists v''.\ (e \mapsto v'')\ *\ ((e \mapsto v'') \mathrel{-\!\!*} q'')\}\ v := [e]\ \{q\}},$$

where, in the last two equations, q'' denotes $q/v \to v''$. Then:

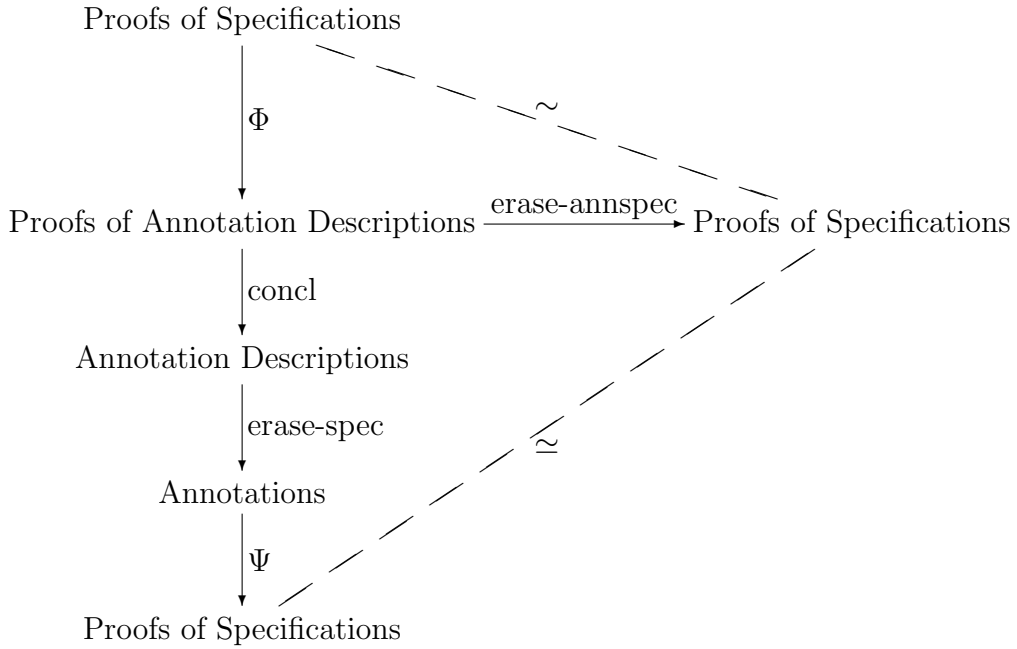
Proposition 14 *If $\mathcal{A} \gg \{p\}\ c\ \{q\}$ is provable by $\mathcal{Q}$, then there is a a proper coherent proof sequence $\mathcal{R}$ from p to q such that:*

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

PROOF The proof is by induction on the structure of the proof $\mathcal{Q}$. Note that parts 2 and 3 are immediate consequences of Part 1b (taking $\mathcal{A}_0$ to be empty) and the definition of Ψ in terms of Ψ_0. END OF PROOF

The developments in this section are summarized by the following diagram, in which $\simeq$ relates proofs containing the same verification conditions,

while $\sim$ relates proofs containing the same verification conditions, except perhaps for additional trivial implications of the form $p \Rightarrow p$ in the set of proofs on the right:



Exercise 3

Fill in the postconditions in

$$\{(e_1 \mapsto -) \ * \ (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}$$

$$\{(e_1 \mapsto -) \ \wedge \ (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}.$$

to give two sound inference rules describing a sequence of two mutations. Your postconditions should be as strong as possible.

Give a derivation of each of these inference rules, exhibited as an annotated specification.

Exercise 4

The alternative inference rule for conditional commands (CDalt), leads to the following rule for annotated specifications:

- Alternative Rule for Conditionals (CDaltan)

$$\frac{\mathcal{A}_1\,\{q\} \gg \{p_1\}\ c_1\ \{q\} \qquad \mathcal{A}_2\,\{q\} \gg \{p_2\}\ c_2\ \{q\}}{\begin{array}{l}(\textbf{if } b \textbf{ then } \mathcal{A}_1 \textbf{ else } \mathcal{A}_2)\,\{q\} \gg \\ \quad \{(b \Rightarrow p_1) \wedge (\neg\, b \Rightarrow p_2)\}\ (\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2)\ \{q\},\end{array}}$$

Examine the annotated specifications in this and the following chapters, and determine how they would need to be changed if (CDan) were replaced by (CDaltan).

Exercise 5

The following are alternative global rules for allocation and lookup that use unmodified variables (v' and v''):

- The unmodified-variable global form for allocation (CONSGG)

$$\frac{}{\{v = v' \wedge r\}\ v := \textbf{cons}(\overline{e})\ \{(v \mapsto \overline{e}')\ *\ r'\},}$$

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

- The unmodified-variable global form for lookup (LKGG)

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

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

Derive (CONSGG) from (CONSG), and (CONSL) from (CONSGG). Derive (LKGG) from (LKG), and (LKL) from (LKGG).

Exercise 6

Derive (LKNOL) from (LKNOG) and vice-versa. (Hint: To derive (LKNOL) from (LKNOG), use the version of (LKNOG) where $v'' = v$. To derive (LKNOG) from (LKNOL), assume v and v'' are distinct, and then apply renaming of v'' in the precondition to cover the case where $v = v''$.)

Exercise 7

Closely akin to (3.1) is the following equivalence of meaning between two lookup commands:

$$v := [e] \cong \textbf{newvar } \hat{v} \textbf{ in } (\hat{v} := [e] \,;\, v := \hat{v}). \tag{3.2}$$

Use this equivalence to derive (LKG) from (LKNOG).

An Introduction
to Separation Logic

October 23, 2008

Chapter 4

Lists and List Segments

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

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

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

These operations obey a variety of laws, including:

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

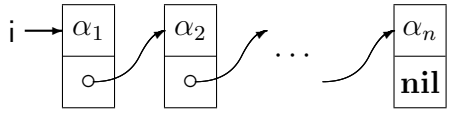
$$\epsilon^\dagger = \epsilon \qquad [a]^\dagger = [a] \qquad (\alpha{\cdot}\beta)^\dagger = \beta^\dagger{\cdot}\alpha^\dagger$$

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

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

4.1 Singly-Linked List Segments

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



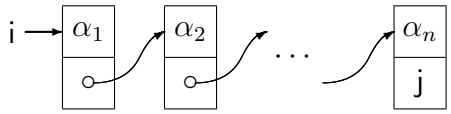
by structural induction on α:

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In general, when

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

we have

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

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

We can define *nontouching list segments* inductively by:

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

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

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

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

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

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

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

$$\mathsf{list}\ \alpha\ \mathsf{i} \Rightarrow \mathsf{ntlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil})$$

$$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{list}\ \beta\ \mathsf{j} \Rightarrow \mathsf{ntlseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{list}\ \beta\ \mathsf{j}$$

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

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

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

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

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\}$

$\mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{i})$; (CONSNOG)

$\{\mathsf{k}\mapsto\mathsf{a},\mathsf{i}\ *\ \mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\}$

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

$\{\mathsf{lseg}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{k},\mathsf{j})\}$

$\mathsf{i} := \mathsf{k}$ (AS)

$\{\mathsf{lseg}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j})\}$,

or, more concisely:

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{k})\}$

$\mathsf{i} := \mathbf{cons}(\mathsf{a},\mathsf{i})$; (CONSG)

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

$\{\mathsf{lseg}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}$.

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

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a},\mathsf{k}\}$

$\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})$; (CONSNOG)

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a},\mathsf{k}\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a}\ *\ \mathsf{j}+1\mapsto\mathsf{k}\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a}\ *\ \mathsf{j}+1\mapsto -\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$[\mathsf{j}+1] := \mathsf{l}$ (MUG)

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a}\ *\ \mathsf{j}+1\mapsto\mathsf{l}\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto\mathsf{a},\mathsf{l}\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\,\alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\ *\ \mathsf{l}\mapsto\mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\,\alpha{\cdot}\mathsf{a}{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{k})\}$.

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

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

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

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:

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

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:

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

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

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

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

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

$[\mathsf{j}]:=\mathsf{x}\,;$ (MUG)

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

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

$\mathsf{j}:=[\mathsf{j}+1]\,;$ (LKG)

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

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

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

$\mathsf{m}:=\mathsf{m}+1$ (AS)

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

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

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

4.3 Preciseness Revisited

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

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

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

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

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

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

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

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

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

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

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

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

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

so that there are j and h' such that $i \neq \mathbf{nil}$ (since **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 $h_0 = h_1$.

