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

In 1972, Burstall [1] gave correctness proofs for imperative programs that alter data structures, by using a novel kind of assertion that he called a "distinct nonrepeating tree system"; this approach was extended by Kowaltowski [2]. In 1975, Cook and Oppen [3, 4] devised an more general approach by extending Hoare logic with extremely complicated inference rules. Then, in 1981, J. M. Morris [5] extended weakest-precondition logic by generalizing the notion of substitution.

In the late 80's Mason and Talcott [6, 7, 8] investigated reasoning about program equivalence for LISP-like functional languages where expression evaluation can alter data structures as a side effect; more recently they and others [9, 10] have extended this approach to a logic for reasoning about programs.

Pitts and Stark [11] have studied operational reasoning about an ML-like language with data-altering expressions. (In this work, however, only simple values can be stored at locations; not structured values that themselves contain locations.) This research builds upon earlier work by Stark [12, 13, 14] on languages that create local names.

Separation logic was inspired by Burstall's ideas, which were pleasantly compatible with Hoare logic [15, 16], as well as earlier work by Floyd [17], Naur [18], and Good [19]. Burstall's "distinct nonrepeating tree system" was a sequence of assertions, written $P_1 * \cdots * P_n$ in the notation of these notes, where each P_i described a distinct region of storage, so that an assignment to a single location could change only one of the P_i. This seems to be the first occurrence of the idea that an assertion about a state can be built out of assertions about disjoint substates — the essence of the notion of separating conjunction.

In lectures in the fall of 1999, Reynolds described separating conjunction explicitly, and embedded it in a flawed extension of Hoare logic. Soon thereafter, a sound intuitionistic version of the logic was discovered independently by Ishtiaq and O'Hearn [20] and by Reynolds [21]. Realizing that this logic was an instance of the logic of bunched implications [22, 23], 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 [24, 25, 20].

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, Reynolds generalized the logic to permit reasoning about unrestricted address arithmetic, by regarding addresses as integers which refer to individual fields [24, 26]. 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, such as the Schorr-Waite algorithm [27], treated in [28, 29], and the Cheney copying garbage collector [30], treated in [31, 32]. 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 [33, 29]. 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 [33].

2. As discussed in Section 1.8 and Chapter 6, an iterative form of separating conjunction has been introduced to reason about arrays [34].

3. The logic has been extended to procedures with global variables, where a "second-order frame rule" permits reasoning with information hiding [35, 36].

4. A rich type system called "Hoare Type Theory" has been developed, in which the specifications of Hoare logic and separation logic reappear as types [37, 38, 39, 40]. This approach has been applied to an Algol-like language with higher-order procedures but no assignment to variables (as opposed to mutation of heap locations) and only integers and addresses as values in the heap [41]. This setting leads to higher-order versions of the frame rule. It is also possible to introduce more general forms of abstraction and information hiding [42, 43] that are based on relational parametricity (as in the polymorphic lambda calculus [44]). This work draws upon the combination of relational reasoning and FM-domain theory [45], which in turn drew upon the work of Pitts and Stark [11, 12, 13, 14].

5. Separation logic has been extended to a higher-order logic [46]. This permits the user to define abstract data types and their implementation directly in the logic (rather than as extensions to the logic).

6. The logic has been extended to deal with heaps that map addresses into procedures [47, 48]. In combination with the extension to a higher-order logic, this provides the concepts that are needed to provide higher-order reasoning about an ML-like language, which is currently being explored by N. Krishnaswami.

7. The logic has been integrated with data refinement [49, 50, 51], and with object-oriented programming (i.e., with a subset of Java) [52, 53, 54].

8. Higher-order extensions of the logic have been use to specify design patterns [55] used in object-oriented programming, and to prove the correctness of implementations of these patterns [56, 57, 38, 58].

9. As discussed in Section 1.10, 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 [59, 60]. Further extensions have been made to nonblocking algorithms [61]. The concept of rely/guarantee reasoning (originally introduced in the 1980's [62]) has led to more general approaches to such algorithms [63, 64, 65]. Other work on concurrency includes [66].

10. The logic has been extended to deal with byte-size data, pointer alignment restrictions, finite-arithmetic, and other complications imposed by low-level languages such as C [67, 68].

11. In the context of proof-carrying code, separation logic has inspired work on proving run-time library code for dynamic allocation [69].

12. Separation logic has been repeatedly applied to the design of automatic systems for insuring memory safety by means of shape analysis [70, 71, 72, 73, 74, 75, 76, 77].

13. As discussed in Section 1.11, fractional permissions (in the sense of Boyland [78]) 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 [79]. This approach has also been applied to program variables [80, 81].

14. The concept that primitive operations are local actions, which makes local reasoning possible, has been used to formulate an abstract form of separation logic which unifies ordinary separation logic with the systems that use fractional permissions (as well as certain kinds of Petri nets) [82].

15. Isabelle/HOL [83, 84] has been used to implement separation logic [85, 68, 32]. The work in [68] was built upon an earlier general system for Hoare logic [86].

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 [87, 88, 89, 90, 91, 92].

