Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds[*]
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, 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. (A partial bibliography is given in Reference [28].)

The problem faced by these approaches is that the correctness of a program that mutates data structures usually depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

$$\begin{array}{l} \mathsf{j} := \mathbf{nil}\,;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do} \\ \qquad (\mathsf{k} := [\mathsf{i}+1]\,;\ [\mathsf{i}+1] := \mathsf{j}\,;\ \mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}). \end{array}$$

(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 **list** $\alpha\ \mathsf{i}$ is defined by induction on the length of α:

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

(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:

$$\begin{array}{l} (\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta) \\ \quad \wedge\ (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}), \end{array} \tag{1}$$

where

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

Even worse, suppose there is some other list x, representing a sequence γ, that is not supposed to be affected by

[*] Portions of the author's own research described in this survey were supported by National Science Foundation Grant CCR-9804014, and by the Basic Research in Computer Science (`http://www.brics.dk/`) Centre of the Danish National Research Foundation.

Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)
1043-6871/02 $17.00 © 2002 **IEEE**

the execution of our program. Then it must not share with either i or j, so that the invariant becomes

$$\begin{aligned}&(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta) \wedge \mathsf{list}\,\gamma\,\mathsf{x}\\ &\quad\wedge (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})\\ &\quad\wedge (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{x},\mathsf{k}) \wedge (\mathbf{reach}(\mathsf{i},\mathsf{k}) \vee \mathbf{reach}(\mathsf{j},\mathsf{k}))\\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathsf{k} = \mathbf{nil}).\end{aligned} \tag{2}$$

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

The key to avoiding this difficulty is to introduce a novel logical operation $P * Q$, called *separating conjunction* (or sometimes *independent* or *spatial* conjunction), that asserts that P and Q hold for *disjoint* portions of the addressable storage. In effect, the prohibition of sharing is built into this operation, so that Invariant (1) can be written as

$$(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} * \mathsf{list}\,\beta\,\mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta, \tag{3}$$

and Invariant (2) as

$$(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} * \mathsf{list}\,\beta\,\mathsf{j} * \mathsf{list}\,\gamma\,\mathsf{x}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta. \tag{4}$$

In fact, one can go further: Using an inference rule called the “frame rule”, one can infer directly that the program does not affect the list x from the fact that assertions such as (3) do not refer to this list.

The central concept of a separating conjunction is implicit in Burstall’s early idea of a “distinct nonrepeating tree system” [2]. In lectures in the fall of 1999, I described the concept explicitly, and embedded it in a flawed extension of Hoare logic [16, 17]. Soon thereafter, an intuitionistic logic based on this idea was discovered independently by Ishtiaq and O’Hearn [19] and by myself [28]. Realizing that this logic was an instance of the logic of bunched implications [23, 26], 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.

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. More recently, I generalized the logic to permit reasoning about unrestricted address arithmetic, by regarding addresses as integers which refer to individual fields [24, 30, 29]. It is this form of the logic that will be described and used in most of the present paper. We will also describe O’Hearn’s frame rule [24, 35, 34, 19], which permits local reasoning about components of programs.

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 logics*, both for the extension of predicate calculus with the separation operators and the resulting extension of Hoare logic. A more precise name might be *storage separation logics*, since it is becoming apparent that the underlying idea can be generalized to describe the separation of other kinds of resources [3, 11, 12, 9, 10].

2. The Programming Language

The programming language we will use is the simple imperative language originally axiomatized by Hoare [16, 17], extended with new commands for the manipulation of mutable shared data structures:

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

Semantically, we extend computational states to contain two components: a store (or 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:

$$\begin{aligned}&\text{Values} = \text{Integers} \cup \text{Atoms} \cup \text{Addresses}\\ &\qquad\text{where Integers, Atoms, and Addresses are disjoint}\\ &\text{Heaps} = \bigcup_{A \overset{\text{fin}}{\subseteq} \text{Addresses}} (A \to \text{Values}^{+}).\end{aligned}$$

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:

$$\begin{aligned}\mathrm{Values} &= \mathrm{Integers}\\ \mathrm{Atoms} \cup \mathrm{Addresses} &\subseteq \mathrm{Integers}\end{aligned}$$

where Atoms and Addresses are disjoint

$$\mathrm{Heaps} = \bigcup_{A \overset{\text{fin}}{\subseteq} \mathrm{Addresses}} (A \to \mathrm{Values}).$$

(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.) In both versions of the logic, we assume

$$\begin{aligned}\mathbf{nil} &\in \mathrm{Atoms}\\ \mathrm{Stores}_V &= V \to \mathrm{Values}\\ \mathrm{States}_V &= \mathrm{Stores}_V \times \mathrm{Heaps},\end{aligned}$$

where V is a finite set of variables.