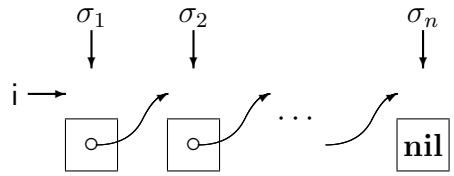
(2) As described in Section 2.3.3, p is precise whenever $p \Rightarrow q$ is valid and q is precise. Thus, since $\mathbf{list}\ \alpha\ \mathsf{i} \Rightarrow \exists\alpha.\ \mathbf{list}\ \alpha\ \mathsf{i}$ is valid, $\mathbf{list}\ \alpha\ \mathsf{i}$ is precise.

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Simple Procedures

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

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

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

An additional peculiarity, which substantially simplifies reasoning about simple procedures, is that we syntactically distinguish parameters that may be modified from those that may not be.

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

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

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

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

where

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

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

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

where

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

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

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

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

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

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

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

We define a hypothetical specification to have the form

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

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

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

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

- Strengthening Precedent (SP)

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

- Substitution (SUB)

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

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

Note that substitutions do not affect procedure names.

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

- Hypothesis (HYPO)

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

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

- Simple Procedures (SPROC)

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

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

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

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

- Simple Recursive Procedures (SRPROC)

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

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

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

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

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

- Call (CALL)

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

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

- General Call (GCALL)

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

  where δ is a substitution

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

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

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

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

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

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

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

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

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

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

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

Then (GCALL) becomes

- General Call (GCALLan)

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

  where δ is the substitution

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

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

It is straightforward to annotate procedure definitions — it is enough to provide a list of the formal ghost parameters, along with annotated specifications of the procedure body and of the command in which the procedure is used. However, the annotation of the procedure body must be complete (at least in the recursive case — for simplicity, we impose this requirement on nonrecursive definitions as well):

- Simple Procedures (SPROCan)

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

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

- Simple Recursive Procedures (SRPROCan)

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

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

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

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

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

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

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

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

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

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

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

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

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

As for the specification of the main-level call

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

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

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

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

4.6 Still More about Annotated Specifications

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

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

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

and extending the definition of cd to our new annotations:

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

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

and similarly for **letrec**.

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

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

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

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

by

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

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

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

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

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

$$\frac{}{\begin{array}{l}\Gamma, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\}, \Gamma' \vdash \\ \qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e_1', \ldots, e_n')\ \{q/\delta\},\end{array}}$$

where δ is the substitution

$$\delta = v_1 \to w_1, \ldots, v_m \to w_m, v_1' \to e_1', \ldots, v_n' \to e_n', v_1'' \to e_1'', \ldots, v_k'' \to e_k'',$$

which acts on all the free variables in $\{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\}$, and $w_1, \ldots, w_m$ are distinct variables that do not occur free in the expressions $e_1' \ldots, e_n'$ or $e_1'' \ldots, e_k''$. Without loss of generality, we can assume that the domain of δ is exactly the set of variables occurring free in $\{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\ \{q\}$. Then $\Phi(\mathcal{P})$ is

$$\frac{}{\begin{array}{l}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v_1', \ldots, v_n')\{v_1'', \ldots, v_k''\}\ \{q\}, \hat{\Gamma}' \vdash \\ \qquad h(w_1, \ldots, w_m; e_1', \ldots, e_n')\{e_1'', \ldots, e_k''\} \gg \\ \qquad\quad \{p/\delta\}\ h(w_1, \ldots, w_m; e_1', \ldots, e_n')\ \{q/\delta\},\end{array}}$$

where $v_1'', \ldots, v_k''$ are listed in the standard ordering.

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

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

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

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

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

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

$$\frac{\mathcal{Q}_1(\hat{\Gamma} \vdash \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\})\quad \mathcal{Q}_2(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}' \gg \{p'\}\ c'\ \{q'\})}{\hat{\Gamma} \vdash \textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \textbf{in}\ \mathcal{A}' \gg \{p'\}\ \textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c'\ \{q'\}.}$$

(SRPROC) is similar to (SPROC).

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

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

Finally, we must extend the function Ψ appropriately. First, we redefine a *premiss from p to q* to be either a hypothetical specification $\Gamma \vdash \{p\}\ c\ \{q\}$ or a verification condition $p \Rightarrow q$, and we require all the hypothetical specifications in a coherent premiss sequence (or the conclusions of a coherent proof sequence) to contain the same context.

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

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

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

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

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

The function Ψ becomes a map from hypothetical annotated specifications to proofs of hypothetical specifications. In the cases given in Section 3.12, one prefixes $\hat{\Gamma}$ to every annotated specification and Γ to every

specification. (We no longer require $\hat{\Gamma}$ to be in standard form, but Γ is still obtained from $\hat{\Gamma}$ by erasing lists of ghost parameters.) For example:

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

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

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

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

To extend Ψ_0 to procedure calls, assume that $\hat{\Gamma}$ contains the hypothesis $\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\}$. Then

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

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

where δ is the substitution

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

Then for simple procedure definitions, we have:

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

$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \{p\}\ \mathcal{A}\ \{q\}\ \mathbf{in}\ \mathcal{A}') =$$
$$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\, \{p'\}),\ \frac{\Psi(\hat{\Gamma} \vdash \{p\}\ \mathcal{A}\ \{q\}) \quad \Psi(\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \mathcal{A}')}{\hat{\Gamma} \vdash \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = \mathrm{cd}(\mathcal{A})\ \mathbf{in}\ c'\ \{q'\}},$$

where

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

Then Proposition 14 generalizes to:

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

1. *If $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\{p\})$ is defined, then:*

   (a) $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\mathcal{A})$ *and* $\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\mathcal{A}\,\{r\})$ *are defined,*

   (b) $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\mathcal{A}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\{p\}), \mathcal{R}\,,$$

   (c) $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\mathcal{A}\,\{r\}) = \Psi_0(\hat{\Gamma} \vdash \mathcal{A}_0\,\{p\}), \mathcal{R}, q \Rightarrow r\,,$$

2. $$\Psi_0(\hat{\Gamma} \vdash \mathcal{A}) = \mathcal{R}\,,$$

3. $$\Psi(\hat{\Gamma} \vdash \mathcal{A})\ \textit{is a proof of}\ \Gamma \vdash \{p\}\ c\ \{q\}\,,$$

4. *The verification conditions in $\hat{\Gamma} \vdash \Psi(\mathcal{A})$ are the verification conditions in erase-annspec($\mathcal{Q}$).*

4.7 An Extended Example: Sorting by Merging

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

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

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

4.7.1 Some Concepts about Sequences

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

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

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

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

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

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

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

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

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

Pointwise extension satisfies the laws:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then

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

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

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

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

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

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

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

4.7.2 The Sorting Procedure mergesort

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

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

Suppose we define the abbreviation

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

Then we will define a sorting procedure mergesort satisfying the hypothesis

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

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

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

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

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

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

4.7.3 A Proof for mergesort

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

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

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

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

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

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

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

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

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

Then β is renamed β_1 in the postcondition:

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

Next, the frame rule is used to infer

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

Then the rule (EQ) for existential quantification gives

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

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

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

4.7.4 A Proof for merge

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

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

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

(Note that merge1 does not modify any variables.)