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 [15, 16], extended with new commands for the manipulation of mutable shared data structures:

$\langle \text{comm} \rangle ::= \cdots$

$\mid \langle \text{var} \rangle := \mathbf{cons}(\langle \text{exp} \rangle, \ldots, \langle \text{exp} \rangle)$	allocation
$\mid \langle \text{var} \rangle := [\langle \text{exp} \rangle]$	lookup
$\mid [\langle \text{exp} \rangle] := \langle \text{exp} \rangle$	mutation
$\mid \mathbf{dispose}\ \langle \text{exp} \rangle$	deallocation

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 either imposed restricted formats on the records in the heap, to reflect the specific usage of the program they are specifying [28, 29, 31], or they have used a programming language with a type system that can describe the layout of records.)

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:

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

(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:

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

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

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

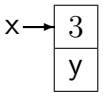
$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 \ * \ \cdots \ * \ e+n-1 \hookrightarrow e_n$$

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

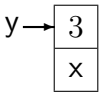
Notice that assertions of the form $e \mapsto e'$, $e \mapsto -$, and $e \mapsto e_1, \ldots, e_n$ determine the extent (i.e., domain) of the heap they describe, while those of the form $e \hookrightarrow e'$ and $e \hookrightarrow e_1, \ldots, e_n$ do not. (Technically, the former are said to be *precise*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

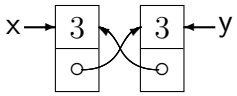
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β, α is an address, and the heap maps α into 3 and $\alpha+1$ into β).



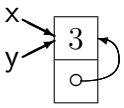
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x.



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.

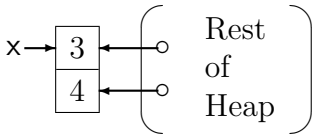


5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

Separating implication is somewhat more subtle, but is illustrated by the following example (due to O'Hearn): Suppose the assertion p asserts various

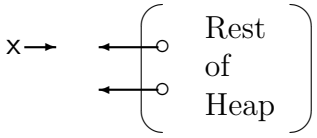
conditions about the store and heap, including that the store maps x into the address of a record containing 3 and 4:

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



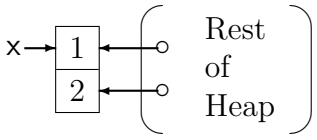
Then $(\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p$ asserts that, if one were to add to the current heap a disjoint heap consisting of a record at address x containing 3 and 4, then the resulting heap would satisfy p. In other words, the current heap is like that described by p, except that the record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

Store : $\mathsf{x}\colon\alpha, \ldots$
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 : $\mathsf{x}\colon\alpha, \ldots$
Heap : $\alpha\colon 1, \alpha+1\colon 2,$
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:

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

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:

$$p \Rightarrow p \ * \ p \qquad \text{(Contraction — unsound)}$$
$$p \ * \ q \Rightarrow p \qquad \text{(Weakening — unsound)}$$

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 [20] 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 [24].

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. [20].

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:

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

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

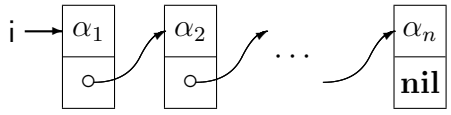
To specify a program adequately, it is usually necessary to describe more than the form of its structures or the sharing patterns between them; one must relate the states of the program to the abstract values that they denote. For instance, to specify the list-reversal program in Section 1.1, it would 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 [28, 29] of the Schorr-Waite algorithm [27] for marking structures that contain sharing and cycles. This proof uses the older form of classical separation logic [20] 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:

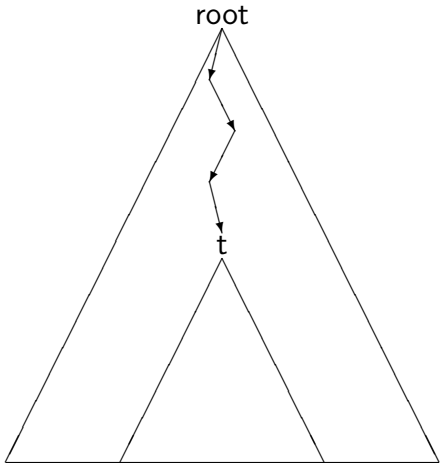
$$\begin{aligned}\mathsf{noDanglingR} \stackrel{\text{def}}{=} \forall \mathsf{x}, \mathsf{l}, \mathsf{r}.\ (\mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, -, -) \Rightarrow \\ \mathsf{noDangling}(\mathsf{l}) \wedge \mathsf{noDangling}(\mathsf{r}).\end{aligned}$$

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:

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

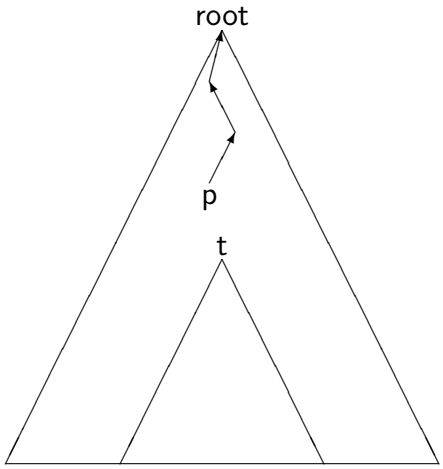
At the point in the computation described by this invariant, the value of the variable t indicates the current subheap that is about to be scanned. At the

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 [93, 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 [94] and Owicki and Gries [95].

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:

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

Associated with the resource buf is an invariant:

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

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:

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

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 [79] has opened this possibility by introducing the concept of permissions, originally devised by John Boyland [78].

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.