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:

$$[\![e \in \langle \mathrm{exp} \rangle]\!]_{\mathrm{exp}} \in (\bigcup_{V \overset{\text{fin}}{\supseteq} \mathrm{FV}(e)} \mathrm{Stores}_V) \to \mathrm{Values}$$

$$[\![b \in \langle \mathrm{boolexp} \rangle]\!]_{\mathrm{bexp}} \in (\bigcup_{V \overset{\text{fin}}{\supseteq} \mathrm{FV}(b)} \mathrm{Stores}_V) \to \{\mathbf{true}, \mathbf{false}\}$$

(where $\mathrm{FV}(p)$ 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 $\mathbf{cons}$ or $[-]$, that refer to the heap. It follows that none of the new heap-manipulating commands are instances of the simple assignment command $\langle \mathrm{var} \rangle := \langle \mathrm{exp} \rangle$ (even though we write them with the familiar operator $:=$). In fact, they 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 $\mathbf{dispose}\ v$) *modify* v.

A simple way to define the meaning of the new commands is by small-step operational semantics, i.e., by defining a transition relation $\rightsquigarrow$ between *configurations*, which are either

- *nonterminal*: A command-state pair $\langle c, (s, h) \rangle$, where $\mathrm{FV}(c) \subseteq \mathrm{dom}\, s$.
- *terminal*: A state (s, h) or $\mathbf{abort}$.

We write $\gamma \rightsquigarrow^* \gamma'$ to indicate that there is a finite sequence of transitions from γ to γ', and $\gamma \uparrow$ to indicate that there is an infinite sequence of transitions beginning with γ.

In this semantics, the heap-manipulating commands are specified by the following inference rules:

- Allocation

$$\frac{}{\begin{array}{l}\langle v := \mathbf{cons}(e_1, \ldots, e_n), (s, h) \rangle\\ \quad \rightsquigarrow ([\, s \mid v\colon \ell \,], [\, h \mid \ell\colon [\![e_1]\!]_{\mathrm{exp}} s \mid \ldots \mid \ell{+}n{-}1\colon [\![e_n]\!]_{\mathrm{exp}} s \,]),\end{array}}$$

  where $\ell, \ldots, \ell + n - 1 \in \mathrm{Addresses} - \mathrm{dom}\, h$.

- Lookup

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle v := [e], (s, h) \rangle \rightsquigarrow ([\, s \mid v\colon h([\![e]\!]_{\mathrm{exp}} s) \,], h),}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle v := [e], (s, h) \rangle \rightsquigarrow \mathbf{abort}.}$$

- Mutation

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle [e] := e', (s, h) \rangle \rightsquigarrow (s, [\, h \mid [\![e]\!]_{\mathrm{exp}} s\colon [\![e']\!]_{\mathrm{exp}} s \,]),}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle [e] := e', (s, h) \rangle \rightsquigarrow \mathbf{abort}.}$$

- Deallocation

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle \mathbf{dispose}\ e, (s, h) \rangle \rightsquigarrow (s, h\rceil(\mathrm{dom}\, h - \{[\![e]\!]_{\mathrm{exp}} s\})),}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle \mathbf{dispose}\ e, (s, h) \rangle \rightsquigarrow \mathbf{abort}.}$$

(Here $[\, f \mid x\colon a \,]$ denotes the function that maps x into a and all other arguments y in the domain of f into $f\, y$. The notation $f\rceil S$ denotes the restriction of the function f to the domain S.)

The allocation operation activates and initializes n cells in the heap. 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 all cause memory faults (denoted by the terminal configuration $\mathbf{abort}$) if an inactive address is dereferenced or deallocated.

An important property of this language is the effect of restricting the heap on the execution of a command. Essentially, if the restriction removes an address that is dereferenced or deallocated by the command, then the restricted execution aborts; otherwise, however, the executions are similar, except for the presence of unchanging extra heap cells in the unrestricted execution.

To state this property precisely, we write $h_0 \perp h_1$ to indicate that the heaps h_0 and h_1 have disjoint domains, and $h_0 \cdot h_1$ to indicate the union of such heaps. Then, when $h_0 \subseteq h$,

- If $\langle c, (s, h)\rangle \rightsquigarrow^* \textbf{abort}$, then $\langle c, (s, h_0)\rangle \rightsquigarrow^* \textbf{abort}$.
- If $\langle c, (s, h)\rangle \rightsquigarrow^* (s', h')$ then $\langle c, (s, h_0)\rangle \rightsquigarrow^* \textbf{abort}$ or $\langle c, (s, h_0)\rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$.
- If $\langle c, (s, h)\rangle \uparrow$ then either $\langle c, (s, h_0)\rangle \rightsquigarrow^* \textbf{abort}$ or $\langle c, (s, h_0)\rangle \uparrow$.