The body of merge1 is verified by the annotated specification:

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

else newvar j **in newvar** $\mathsf{a1}$ **in**

$\Big(\mathsf{n1} := \mathsf{n1} - 1\ ;\ \mathsf{j} := \mathsf{i1}\ ;\ \mathsf{i1} := [\mathsf{j}+1]\ ;\ \mathsf{a1} := [\mathsf{i1}]\ ;$

$\{\exists \beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2.$

$(\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}', \mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1'\ (\mathsf{j1}, -)$

$*\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -))$

$\wedge\ \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$

$\wedge\ \mathbf{ord}\ (\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\ \beta$

$\wedge\ \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1}' \leq \mathsf{a2}\}$

$\{\exists \beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2.$

$(\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}', \mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1'\ (\mathsf{j1}, -)$

$*\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -))$

$\wedge\ \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge (\beta{\cdot}\mathsf{a1}'){\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$

$\wedge\ \mathbf{ord}\ (\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\ (\beta{\cdot}\mathsf{a1}') \wedge \{\beta{\cdot}\mathsf{a1}'\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}$

if $\mathsf{a1} \leq \mathsf{a2}$ **then**

$\{\exists \beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2.$

$(\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{i1})\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -)\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -))$

$\wedge\ \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2$

$\wedge\ \mathbf{ord}\ (\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\ \beta$

$\wedge\ \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\} \wedge \mathsf{a1} \leq \mathsf{a2}\}$

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

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

else $([\mathsf{j}+1] := \mathsf{i2}\ ;$

$\{\exists \beta, \mathsf{a2}, \mathsf{j2}, \gamma_2, \mathsf{j1}, \gamma_1.$

$(\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{i2})\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -)\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -))$

$\wedge\ \#\gamma_2 = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \beta{\cdot}\mathsf{a2}{\cdot}\gamma_2{\cdot}\mathsf{a1}{\cdot}\gamma_1 \sim \beta_2{\cdot}\beta_1$

$\wedge\ \mathbf{ord}\ (\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\ (\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\ \beta$

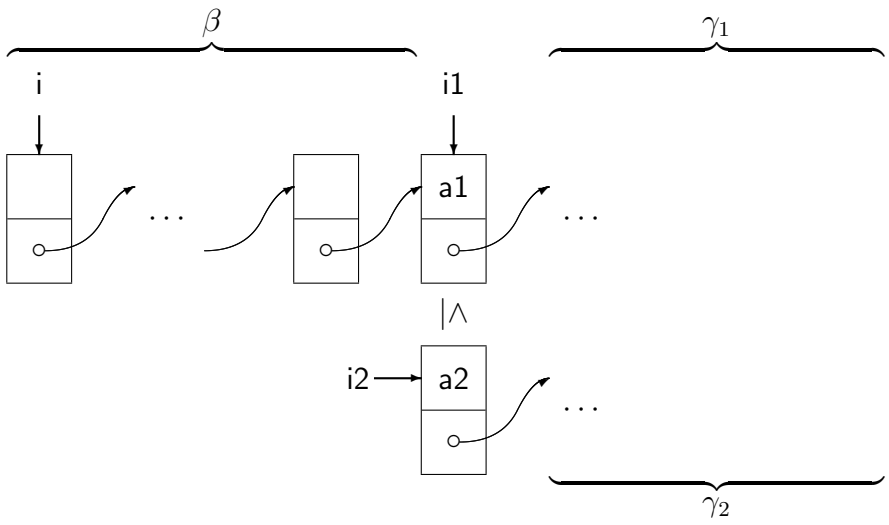
$\wedge\ \{\beta\} \leq^* \{\mathsf{a2}{\cdot}\gamma_2\} \cup \{\mathsf{a1}{\cdot}\gamma_1\} \wedge \mathsf{a2} \leq \mathsf{a1}\}$

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

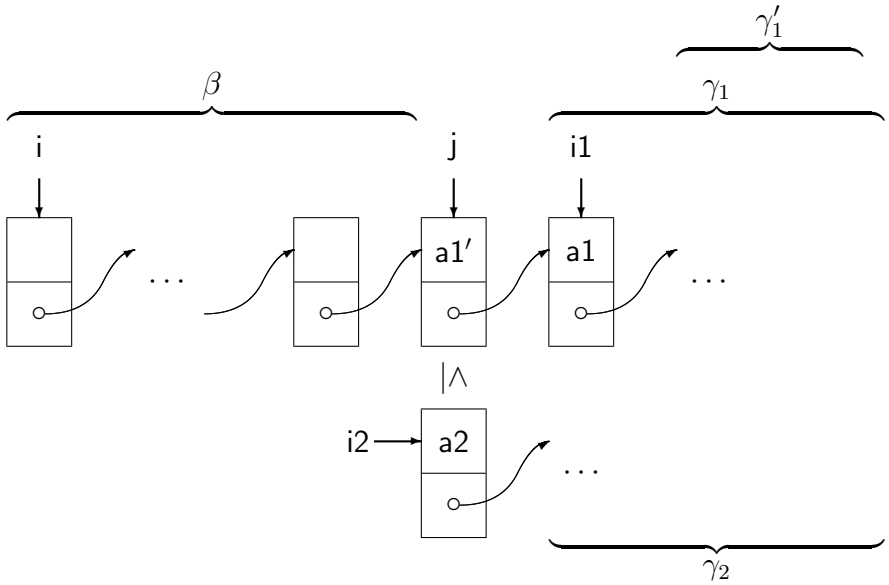
$\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta\}$

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

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



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



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

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

since

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

(Here the asterisks indicate conclusions.)

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

4.7.5 merge with goto commands

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

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

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

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

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

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

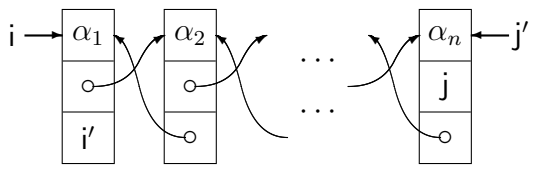
The following is an annotated specification of the procedure. The assertions for the labels $\ell 1$ and $\ell 2$ correspond to the preconditions of the two recursive calls in the procedure in the previous section. The assertion for the label out is the postcondition of the procedure body.

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

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

4.8 Doubly-Linked List Segments

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

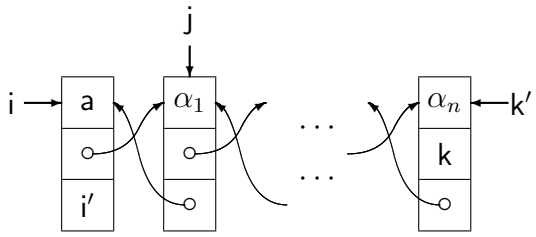


The author knows of no way to verbalize this predicate succinctly — "$(\mathsf{i}, \mathsf{i}')$ to $(\mathsf{j}, \mathsf{j}')$ is a doubly-linked list representing α" is both unwieldly and unlikely to summon the right picture to mind. But, as usual, the concept can be defined precisely by structural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

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

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

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



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

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

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

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

One can also define a doubly-linked list by

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

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

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

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

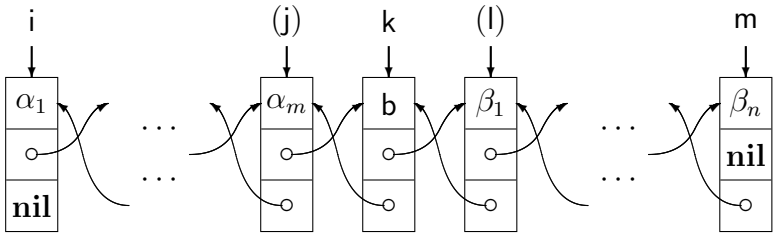
$$\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{j}' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}))$$

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

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

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

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

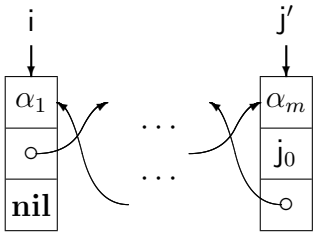


$\{\exists \mathsf{j}, \mathsf{l}.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
$\mathsf{l} := [\mathsf{k} + 1]\ ;\ \mathsf{j} := [\mathsf{k} + 2]\ ;$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
$\mathbf{dispose}\ \mathsf{k}\ ;\ \mathbf{dispose}\ \mathsf{k} + 1\ ;\ \mathbf{dispose}\ \mathsf{k} + 2\ ;$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
$\mathbf{if}\ \mathsf{j} = \mathbf{nil}\ \mathbf{then}$
  $\{\mathsf{i} = \mathsf{k} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
  $\mathsf{i} := \mathsf{l}$
  $\{\mathsf{i} = \mathsf{l} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
$\mathbf{else}$
  $\{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}, \mathsf{n}$
    $*\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$
  $[\mathsf{j} + 1] := \mathsf{l}\ ;$
  $\{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}, \mathsf{n}$
    $*\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$
$\mathbf{if}\ \mathsf{l} = \mathbf{nil}\ \mathbf{then}$
  $\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{k} = \mathsf{m} \wedge \beta = \epsilon\}$
  $\mathsf{m} := \mathsf{j}$
  $\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{j} = \mathsf{m} \wedge \beta = \epsilon\}$
$\mathbf{else}$
  $\{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{k}$
    $*\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a} \cdot \beta'\}$
  $[\mathsf{l} + 2] := \mathsf{j}$
  $\{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{j}$
    $*\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a} \cdot \beta'\}$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{j}, \mathbf{nil}, \mathsf{m})\}$
$\{\mathsf{dlseg}\ \alpha \cdot \beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}$

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

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

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



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

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

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

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

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

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

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

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

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

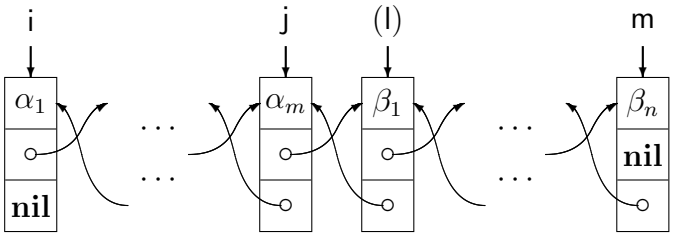
One can define a symmetric pair of procedures that act upon right-end segments. By similar proofs, one can establish

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

and

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

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



$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}_0, \mathsf{j}')\}\ \mathsf{lookuprpt}(\mathsf{j}; \mathsf{i}, \mathsf{j}')\{\alpha, \mathsf{j}_0\}\ \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}_0, \mathsf{j}') \wedge \mathsf{j} = \mathsf{j}_0\},$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}_0, \mathsf{j}')\}\ \mathsf{setrpt}(\mathsf{i}; \mathsf{j}, \mathsf{j}')\{\alpha, \mathsf{j}_0\}\ \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{j}')\},$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}'_0, \mathbf{nil}, \mathsf{j}')\ \mathsf{lookuplpt}(\mathsf{i}'; \mathsf{i}, \mathsf{j}')\{\alpha, \mathsf{i}'_0\}\ \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}'_0, \mathbf{nil}, \mathsf{j}') \wedge \mathsf{i}' = \mathsf{i}'_0\},$
$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}'_0, \mathbf{nil}, \mathsf{j}')\ \mathsf{setlpt}(\mathsf{j}'; \mathsf{i}, \mathsf{i}')\{\alpha, \mathsf{i}'_0\}\ \{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathbf{nil}, \mathsf{j}')\} \vdash$

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

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

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

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

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

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

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

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

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

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

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

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

Beginning with the hypothesis

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

we use (GCALLan) to infer

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

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

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

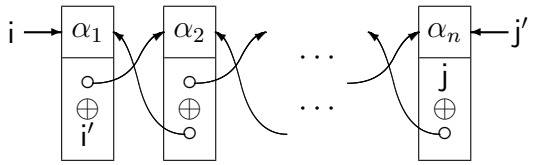
Finally, after checking that j_0 is not modified by $\mathsf{lookuprpt}(\mathsf{l}; \mathsf{i}, \mathsf{j})$, we use the rule for existential quantification, as well as predicate-calculus rules for renaming and eliminating existential quantifiers, to obtain

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

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

4.9 Xor-Linked List Segments

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



which is defined inductively by

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

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

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

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

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

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

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

while the emptyness conditions are the same:

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

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

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

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

as is the definition of lists:

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

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

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

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

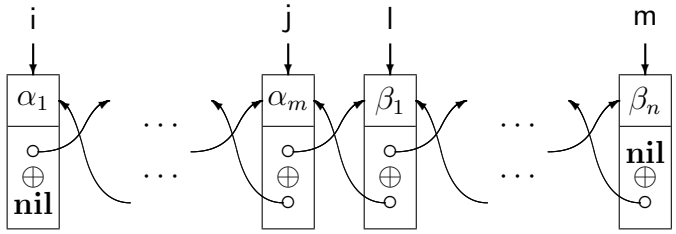
(Notice that what was the ghost variable j_0 in the specification of setrpt has become a variable j that is evaluated (but not modified) by xsetrpt.

Similarly, one can define a procedure

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

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

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



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

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

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

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

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

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

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

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

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

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

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

and thus

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

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

Exercise 8

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

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

...

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

Exercise 9

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

Exercise 10

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 11

A *braced list segment* is a list segment with an interior pointer j to its last element; in the special case where the list segment is empty, j is **nil**. Formally,

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

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

Prove the assertion

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

Exercise 12

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

1. A procedure for looking up the final pointer:

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

   (This procedure should not alter the heap.)

2. A procedure for setting the final pointer:

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

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

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

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

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

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

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

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

6. A procedure for concatenating two segments:

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

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

Exercise 13

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

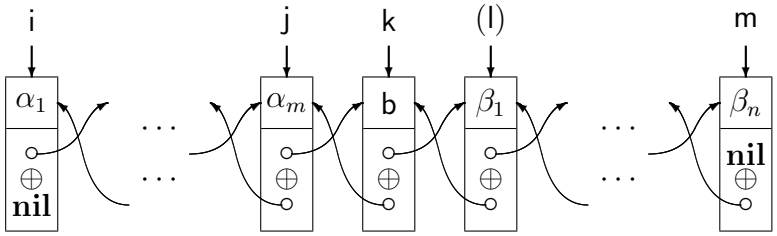
$$\{\exists \mathsf{j}, \mathsf{l}.\ \ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$$

$$\ldots$$

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

Exercise 14

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:

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

$$\ldots$$

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

Exercise 15

Prove, by structural induction on α, that

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

An Introduction
to Separation Logic

October 23, 2008

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

$$\tau \in \text{S-exps iff } \tau \in \text{Atoms}$$
$$\text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps}.$$

Here atoms are values that are not addresses, while $(\tau_1 \cdot \tau_2)$ 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:

$$\mathsf{tree}\,a\,(i) \text{ iff } \mathbf{emp} \wedge i = a \qquad \text{when } a \text{ is an atom}$$
$$\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff } \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ * \ \mathsf{tree}\,\tau_1\,(i_1) \ * \ \mathsf{tree}\,\tau_2\,(i_2).$$

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 $\mathsf{copytree}(\mathsf{j};\mathsf{i})$ that nondestructively copies the tree i to j, i.e., that satisfies $\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau\}\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}$. (Here τ is a ghost parameter.) Essentially, our proof is an annotated specification that is an

instance of the first premiss of the rule (SRPROC):

$$
\begin{array}{l}
\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau\}\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\} \vdash \\
\quad \{\mathsf{tree}\,\tau(\mathsf{i})\} \\
\quad\quad \textbf{if}\ \mathsf{isatom}(\mathsf{i})\ \textbf{then} \\
\quad\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\} \\
\quad\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{i} = \tau))\} \\
\quad\quad \mathsf{j} := \mathsf{i} \\
\quad\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{j} = \tau))\} \\
\quad \textbf{else} \\
\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{tree}\,(\tau_1 \cdot \tau_2)(\mathsf{i})\} \\
\quad\quad \textbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \textbf{in} \\
\quad\quad\quad \Big(\mathsf{i}_1 := [\mathsf{i}]\,;\, \mathsf{i}_2 := [\mathsf{i}+1]\,; \\
\quad\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2))\} \\
\quad\quad\quad \left.\begin{array}{l} \{\mathsf{tree}\,\tau_1(\mathsf{i}_1)\} \\ \mathsf{copytree}(\mathsf{j}_1;\mathsf{i}_1)\{\tau_1\} \\ \{\mathsf{tree}\,\tau_1(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\} \end{array}\right\} * \left\{\begin{array}{l} \tau = (\tau_1 \cdot \tau_2) \wedge \\ (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\ \quad \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)) \end{array}\right\} \exists \tau_1, \tau_2 \\
\quad\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad\quad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\} \\
\quad\quad\quad \left.\begin{array}{l} \{\mathsf{tree}\,\tau_2(\mathsf{i}_2)\} \\ \mathsf{copytree}(\mathsf{j}_2;\mathsf{i}_2)\{\tau_2\} \\ \{\mathsf{tree}\,\tau(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_2(\mathsf{j}_2)\} \end{array}\right\} * \left\{\begin{array}{l} \tau = (\tau_1 \cdot \tau_2) \wedge \\ (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ * \\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)) \end{array}\right\} \exists \tau_1, \tau_2 \\
\quad\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad\quad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2))\} \\
\quad\quad\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2) \\
\quad\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ \mathsf{j} \mapsto \mathsf{j}_1, \mathsf{j}_2\ * \\
\quad\quad\quad\quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{j}_2))\} \\
\quad\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(\mathsf{i})\ *\ \mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(\mathsf{j}))\}\Big) \\
\quad \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \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

$$\begin{array}{l}\mathsf{copytree(j;i)} = \\ \quad \textbf{if } \mathsf{isatom(i)} \textbf{ then } \mathsf{j} := \mathsf{i} \textbf{ else} \\ \quad\quad \textbf{newvar } \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2 \textbf{ in} \\ \quad\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\ \quad\quad\quad\ \mathsf{copytree(j}_1\mathsf{;i}_1)\ ;\ \mathsf{copytree(j}_2\mathsf{;i}_2)\ ;\ \mathsf{j} := \textbf{cons}(\mathsf{j}_1, \mathsf{j}_2)\big).\end{array} \tag{5.1}$$