3. Assertions and their Inference Rules

In addition to the usual formulae of predicate calculus, including boolean expressions and quantifiers, we introduce four new forms of assertion that describe the heap:

$\langle\text{assert}\rangle ::= \cdots$	
$\mid \textbf{emp}$	empty heap
$\mid \langle\text{exp}\rangle \mapsto \langle\text{exp}\rangle$	singleton heap
$\mid \langle\text{assert}\rangle * \langle\text{assert}\rangle$	separating conjunction
$\mid \langle\text{assert}\rangle \mathrel{-\!\!*} \langle\text{assert}\rangle$	separating implication

Because of these new forms, the meaning of an assertion (unlike that of a boolean expression) depends upon both the store and the heap:

$$\llbracket p \in \langle\text{assert}\rangle\rrbracket_{\text{asrt}} \in \\ \Big(\bigcup^{\text{fin}}_{V \supseteq \mathrm{FV}(p)} \mathrm{Stores}_V\Big) \to \mathrm{Heaps} \to \{\textbf{true}, \textbf{false}\}.$$

Specifically, $\textbf{emp}$ asserts that the heap is empty:

$$\llbracket \textbf{emp}\rrbracket_{\text{asrt}} s\, h \ \text{ iff } \ \mathrm{dom}\, h = \{\},$$

$e \mapsto e'$ asserts that the heap contains one cell, at address e with contents e':

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

$p_0 * p_1$ asserts that the heap can be split into two disjoint parts in which p_0 and p_1 hold respectively:

$$\llbracket p_0 * p_1\rrbracket_{\text{asrt}} s\, h \ \text{ iff} \\ \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\ \llbracket p_0\rrbracket_{\text{asrt}} s\, h_0 \text{ and } \llbracket p_1\rrbracket_{\text{asrt}} s\, h_1,$$

and $p_0 \mathrel{-\!\!*} p_1$ asserts that, if the current heap is extended with a disjoint part in which p_0 holds, then p_1 will hold in the extended heap:

$$\llbracket p_0 \mathrel{-\!\!*} p_1\rrbracket_{\text{asrt}} s\, h \ \text{ iff} \\ \forall h'.\ (h' \perp h \text{ and } \llbracket p_0\rrbracket_{\text{asrt}} s\, h') \text{ implies} \\ \llbracket p_1\rrbracket_{\text{asrt}} s\, (h \cdot h').$$

When this semantics is coupled with the usual interpretation of the classical connectives, the result is an instance of the "resource semantics" of bunched implications as advanced by David Pym [23, 26].

It is useful to introduce several more complex forms as abbreviations:

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

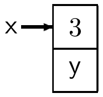
$$e \hookrightarrow e' \overset{\text{def}}{=} e \mapsto e' * \textbf{true}$$

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

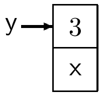
$$e \hookrightarrow e_1, \ldots, e_n \\ \overset{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e + n - 1 \hookrightarrow e_n \\ \text{iff } e \mapsto e_1, \ldots, e_n * \textbf{true}.$$

By using $\mapsto$, $\hookrightarrow$, and the two forms of conjunction, it is easy to describe simple sharing patterns concisely:

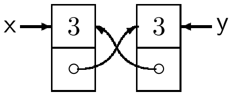
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β, α is a address, and the heap maps α into 3 and $\alpha + 1$ into β):



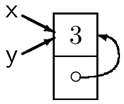
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x:



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.

There are also simple examples that reveal the occasionally surprising behavior of separating conjunction. Suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct addresses, so that

$$h_1 = \{\langle s\,\mathsf{x}, 1\rangle\} \qquad \text{and} \qquad h_2 = \{\langle s\,\mathsf{y}, 2\rangle\}$$

are heaps with disjoint singleton domains. Then

If p is:	then $[\![p]\!]_{\mathrm{asrt}} s\, h$ is:
$\mathsf{x} \mapsto 1$	$h = h_1$
$\mathsf{y} \mapsto 2$	$h = h_2$
$\mathsf{x} \mapsto 1 * \mathsf{y} \mapsto 2$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 * \mathsf{x} \mapsto 1$	**false**
$\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2$	$h = h_1$ or $h = h_2$
$\mathsf{x} \mapsto 1 * (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$(\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$ $* (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 * \mathsf{y} \mapsto 2$ $* (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	**false**
$\mathsf{x} \mapsto 1 * \mathbf{true}$	$h_1 \subseteq h$
$\mathsf{x} \mapsto 1 * \neg\, \mathsf{x} \mapsto 1$	$h_1 \subseteq h.$

To illustrate separating implication, suppose that an assertion p holds for a store that maps the variable x into an address α and a heap h that maps α into 16. Then

$$(\mathsf{x} \mapsto 16) \mathrel{-\!\!*} p$$

holds for the same store and the heap $h \rceil (\mathrm{dom}\, h - \{\alpha\})$ obtained by removing α from the domain of h, and

$$\mathsf{x} \mapsto 9 * ((\mathsf{x} \mapsto 16) \mathrel{-\!\!*} p)$$

holds for the same store and the heap $[\, h \mid \alpha{:}\, 9\,]$ that differs from h by mapping α into 9. (Thus, anticipating the concept of partial-correctness specification introduced in the next section, $\{\mathsf{x} \mapsto 9 * ((\mathsf{x} \mapsto 16) \mathrel{-\!\!*} p)\}\ [\mathsf{x}] := 16\ \{p\}$.)