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:

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

$$\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff } \exists i_1, i_2.\ \ i \mapsto i_1, i_2\ *\ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)).$$

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 $\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)$ would imply that τ_1 and τ_2 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 16 (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 $p\ *\ \textbf{true} \Rightarrow p$. If τ is an atom a, then

$$\begin{aligned}&\mathsf{dag}\,a\,(i)\ *\ \textbf{true} \\ &\quad\Rightarrow i = a\ *\ \textbf{true} \\ &\quad\Rightarrow i = a \\ &\quad\Rightarrow \mathsf{dag}\,a\,(i),\end{aligned}$$

since $i = a$ is pure. Otherwise, $\tau = (\tau_1 \cdot \tau_2)$, and

$$\begin{aligned}
&\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \;*\; \mathbf{true} \\
&\quad \Rightarrow \exists i_1, i_2.\;\; i \mapsto i_1, i_2 \;*\; (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)) \;*\; \mathbf{true} \\
&\quad \Rightarrow \exists i_1, i_2.\;\; i \mapsto i_1, i_2 \;* \\
&\qquad\quad ((\mathsf{dag}\,\tau_1\,(i_1) \;*\; \mathbf{true}) \wedge (\mathsf{dag}\,\tau_2\,(i_2) \;*\; \mathbf{true})) \\
&\quad \Rightarrow \exists i_1, i_2.\;\; i \mapsto i_1, i_2 \;*\; (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)) \\
&\quad \Rightarrow \mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i),
\end{aligned}$$

by the induction hypothesis for τ_1 and τ_2.

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

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

then $\tau_0 = \tau_1$ *and*

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

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

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

(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

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

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

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

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,

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

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 $\mathsf{copytree}(\mathsf{j};\mathsf{i})$. 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

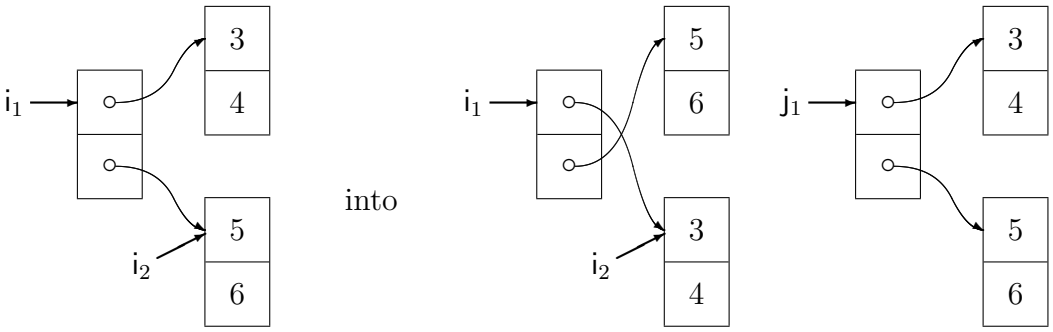
$$\{\mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau\}\ \{\mathsf{dag}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.$$

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:

$$\{\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ (\mathsf{dag}\,\tau_1(\mathsf{i}_1) \wedge \mathsf{dag}\,\tau_2(\mathsf{i}_2))\}$$
$$\mathsf{copytree}(\mathsf{j}_1;\mathsf{i}_1)\{\tau_1\}$$
$$\{\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ (\mathsf{dag}\,\tau_1(\mathsf{i}_1) \wedge \mathsf{dag}\,\tau_2(\mathsf{i}_2))\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\}.$$

(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 $\tau_1 = ((3 \cdot 4) \cdot (5 \cdot 6))$ and $\tau_2 = (5 \cdot 6)$. Then, even though it satisfies the hypothesis, the call $\mathsf{copytree}(\mathsf{i}_1, \tau; \mathsf{j}_1)$ might change the state from



where $\mathsf{dag}\ \tau_2\ (\mathsf{i}_2)$ 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 and parameters 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

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

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:

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

3. Introduce fractional permissions [32], 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:

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

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:

$$\text{AStores}_A = A \to (\text{Heaps} \to \mathbf{B})$$

$$\text{States}_{AV} = \text{AStores}_A \times \text{Stores}_V \times \text{Heaps},$$

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:

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

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 $\text{AV}(p)$ 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 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

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

and let as be an assertion store such that

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

and let

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

Then

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

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)

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

  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.

- Substitution (SUBan)

$$\frac{\mathcal{A} \gg \{p\}\ c\ \{q\}}{\big\{\mathcal{A}\big\}/\delta \ \gg \{p/\delta\}\ c\ \{q/\delta\}},$$

  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

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

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

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

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

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

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

Our proof is again an annotated instance of the first premiss of (SRPROC):

$$
\begin{array}{l}
\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau,\mathsf{p}\}\ \{\mathsf{p}\ *\ \mathsf{tree}\ \tau(\mathsf{j})\} \vdash \\
\quad \{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\} \\
\quad \textbf{if}\ \mathsf{isatom}(\mathsf{i})\ \textbf{then} \\
\qquad \{\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = \mathsf{i}\} \\
\qquad \{\mathsf{p}\ *\ (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp})\} \\
\qquad \mathsf{j} := \mathsf{i} \\
\qquad \{\mathsf{p}\ *\ (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{j} \wedge \mathbf{emp})\} \\
\quad \textbf{else} \\
\qquad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ (\tau_1 \cdot \tau_2)(\mathsf{i})\} \\
\qquad \textbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \textbf{in} \\
\qquad\quad \Big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\qquad \mathsf{p} \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ (\mathsf{dag}\ \tau_1\ (\mathsf{i}_1) \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)))\} \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\qquad \mathsf{p} \wedge (\mathbf{true}\ *\ (\mathsf{dag}\ \tau_1\ (\mathsf{i}_1) \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)))\} \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad\qquad \mathsf{p} \wedge ((\mathbf{true}\ *\ \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)) \wedge (\mathbf{true}\ *\ \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)))\} \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2) \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)\} \qquad (*) \\
\qquad\quad \left.\begin{array}{l} \{\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2) \wedge \mathsf{dag}\ \tau_1(\mathsf{i}_1)\} \\ \mathsf{copytree}(\mathsf{j}_1;\mathsf{i}_1)\{\tau_1, \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)\} \\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2))\ *\ \mathsf{tree}\ \tau_1(\mathsf{j}_1)\} \end{array}\right\} \exists \tau_1, \tau_2 \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2))\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\} \\
\qquad\quad \left.\begin{array}{l} \left.\begin{array}{l} \{\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2(\mathsf{i}_2)\} \\ \mathsf{copytree}(\mathsf{j}_2;\mathsf{i}_2)\{\tau_2, \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p}\} \\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ *\ \mathsf{tree}\ \tau_2(\mathsf{j}_2)\} \end{array}\right\} *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1) \end{array}\right\} \exists \tau_1, \tau_2 \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2)\} \\
\qquad\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2) \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ * \\
\qquad\qquad \mathsf{j} \mapsto \mathsf{j}_1, \mathsf{j}_2\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2)\} \\
\qquad\quad \{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p})\ *\ \mathsf{tree}\ (\tau_1 \cdot \tau_2)\ (\mathsf{j})\}\Big) \\
\quad \{\mathsf{p}\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}
\end{array}
$$

(Here, the assertion marked $(*)$ is obtained from the preceeding assertion by using $\mathbf{true} \ * \ \mathsf{dag}\,\tau\,(\mathsf{i}) \Rightarrow \mathsf{dag}\,\tau\,(\mathsf{i})$, which holds since $\mathsf{dag}\,\tau\,(\mathsf{i})$ is intuitionistic.)