The inference rules for predicate calculus remain sound in this enriched setting. Before presenting sound rules for the new forms of assertions, however, we note two rules that fail. Taking p to be $\mathsf{x} \mapsto 1$ and q to be $\mathsf{y} \mapsto 2$, one can see that neither contraction, $p \Rightarrow p * p$, nor weakening, $p * q \Rightarrow p$, are sound. Thus separation logic is a substructural logic. (It is not, however, a species of linear logic, since in linear logic $P \Rightarrow Q$ can be written $(!P) \mathrel{-\!\!*} Q$, so the rule of dereliction gives the validity of $(P \mathrel{-\!\!*} Q) \Rightarrow (P \Rightarrow Q)$. But in a state where $\mathsf{x} \mapsto 1$ is true, $(\mathsf{x} \mapsto 1) \mathrel{-\!\!*} \mathbf{false}$ is true but $(\mathsf{x} \mapsto 1) \Rightarrow \mathbf{false}$ is false [19].)

Sound axiom schemata for separating conjunction include commutative and associative laws, the fact that $\mathbf{emp}$ is a neutral element, and various distributive and semidistributive laws:

$$p_1 * p_2 \Leftrightarrow p_2 * p_1$$

$$(p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3)$$

$$p * \mathbf{emp} \Leftrightarrow p$$

$$(p_1 \vee p_2) * q \Leftrightarrow (p_1 * q) \vee (p_2 * q)$$

$$(p_1 \wedge p_2) * q \Rightarrow (p_1 * q) \wedge (p_2 * q)$$

$$(\exists x.\ p) * q \Leftrightarrow \exists x.\ (p * q) \quad \text{when } x \text{ not free in } q$$

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

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

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2,}$$

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

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

There are several semantically defined classes of assertions that have useful special properties, and contain an easily defined syntactic subclass.

An assertion is said to be *pure* if, for any store, it is independent of the heap. Syntactically, an assertion is pure if it does not contain $\mathbf{emp}$, $\mapsto$, or $\hookrightarrow$. The following axiom schemata show that, when assertions are pure, the distinction between the two kinds of conjunctions, or of implications, collapses:

$$p_1 \wedge p_2 \Rightarrow p_1 * p_2 \qquad \text{when } p_1 \text{ or } p_2 \text{ is pure}$$

$$p_1 * p_2 \Rightarrow p_1 \wedge p_2 \qquad \text{when } p_1 \text{ and } p_2 \text{ are pure}$$

$$(p \wedge q) * r \Leftrightarrow p \wedge (q * r) \qquad \text{when } p \text{ is pure}$$

$$(p_1 \mathrel{-\!\!*} p_2) \Rightarrow (p_1 \Rightarrow p_2) \qquad \text{when } p_1 \text{ is pure}$$

$$(p_1 \Rightarrow p_2) \Rightarrow (p_1 \mathrel{-\!\!*} p_2) \qquad \text{when } p_1 \text{ and } p_2 \text{ are pure.}$$

We say that an assertion p is *intuitionistic* iff, for all stores s and heaps h and h':

$$h \subseteq h' \text{ and } [\![p]\!]_{\mathrm{asrt}}(s, h) \text{ implies } [\![p]\!]_{\mathrm{asrt}}(s, h').$$

1043-6871/02 $17.00 © 2002 **IEEE**

One can show that $p \ast \mathbf{true}$ is the strongest intuitionistic assertion weaker than p, and that $\mathbf{true} \mathrel{-\!\!\ast} p$ is the weakest intuitionistic assertion stronger than p. Syntactically, If p and q are intuitionistic assertions, r is any assertion, and e and e' are expressions, then the following are intuitionistic assertions:

$$\begin{array}{cc}
\text{Any pure assertion} & e \hookrightarrow e' \\
r \ast \mathbf{true} & \mathbf{true} \mathrel{-\!\!\ast} r \\
p \wedge q & p \vee q \\
\forall v.\ p & \exists v.\ p \\
p \ast q & p \mathrel{-\!\!\ast} q.
\end{array}$$

For intuitionistic assertions, we have

$$\begin{array}{rl}
(p \ast \mathbf{true}) \Rightarrow p & \text{when } p \text{ is intuitionistic} \\
p \Rightarrow (\mathbf{true} \mathrel{-\!\!\ast} p) & \text{when } p \text{ is intuitionistic.}
\end{array}$$

It should also be noted that, if we define the operations

$$\begin{aligned}
\overset{\mathrm{i}}{\neg}\, p &\overset{\mathrm{def}}{=} \mathbf{true} \mathrel{-\!\!\ast} (\neg\, p) \\
p \overset{\mathrm{i}}{\Rightarrow} q &\overset{\mathrm{def}}{=} \mathbf{true} \mathrel{-\!\!\ast} (p \Rightarrow q) \\
p \overset{\mathrm{i}}{\Leftrightarrow} q &\overset{\mathrm{def}}{=} \mathbf{true} \mathrel{-\!\!\ast} (p \Leftrightarrow q),
\end{aligned}$$

then the assertions built from pure assertions and $e \hookrightarrow e'$, using these operations and $\wedge$, $\vee$, $\forall$, $\exists$, $\ast$, and $\mathrel{-\!\!\ast}$ form the image of Ishtiaq and O'Hearn's modal translation from intuitionistic separation logic to the classical version [19].