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:

$$\begin{aligned} a/a \to \tau' &= \tau' \\ b/a \to \tau' &= b \qquad \text{when } b \in \text{Atoms} - \{a\} \\ (\tau_1 \cdot \tau_2)/a \to \tau' &= ((\tau_1/a \to \tau') \cdot (\tau_2/a \to \tau')). \end{aligned}$$

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.,

$$\begin{aligned} &\{\mathsf{tree}\,\tau\,(\mathsf{i}) \ * \ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\ &\mathsf{subst1}(\mathsf{i};\mathsf{a},\mathsf{j})\{\tau.\tau', \mathsf{p}\} \\ &\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i}) \ * \ \mathsf{p}\}, \end{aligned}$$

The procedure 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 for the procedure body:

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

$$
\begin{array}{l}
\quad \vdots \\
\textbf{else newvar } \mathsf{i}_1, \mathsf{i}_2 \textbf{ in } \big(\\
\quad \{\exists \tau_1, \tau_2, \mathsf{i}_1, \mathsf{i}_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *\ \\
\qquad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto -, -\ * \qquad (*) \\
\qquad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \left.\begin{array}{l}
\{\mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))) \wedge \mathsf{dag}\,\tau'(\mathsf{j}))\} \\
\mathsf{subst1}(\mathsf{i}_1; \mathsf{a}, \mathsf{j})\{\tau_1, \tau', \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\{\mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))))\}
\end{array}\right\} \exists \tau_1, \tau_2 \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto -, -\ * \qquad (**) \\
\qquad \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\} \\
\quad \left.\begin{array}{l}
\{\mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{p}) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\} \\
\mathsf{subst1}(\mathsf{i}_2; \mathsf{a}, \mathsf{j})\{\tau_2, \tau', \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{p})\} \\
\{\mathsf{tree}\,(\tau_2/\mathsf{a} \to \tau')\,(\mathsf{i}_2)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto -, -\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{p}))\}
\end{array}\right\} \exists \tau_1, \tau_2 \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto -, -\ * \\
\qquad \mathsf{tree}\,(\tau_2/\mathsf{a} \to \tau')\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{p})\} \\
\quad [\mathsf{i}] := \mathsf{i}_1\ ;\ [\mathsf{i}+1] := \mathsf{i}_2 \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\qquad \mathsf{tree}\,(\tau_2/\mathsf{a} \to \tau')\,(\mathsf{i}_2)\ *\ \mathsf{tree}\,(\tau_1/\mathsf{a} \to \tau')\,(\mathsf{i}_1)\ *\ \mathsf{p})\} \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{tree}\,((\tau_1/\mathsf{a} \to \tau') \cdot (\tau_2/\mathsf{a} \to \tau'))\,(\mathsf{i})\ *\ \mathsf{p})\}\big) \\
\{\mathsf{tree}\,(\tau/\mathsf{a} \to \tau')\,(\mathsf{i})\ *\ \mathsf{p}\}
\end{array}
$$

The following argument explains why the assertion marked $(*)$ implies the precondition of the application of (EQan) that follows it. From $(*)$:

$$\{\exists\tau_1,\tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto -,- \ *\\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\},$$

we obtain, since $\tau = (\tau_1 \cdot \tau_2)$ is pure,

$$\{\exists\tau_1,\tau_2.\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge\\ (\mathsf{i} \mapsto -,- \ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))))\}.$$

Then by the semidistributive law for $*$ and $\wedge$,

$$\{\exists\tau_1,\tau_2.\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge\\ (\mathsf{i} \mapsto -,- \ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\\ \wedge\ (\mathsf{i} \mapsto -,- \ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ \mathsf{dag}\,\tau'\,(\mathsf{j})))\},$$

and by the monotonicity of $*$,

$$\{\exists\tau_1,\tau_2.\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge\\ (\mathsf{i} \mapsto -,- \ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j})))\\ \wedge\ (\mathbf{true}\ *\ \mathsf{dag}\,\tau'\,(\mathsf{j})))\},$$

and since $\mathbf{dag}\,\tau'\,(\mathsf{j})$ is intuitionistic,

$$\{\exists\tau_1,\tau_2.\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ (\tau = (\tau_1 \cdot \tau_2) \wedge\\ (\mathsf{i} \mapsto -,- \ *\ \mathsf{tree}\,\tau_2\,(\mathsf{i}_2)\ *\ (\mathsf{p} \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))) \wedge \mathsf{dag}\,\tau'\,(\mathsf{j}))\}.$$

A similar argument applies to the assertion marked $(**)$.

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

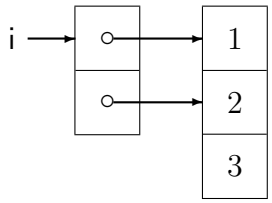
$$\begin{array}{l}\mathsf{subst1}(\mathsf{i};\mathsf{a},\mathsf{j}) =\\ \quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then\ if}\ \mathsf{i} = \mathsf{a}\ \mathbf{then}\ \mathsf{copytree}(\mathsf{i}\,;\mathsf{j})\ \mathbf{else\ skip}\\ \quad \mathbf{else\ newvar}\ \mathsf{i}_1,\mathsf{i}_2\ \mathbf{in}\ (\mathsf{i}_1 := [\mathsf{i}]\,;\mathsf{i}_2 := [\mathsf{i}+1]\,;\\ \qquad \mathsf{subst1}(\mathsf{i}_1;\mathsf{a},\mathsf{j})\,;\mathsf{subst1}(\mathsf{i}_2;\mathsf{a},\mathsf{j})\,;[\mathsf{i}] := \mathsf{i}_1\,;[\mathsf{i}+1] := \mathsf{i}_2).\end{array}$$

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.

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

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

We also introduce the following abbreviations:

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

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

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

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

$$\text{iff } e \overset{!}{\mapsto} e_1, \ldots, e_n \ * \ \mathbf{true}.$$

Axiom schema for reasoning about these new assertions include:

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

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

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

$$e \overset{!}{\hookrightarrow} e_1, \ldots, e_m \wedge e' \overset{!}{\hookrightarrow} e'_1, \ldots, e'_n \wedge e \neq e' \Rightarrow$$
$$e \overset{!}{\mapsto} e_1, \ldots, e_m \ * \ e' \overset{!}{\mapsto} e'_1, \ldots, e'_n \ * \ \mathbf{true}.$$

(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 16

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

$$|a| = [a] \quad \text{when } a \text{ is an atom}$$
$$|(t_1 \cdot t_2)| = |\tau_1| \cdot |\tau_2|.$$

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 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

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

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

An Introduction
to Separation Logic

October 23, 2008

Chapter 6

Iterated Separating Conjunction

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

6.1 A New Form of Assertion

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Arrays

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

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

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

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

- The local nonoverwriting form (ALLOCNOL)

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

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

- The global nonoverwriting form (ALLOCNOG)

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

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

- The local form (ALLOCL)

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

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

- The global form (ALLOCG)

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

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

- The backward-reasoning form (ALLOCBR)

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

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

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

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

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

Notice that, since the length of a sequence is never negative, the assertion $\mathsf{array}\,\alpha\,(\mathsf{a},\mathsf{b})$ implies that $\mathsf{a} \leq \mathsf{b}+1$. In fact, it would be consistent to define a to b to represent the empty sequence when $\mathsf{a} > \mathsf{b}+1$ (as well as $\mathsf{a} = \mathsf{b}+1$), but we will not use such "irregular" representations in these notes. (An integrated approach to regular and irregular represetations is discussed in Reference [41, Chapter 2].)

This new form of assertion satisfies the following axioms:

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

6.3 Partition

As an example, we present a program that, given an array representing a sequence, and a *pivot* value r, rearranges the sequence so that it splits into two contiguous parts, containing values smaller or equal to r and values larger than r, respectively. (This is a variant of the well-known program "Partition" by C. A. R. Hoare [50].)