Yang [33, 34] has singled out the class of *strictly exact* assertions; an assertion q is strictly exact iff, for all s, h, and h', $[\![q]\!]_{\mathrm{asrt}} sh$ and $[\![q]\!]_{\mathrm{asrt}} sh'$ implies $h = h'$. Syntactically, assertions built from expressions using $\mapsto$ and $\ast$ are strictly exact. The utility of this concept is that

$$(q \ast \mathbf{true}) \wedge p \Rightarrow q \ast (q \mathrel{-\!\!\ast} p) \quad \text{when } q \text{ is strictly exact.}$$

Strictly exact assertions also belong to the broader class of *domain-exact* assertions; an assertion q is domain-exact iff, for all s, h, and h', $[\![q]\!]_{\mathrm{asrt}} sh$ and $[\![q]\!]_{\mathrm{asrt}} sh'$ implies $\operatorname{dom} h = \operatorname{dom} h'$. Syntactically, assertions built from expressions using $\mapsto$, $\ast$, and quantifiers are domain-exact. When q is such an assertion, the semidistributive laws given earlier become full distributive laws:

$$(p_1 \ast q) \wedge (p_2 \ast q) \Rightarrow (p_1 \wedge p_2) \ast q$$
$$\text{when } q \text{ is domain-exact}$$

$$\forall x.\ (p \ast q) \Rightarrow (\forall x.\ p) \ast q$$
$$\text{when } x \text{ not free in } q \text{ and } q \text{ is domain-exact.}$$

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

$$\begin{gathered}
e_1 \mapsto e_1' \wedge e_2 \mapsto e_2' \Leftrightarrow e_1 \mapsto e_1' \wedge e_1 = e_2 \wedge e_1' = e_2' \\
e_1 \hookrightarrow e_1' \ast e_2 \hookrightarrow e_2' \Rightarrow e_1 \neq e_2 \\
\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -).
\end{gathered}$$

Both the assertion language developed in this section and the programming language developed in the previous section are limited by the fact that all variables range over the integers. Later in this paper we will go beyond this limitation in ways that are sufficiently obvious that we will not formalize their semantics, e.g., we will use variables denoting abstract data types, predicates, and assertions in the assertion language, and variables denoting procedures in the programming language. (We will not, however, consider assignment to such variables.)

4. Specifications and their Inference Rules

We use a notion of program specification that is similar to that of Hoare logic, with variants for both partial and total correctness:

$$\begin{array}{lr}
\langle\text{spec}\rangle ::= \{\langle\text{assert}\rangle\}\ \langle\text{comm}\rangle\ \{\langle\text{assert}\rangle\} & \text{partial} \\
\quad\mid [\ \langle\text{assert}\rangle\ ]\ \langle\text{comm}\rangle\ [\ \langle\text{assert}\rangle\ ] & \text{total}
\end{array}$$

Let $V = \mathrm{FV}(p) \cup \mathrm{FV}(c) \cup \mathrm{FV}(q)$. Then

$$\begin{aligned}
&\{p\}\ c\ \{q\} \text{ holds iff } \forall (s,h) \in \mathrm{States}_V.\ [\![p]\!]_{\mathrm{asrt}} s\, h \text{ implies} \\
&\qquad \neg\, (c, (s,h) \rightsquigarrow^* \mathbf{abort}) \\
&\qquad \text{and } (\forall (s',h') \in \mathrm{States}_V. \\
&\qquad\qquad c, (s,h) \rightsquigarrow^* (s',h') \text{ implies } [\![q]\!]_{\mathrm{asrt}} s'\, h'),
\end{aligned}$$

and

$$\begin{aligned}
&[\,p\,]\ c\ [\,q\,] \text{ holds iff } \forall (s,h) \in \mathrm{States}_V.\ [\![p]\!]_{\mathrm{asrt}} s\, h \text{ implies} \\
&\qquad \neg\, (c, (s,h) \rightsquigarrow^* \mathbf{abort}) \\
&\qquad \text{and } \neg\, (c, (s,h) \uparrow) \\
&\qquad \text{and } (\forall (s',h') \in \mathrm{States}_V. \\
&\qquad\qquad c, (s,h) \rightsquigarrow^* (s',h') \text{ implies } [\![q]\!]_{\mathrm{asrt}} s'\, h').
\end{aligned}$$

Notice that specifications are implicitly quantified over both stores and heaps, and also (since allocation is indeterminate) over all possible executions. Moreover, any execution giving a memory fault falsifies both partial and total specifications.

As O'Hearn [19] paraphrased Milner, "Well-specified programs don't go wrong." As a consequence, during the execution of programs that have been proved to meet their specifications, it is unnecessary to check for memory faults, or even to equip heap cells with activity bits (assuming that programs are only executed in initial states satisfying the relevant precondition).

Roughly speaking, the fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. But sometimes the notion of record boundaries dissolves, as in the following valid specification of a program that tries to form a two-field record by gluing together two one-field records:

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

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

- Consequence

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

- Auxiliary Variable Elimination

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

  where v is not free in c.

- Substitution

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \dots\ , v_n \to e_n,}$$

  where $v_1, \dots, v_n$ 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]:

$$\frac{\{p\}\ c\ \{q\}}{\{p \wedge r\}\ c\ \{q \wedge r\},}$$

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.

Unfortunately, however, the rule of constancy is not sound for 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 aliasing that will occur if $\mathsf{x} = \mathsf{y}$.

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

$$\frac{\{p\}\ c\ \{q\}}{\{p \ast r\}\ c\ \{q \ast r\},}$$

  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 *parts of the heap* that are actually used by c (which O'Hearn calls the *footprint* of c), by adding arbitrary predicates about variables and parts of the heap 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].

Every valid specification $\{p\}\ c\ \{q\}$ is "tight" in the sense that every cell in its footprint must either be allocated by c or asserted to be active by p; "locality" is the opposite property that everything asserted to be active 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 larger footprint of an enclosing command.

To see the soundness of the frame rule [35, 34], assume $\{p\}\ c\ \{q\}$, and $[\![p \ast r]\!]_{\mathrm{asrt}} s\, h$. Then there are h_0 and h_1 such that $h_0 \perp h_1$, $h = h_0 \cdot h_1$, $[\![p]\!] s\, h_0$ and $[\![r]\!] s\, h_1$.

- Suppose $\langle c, (s, h)\rangle \leadsto^* \mathbf{abort}$. Then, by by the property described at the end of Section 2, we would have $\langle c, (s, h_0)\rangle \leadsto^* \mathbf{abort}$, which contradicts $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$.

- Suppose $\langle c, (s, h)\rangle \leadsto^* (s', h')$. As in the previous case, $\langle c, (s, h_0)\rangle \leadsto^* \mathbf{abort}$ would contradict $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$, so that, by the property at the end of Section 2, $\langle c, (s, h_0)\rangle \leadsto^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$. Then $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$ implies that $[\![q]\!] s' h'_0$. Moreover, since $\langle c, (s, h)\rangle \leadsto^* (s', h')$, the stores s and s' will give the same value to the variables that are not modified by c. Then, since these include all the free variables of r, $[\![r]\!]s\,h_1$ implies that $[\![r]\!]s'h_1$. Thus $[\![q\ *\ r]\!]s'h'$.

Yang [35, 34] has also shown that the frame rule is complete in the following sense: Suppose that all we know about a command c is that some partial correctness specification $\{p\}\ c\ \{q\}$ is valid. Then, whenever the validity of $\{p'\}\ c\ \{q'\}$ is a semantic consequence of this knowledge, $\{p'\}\ c\ \{q'\}$ can be derived from $\{p\}\ c\ \{q\}$ by use of the frame rule and the rules of consequence, auxiliary variable elimination, and substitution.

Using the frame rule, one can move from local versions of inference rules for the primitive heap-manipulating commands to equivalent global versions. For mutation, for example, the obvious local rule

- Mutation (local)

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

leads directly to

- Mutation (global)

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

(One can rederive the local rule from the global one by taking r to be $\mathbf{emp}$.) Moreover, by taking r in the global rule to be $(e \mapsto e') \mathrel{-\!\!*} p$ and using the valid implication $q\ *\ (q \mathrel{-\!\!*} p) \Rightarrow p$, one can derive a third rule for mutation that is suitable for backward reasoning [19], since one can substitute any assertion for the postcondition p:

- Mutation (backwards reasoning)

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

(One can rederive the global rule from the backward one by taking p to be $(e \mapsto e')\ *\ r$ and using the valid implication $(p\ *\ r) \Rightarrow (p\ *\ (q \mathrel{-\!\!*} (q\ *\ r)))$.)

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 "noninterfering" allocation commands that modify "fresh" variables. Here we abbreviate $e_1, \ldots, e_n$ by $\overline{e}$.

- Allocation (noninterfering, local)

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

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

- Allocation (noninterfering, global)

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

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

However, to avoid the restrictions on occurrences of the assigned variable, or to give a backward-reasoning rule, or rules for lookup, we must introduce and often quantify additional variables. For both allocation and lookup, we can give equivalent rules of the three kinds, but the relevant derivations are more complicated than before since one must use auxiliary variable elimination, properties of quantifiers, and other laws.