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

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

$\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{d}-1)\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b}))$
$\quad\wedge\ \alpha_1{\cdot}\alpha_2{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r}\}$

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

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

$\qquad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{c}+1 \mapsto \mathsf{x}\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+2,\mathsf{d}-1)$
$\qquad\quad *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b})) \wedge \alpha_1{\cdot}\mathsf{x}{\cdot}\alpha_2{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1{\cdot}\mathsf{x}\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r}\}$

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

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

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

$\qquad\quad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+1,\mathsf{d}-2)\ *\ \mathsf{d}-1 \mapsto \mathsf{y}$
$\qquad\quad\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b})) \wedge \alpha_1{\cdot}\alpha_2{\cdot}\mathsf{y}{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\mathsf{y}{\cdot}\alpha_3\} >^* \mathsf{r}\}$

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

$\qquad\mathbf{else}$

$\qquad\quad\{\exists\alpha_1,\alpha_2,\alpha_3.\ (\mathsf{array}\ \alpha_1\,(\mathsf{a},\mathsf{c})\ *\ \mathsf{c}+1 \mapsto \mathsf{x}$
$\qquad\quad\quad *\ \mathsf{array}\ \alpha_2\,(\mathsf{c}+2,\mathsf{d}-2)\ *\ \mathsf{d}-1 \mapsto \mathsf{y}\ *\ \mathsf{array}\ \alpha_3\,(\mathsf{d},\mathsf{b}))$
$\qquad\quad\quad\wedge\ \alpha_1{\cdot}\mathsf{x}{\cdot}\alpha_2{\cdot}\mathsf{y}{\cdot}\alpha_3 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \{\alpha_3\} >^* \mathsf{r} \wedge \mathsf{x} > \mathsf{r} \wedge \mathsf{y} \leq \mathsf{r}\}$

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

$\{\exists\alpha_1,\alpha_2.\ (\mathsf{array}\ \alpha_1(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2(\mathsf{c}+1,\mathsf{b}))$
$\quad\wedge\ \alpha_1{\cdot}\alpha_2 \sim \alpha \wedge \{\alpha_1\} \leq^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}.$

For the most part, the reasoning here is straightforward. It should be noticed, however, that the assertion following the second **else** depends upon the validity of the implication

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

It is also straightforward to encapsulate the above program as a nonre-

cursive procedure. If we define

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

then partition satisfies

$$\begin{array}{l}
\{\mathsf{array}\ \alpha(\mathsf{a},\mathsf{b})\} \\
\mathsf{partition}(\mathsf{c};\mathsf{a},\mathsf{b},\mathsf{r})\{\alpha\} \\
\{\exists\alpha_1,\alpha_2.\ (\mathsf{array}\ \alpha_1(\mathsf{a},\mathsf{c})\ *\ \mathsf{array}\ \alpha_2(\mathsf{c}+1,\mathsf{b})) \\
\qquad \wedge\ \alpha_1{\cdot}\alpha_2 \sim \alpha \wedge \{\alpha_1\} \le^* \mathsf{r} \wedge \mathsf{r} <^* \{\alpha_2\}\}.
\end{array}$$

6.4 From Partition to Quicksort

We can use the procedure partition to define a version of the recursive sorting procedure called quicksort (which is again a variant of a well-known algorithm by Hoare [51]).

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

$$\begin{array}{l}
\{\mathsf{array}\ \alpha\,(\mathsf{a},\mathsf{b})\} \\
\mathsf{quicksort}(;\mathsf{a},\mathsf{b})\{\alpha\} \\
\{\exists\beta.\ \mathsf{array}\ \beta\,(\mathsf{a},\mathsf{b}) \wedge \beta \sim \alpha \wedge \mathbf{ord}\ \beta\}.
\end{array} \tag{6.8}$$

(Notice that quicksort does not modify any variables.)

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

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

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

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

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

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

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

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

6.5 Another Cyclic Buffer

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

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

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

We will also use the following variables:

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

Let R abbreviate the assertion

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

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

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

and

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

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

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

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

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

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

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

$[j] := x\,;$

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

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

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

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

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

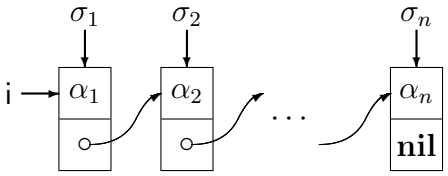
$\{((\bigodot_{k=0}^{m} i \oplus k \mapsto (\alpha{\cdot}x)_{k+1}) \ * \ (\bigodot_{k=0}^{n-m-2} j \oplus k \oplus 1 \mapsto -))$ (6.4)
$\quad \wedge m + 1 = \#(\alpha{\cdot}x) \wedge R \wedge m < n\}$

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

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

6.6 Connecting Two Views of Simple Lists

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



From the definitions

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

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

and

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

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

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

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

The proof is by structural induction on α.

6.7 Specifying a Program for Subset Lists

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

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

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

In this section, we reason about an iterative version of the algorithm.

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

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

Then our program will satisfy the specification

$$\begin{array}{l} \{\mathsf{list}\,\alpha\,\mathsf{i}\} \\ \text{“Set } \mathsf{j} \text{ to list of lists of subsets of } \mathsf{i}\text{”} \\ \{\exists\sigma, \beta.\ \ \mathsf{ss}(\alpha^\dagger, \sigma) \wedge (\mathsf{list}\,\beta\,\mathsf{j}\ *\ (Q(\sigma,\beta) \wedge R(\beta)))\}. \end{array} \tag{6.9}$$

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

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

where ext_a is a function that prefixes a to every element of its argument:

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

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

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

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

The formula $R(\beta)$ uses the iterated separating conjunction to assert that the final element of β is the empty list, and that every previous element is a list consisting of a single record followed by a list occurring later in β:

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

We will also need to define the formula

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

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

It is immediately evident that:

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

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

Less trivial are the following:

Proposition 19

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

Proof

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

END OF PROOF

Proposition 20

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

PROOF Let

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

Then

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

Here (a) implies (b) by the semidistributive law for $*$ and $\forall$, while, as the reader may verify, (b) implies (c) since $n = \#\beta$ is larger than zero.

END OF PROOF

Proposition 21

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

PROOF

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

END OF PROOF

From the last two propositions, we have

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

Using these results, we can verify a program satisfying (6.9). The program contains two nested **while** commands. The invariant of the outer **while** is

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

and the invariant of the inner **while** is

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

At the completion of the inner **while**, we will have

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

In full detail, the annotated specification is:

$\{\mathsf{list}\ \alpha\ \mathsf{i}\}$
$\mathsf{j} := \mathbf{cons}(\mathbf{nil}, \mathbf{nil})$;
$\{\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{j} \mapsto \mathbf{nil}, \mathbf{nil}\}$
$\{\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ [\mathbf{nil}]\ \mathsf{j}\}$
$\{\mathsf{ss}(\epsilon, [\epsilon]) \wedge (\mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ [\mathbf{nil}]\ \mathsf{j}\ *\ (Q([\epsilon], [\mathbf{nil}]) \wedge R([\mathbf{nil}])))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta.\ \ \alpha'^{\dagger} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\}$
while $\mathsf{i} \neq \mathbf{nil}$ **do** *Body of the Outer* **while** ;
$\{\exists \sigma, \beta.\ \ \mathsf{ss}(\alpha^{\dagger}, \sigma) \wedge (\mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\}$,

where the body of the outer **while** is:

$\{\exists \alpha', \alpha'', \sigma, \beta, \mathsf{a}, \mathsf{k}.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathsf{list}\ \alpha''\ \mathsf{k}\ *\ \mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\}$
$\big(\mathsf{a} := [\mathsf{i}]$; $\mathsf{k} := [\mathsf{i} + 1]$; **dispose** i ; **dispose** $\mathsf{i} + 1$; $\mathsf{i} := \mathsf{k}$;
$\{\exists \alpha', \alpha'', \sigma, \beta.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \epsilon\ (\mathsf{j}, \mathsf{j})\ *\ \mathsf{lseg}\ \epsilon\ (\mathsf{j}, \mathsf{j})\ *\ \mathsf{list}\ \beta\ \mathsf{j}\ *$
$\quad\quad (Q(\sigma, \epsilon \cdot \beta) \wedge R(\epsilon \cdot \beta))\ *\ W(\epsilon, \epsilon, \mathsf{a}))\}$

$\mathsf{l} := \mathsf{j}\,;\, \mathsf{m} := \mathsf{j}\,;$
$\{\exists \alpha', \alpha'', \sigma, \beta.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \epsilon\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \epsilon\ (\mathsf{j}, \mathsf{m})\ *\ \mathsf{list}\ \beta\ \mathsf{m}\ *$
$\quad\quad (Q(\sigma, \epsilon \cdot \beta) \wedge R(\epsilon \cdot \beta))\ *\ W(\epsilon, \epsilon, \mathsf{a}))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta', \beta'', \gamma.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta'\ (\mathsf{j}, \mathsf{m})\ *\ \mathsf{list}\ \beta''\ \mathsf{m}\ *$
$\quad\quad (Q(\sigma, \beta' \cdot \beta'') \wedge R(\beta' \cdot \beta''))\ *\ W(\beta', \gamma, \mathsf{a}))\}$
while $\mathsf{m} \neq$ **nil do** *Body of the Inner* **while** ;
$\{\exists \alpha', \alpha'', \sigma, \beta', \gamma.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{list}\ \beta'\ \mathsf{j}\ *$
$\quad\quad (Q(\sigma, \beta') \wedge R(\beta'))\ *\ W(\beta', \gamma, \mathsf{a}))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta', \gamma.\ \ (\mathsf{a} \cdot \alpha')^{\dagger} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\mathsf{a} \cdot \alpha', (\mathsf{ext}_{\mathsf{a}}\sigma)^{\dagger} \cdot \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{list}\ (\gamma \cdot \beta')\ \mathsf{l}\ *\ (Q((\mathsf{ext}_{\mathsf{a}}\sigma)^{\dagger} \cdot \sigma, \gamma \cdot \beta') \wedge R(\gamma \cdot \beta')))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta.\ \ \alpha'^{\dagger} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{l}\ *\ (Q(\sigma, \beta) \wedge R(\beta)))\}$
$\mathsf{j} := \mathsf{l}\big),$

and the body of the inner **while** is:

$\{\exists \alpha', \alpha'', \sigma, \beta', \beta'', \gamma, \mathsf{b}, \mathsf{m}''.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta'\ (\mathsf{j}, \mathsf{m})\ *\ \mathsf{m} \mapsto \mathsf{b}, \mathsf{m}''\ *$
$\quad\quad \mathsf{list}\ \beta''\ \mathsf{m}''\ *\ (Q(\sigma, \beta' \cdot \mathsf{b} \cdot \beta'') \wedge R(\beta' \cdot \mathsf{b} \cdot \beta''))\ *\ W(\beta', \gamma, \mathsf{a}))\}$
$\big(\mathsf{b} := [\mathsf{m}]\,;$
$\{\exists \alpha', \alpha'', \sigma, \beta', \beta'', \gamma, \mathsf{m}''.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta'\ (\mathsf{j}, \mathsf{m})\ *\ \mathsf{m} \mapsto \mathsf{b}, \mathsf{m}''\ *$
$\quad\quad \mathsf{list}\ \beta''\ \mathsf{m}''\ *\ (Q(\sigma, \beta' \cdot \mathsf{b} \cdot \beta'') \wedge R(\beta' \cdot \mathsf{b} \cdot \beta''))\ *\ W(\beta', \gamma, \mathsf{a}))\}$
$\mathsf{m} := [\mathsf{m} + 1]\,;$
$\{\exists \alpha', \alpha'', \sigma, \beta', \beta'', \gamma, \mathsf{m}'.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta'\ (\mathsf{j}, \mathsf{m}')\ *\ \mathsf{m}' \mapsto \mathsf{b}, \mathsf{m}\ *$
$\quad\quad \mathsf{list}\ \beta''\ \mathsf{m}\ *\ (Q(\sigma, \beta' \cdot \mathsf{b} \cdot \beta'') \wedge R(\beta' \cdot \mathsf{b} \cdot \beta''))\ *\ W(\beta', \gamma, \mathsf{a}))\}$
$\{\exists \alpha', \alpha'', \sigma, \beta', \beta'', \gamma.\ \ \alpha'^{\dagger} \cdot \mathsf{a} \cdot \alpha'' = \alpha \wedge \mathsf{ss}(\alpha', \sigma) \wedge$
$\quad (\mathsf{list}\ \alpha''\ \mathsf{i}\ *\ \mathsf{lseg}\ \gamma\ (\mathsf{l}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta' \cdot \mathsf{b}\ (\mathsf{j}, \mathsf{m})\ *\ \mathsf{list}\ \beta''\ \mathsf{m}\ *$

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

Exercise 17

Derive the axiom scheme

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

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

Exercise 18

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

- The ghost-variable global form (ALLOCGG)

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

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

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

Bibliography

[1] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[2] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[3] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[4] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

[5] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM.

[6] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[7] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, Boston, Massachusetts, 2002. (to appear).

[8] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg,

editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2001. Springer-Verlag.

[9] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416, Berlin, 2002. Springer-Verlag.

[10] Hongseok Yang. *Local Reasoning for Stateful Programs*. Ph. D. dissertation, University of Illinois, Urbana-Champaign, Illinois, July 2001.

[11] John C. Reynolds and Peter W. O'Hearn. Reasoning about shared mutable data structure (abstract of invited lecture). In Fritz Henglein, John Hughes, Henning Makholm, and Henning Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, page 7. IT University of Copenhagen, 2001. The slides for this lecture are available at ftp://ftp.cs.cmu.edu/user/jcr/spacetalk.ps.gz.

[12] Hongseok Yang. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In Fritz Henglein, John Hughes, Henning Makholm, and Henning Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, pages 41–68. IT University of Copenhagen, 2001.

[13] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, New York, 2004. ACM Press.

[14] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. To appear in the ACM Transactions on Programming Languages and Systems, 2008.

[15] Carsten Varming and Lars Birkedal. Higher-order separation logic in Isabelle/HOL. To appear in the Proceedings of the 24th Annual Conference on Mathematical Foundations of Programming Semantics, Electronic Notes in Theoretical Computer Science, 2008.

[16] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119, Berlin, 2001. Springer-Verlag.

[17] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society.

[18] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280, New York, 2004. ACM Press.

[19] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. To appear in the ACM Transactions on Programming Languages and Systems, 2009.

[20] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings Twentieth Annual IEEE Symposium on Logic in Computer Science*, Los Alamitos, California, 2005. IEEE Computer Society.

[21] Ivana Mijajlović and Noah Torp-Smith. Refinement in a separation context. In *SPACE 2004: Informal Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.

[22] Ivana Mijajlović, Noah Torp-Smith, and Peter W. O'Hearn. Refinement and separation contexts. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 421–433, Berlin, 2004. Springer-Verlag.

[23] Matthew J. Parkinson and Gavin Bierman. Separation logic and abstraction. In *Conference Record of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–258, New York, 2005. ACM Press.

[24] Matthew J. Parkinson. *Local Reasoning in Java*. Ph. D. dissertation, University of Cambridge, Cambridge, England, August 2005.

[25] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 — Concurrency Theory, Proceedings of the 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67, Berlin, 2004. Springer-Verlag.

[26] Stephen D. Brookes. A semantics for concurrent separation logic. In *CONCUR 2004 — Concurrency Theory, Proceedings of the 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34, Berlin, 2004. Springer-Verlag.

[27] Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *Conference Record of POPL 2007: The 34nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, 2007. ACM Press.

[28] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 — Concurrency Theory, Proceedings of the 18th International Conference*, Lecture Notes in Computer Science, Berlin, 2007. Springer-Verlag.

[29] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50:101–127, 2004.

[30] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109, Berlin, 2004. Springer-Verlag.

[31] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, 2003. Springer-Verlag.

[32] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Conference*

Record of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 259–270, New York, 2005. ACM Press.

[33] Richard Bornat. Variables as resource in separation logic. *Electronic Notes in Computer Science, 21st Annual Conference on Mathematical Foundations of Programming Semantics*, 155:247–276, 2005.

[34] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines and higher order separation logic. In *Proceedings of ESOP 2005: The European Symposium on Programming*, pages 233–247, 2005.

[35] Luis Caires and L. Monteiro. Verifiable and executable specifications of concurrent objects in $\mathcal{L}_\pi$. In C. Hankin, editor, *Programming Languages and Systems — ESOP '98*, volume 1381 of *Lecture Notes in Computer Science*, pages 42–56, Berlin, 1998. Springer-Verlag.

[36] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, 2000. ACM.

[37] G. Conforti, Giorgio Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The query language TQL. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, Madison, Wisconsin, 2002.

[38] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In Matthew Hennessy and P. Widmayer, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, Berlin, 2002. Springer-Verlag.

[39] Luca Cardelli and Giorgio Ghelli. A query language based on the ambient logic. In D. Sands, editor, *Programming Languages and Systems — ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 1–22, Berlin, 2001. Springer-Verlag.

[40] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Conference Record of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–282, New York, 2005. ACM Press.

[41] John C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.

[42] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71, London, 1972. Academic Press.

[43] Susan Speer Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[44] Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, Amsterdam, 1952.

[45] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.

[46] Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The Foundations of Program Verification*. Wiley, Chichester, England, second edition, 1987.

[47] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[48] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[49] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[50] C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, July 1961.

[51] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.