In these rules we indicate substitution by priming metavariables denoting expressions and assertions, e.g., we write e_i' for $e_i/v \to v'$ and r' for $r/v \to v'$. We also abbreviate $e_1, \ldots, e_n$ by $\overline{e}$ and $e_1', \ldots, e_n'$ by $\overline{e}'$.

- Allocation (local)

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

where v' is distinct from v.

- Allocation (global)

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

where v' is distinct from v, and is not free in $\overline{e}$ or r.

- Allocation (backwards reasoning)

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

where v' is distinct from v, and is not free in $\overline{e}$ or p.

- Lookup (local)

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

where v, v', and v'' are distinct.

- Lookup (global)

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

where v, v', and v'' are distinct, v' and v'' do not occur free in e, and v is not free in r.

- Lookup (backwards reasoning)

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

where v' is not free in e, nor free in p unless it is v.

Finally, since $e \mapsto v'$ is strictly exact, it is easy to obtain an equivalent but more succinct rule for backward reasoning about lookup:

- Lookup (alternative backward reasoning)

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

where v' is not free in e, nor free in p unless it is v.

For all four new commands, the backward reasoning forms give complete weakest preconditions [19, 34].

As a simple illustration, the following is a detailed proof outline of a local specification of a command that uses allocation and mutation to construct a two-element cyclic structure containing relative addresses:

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

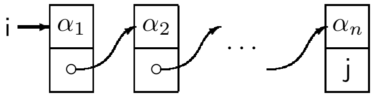
5. Lists

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. To do so, it is necessary to define the set of abstract values algebraically or recursively, and to define predicates on the abstract values by structural induction. Since these are standard methods of definition, we will treat them less formally than the novel aspects of our logic.

For lists, the relevant abstract values are sequences, for which we use the following notation: When α and β are sequences, we write

- ϵ for the empty sequence.
- $[x]$ for the single-element sequence containing x. (We will omit the brackets when x 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.

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



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

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

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

and to prove some basic properties:

$$\mathsf{list}\ \mathsf{a}\ (\mathsf{i}, \mathsf{j}) \Leftrightarrow \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}$$

$$\mathsf{list}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i}, \mathsf{j}) * \mathsf{list}\ \beta\ (\mathsf{j}, \mathsf{k})$$

$$\mathsf{list}\ \alpha{\cdot}\mathsf{b}\ (\mathsf{i}, \mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i}, \mathsf{j}) * \mathsf{j} \mapsto \mathsf{b}, \mathsf{k}.$$

(The second property is a composition law that can be proved by structural induction on α.)

In comparison with the definition of list in the introduction, we have generalized from lists to list segments, and we have used separating conjunction to prohibit cycles *within* the list segment. More precisely, when $\mathsf{list}\ \alpha_1 \cdot \cdots \cdot \alpha_n\ (\mathsf{i}_0, \mathsf{i}_n)$, we have

$$\exists \mathsf{i}_1, \ldots \mathsf{i}_{n-1}.$$
$$(\mathsf{i}_0 \mapsto \alpha_1, \mathsf{i}_1) * (\mathsf{i}_1 \mapsto \alpha_2, \mathsf{i}_2) * \cdots * (\mathsf{i}_{n-1} \mapsto \alpha_n, \mathsf{i}_n).$$

Thus $\mathsf{i}_0, \ldots, \mathsf{i}_{n-1}$ are distinct, but i_n is not constrained, so that $\mathsf{list}\ \alpha_1 \cdot \cdots \cdot \alpha_n\ (\mathsf{i}, \mathsf{i})$ may hold for any $n \geq 0$.

Thus the obvious properties

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

$$\mathsf{list}\ \alpha\ (\mathsf{i}, \mathsf{j}) \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon)$$

do not say whether α is empty when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$. However, there are some common situations that insure that α is empty when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$, for example, when j refers to a heap cell separate from the list segment, or to the beginning of a separate list:

$$\mathsf{list}\,\alpha\,(\mathsf{i},\mathsf{j}) \;*\; \mathsf{j} \hookrightarrow - \Rightarrow (\mathsf{i}=\mathsf{j} \Leftrightarrow \alpha=\epsilon)$$
$$\mathsf{list}\,\alpha\,(\mathsf{i},\mathsf{j}) \;*\; \mathsf{list}\,\beta\,(\mathsf{j},\mathbf{nil}) \Rightarrow (\mathsf{i}=\mathsf{j} \Leftrightarrow \alpha=\epsilon).$$

On the other hand, sometimes $\mathsf{i}=\mathsf{j}$ simply does not determine emptiness. For example, a cyclic buffer containing α in its active segment and β in its inactive segment is described by $\mathsf{list}\,\alpha\,(\mathsf{i},\mathsf{j}) \;*\; \mathsf{list}\,\beta\,(\mathsf{j},\mathsf{i})$. Here, the buffer may be either empty or full when $\mathsf{i}=\mathsf{j}$.

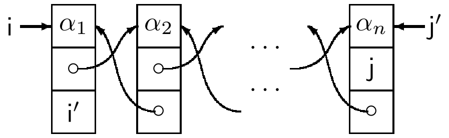
The use of list is illustrated by a proof outline for a command that deletes the first element of a list:

$$\{\mathbf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}$$
$$\{\exists \mathsf{j}.\ \mathsf{i}\mapsto \mathsf{a},\mathsf{j} \;*\; \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\{\mathsf{i}\mapsto \mathsf{a} \;*\; \exists \mathsf{j}.\ \mathsf{i}+1\mapsto \mathsf{j} \;*\; \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathsf{j} := [\mathsf{i}+1]\ ;$$
$$\{\mathsf{i}\mapsto \mathsf{a} \;*\; \mathsf{i}+1\mapsto \mathsf{j} \;*\; \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathbf{dispose}\ \mathsf{i}\ ;$$
$$\{\mathsf{i}+1\mapsto \mathsf{j} \;*\; \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathbf{dispose}\ \mathsf{i}+1\ ;$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathsf{i} := \mathsf{j}$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{k})\}$$

A more complex example is the body of the while command in the list-reversing program in the Introduction; here the final assertion is the invariant of the while command:

$$\{\exists \alpha,\beta.\ (\mathbf{list}\ \alpha\,(\mathsf{i},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta \wedge \mathsf{i} \neq \mathbf{nil}\}$$
$$\{\exists \mathsf{a},\alpha,\beta.\ (\mathbf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\{\exists \mathsf{a},\alpha,\beta,\mathsf{k}.\ (\mathsf{i}\mapsto \mathsf{a},\mathsf{k} \;*\; \mathbf{list}\ \alpha\,(\mathsf{k},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\mathsf{k} := [\mathsf{i}+1]\ ;$$
$$\{\exists \mathsf{a},\alpha,\beta.\ (\mathsf{i}\mapsto \mathsf{a},\mathsf{k} \;*\; \mathbf{list}\ \alpha\,(\mathsf{k},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$[\mathsf{i}+1] := \mathsf{j}\ ;$$
$$\{\exists \mathsf{a},\alpha,\beta.\ (\mathsf{i}\mapsto \mathsf{a},\mathsf{j} \;*\; \mathbf{list}\ \alpha\,(\mathsf{k},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\{\exists \mathsf{a},\alpha,\beta.\ (\mathbf{list}\ \alpha\,(\mathsf{k},\mathbf{nil}) \;*\; \mathbf{list}\ \mathsf{a}{\cdot}\beta\,(\mathsf{i},\mathbf{nil}))$$
$$\quad\wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\mathsf{a}{\cdot}\beta\}$$
$$\{\exists \alpha,\beta.\ (\mathbf{list}\ \alpha\,(\mathsf{k},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{i},\mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$$
$$\mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}$$
$$\{\exists \alpha,\beta.\ (\mathbf{list}\ \alpha\,(\mathsf{i},\mathbf{nil}) \;*\; \mathbf{list}\ \beta\,(\mathsf{j},\mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}.$$

A more elaborate representation of sequences is provided by *doubly-linked* lists. Here, we write $\mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}')$ when α is represented by a doubly-linked list segment with a forward linkage (via second fields) from i to j, and a backward linkage (via third fields) from j' to i':



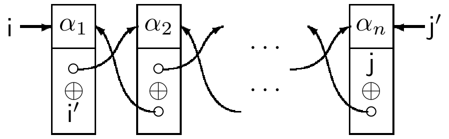
The inductive definition is

$$\mathsf{dlist}\,\epsilon\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \overset{\mathrm{def}}{=} \mathbf{emp} \wedge \mathsf{i}=\mathsf{j} \wedge \mathsf{i}'=\mathsf{j}'$$
$$\mathsf{dlist}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \overset{\mathrm{def}}{=} \exists \mathsf{j}.\ \mathsf{i}\mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \;*\; \mathsf{dlist}\,\alpha\,(\mathsf{j},\mathsf{i},\mathsf{k},\mathsf{k}'),$$

from which one can prove the basic properties:

$$\mathsf{dlist}\,\mathsf{a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \Leftrightarrow \mathsf{i}\mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \wedge \mathsf{i}=\mathsf{j}'$$
$$\mathsf{dlist}\,\alpha{\cdot}\beta\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \Leftrightarrow$$
$$\qquad \exists \mathsf{j},\mathsf{j}'.\ \mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \;*\; \mathsf{dlist}\,\beta\,(\mathsf{j},\mathsf{j}',\mathsf{k},\mathsf{k}')$$
$$\mathsf{dlist}\,\alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \Leftrightarrow$$
$$\qquad \exists \mathsf{j}'.\ \mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k}',\mathsf{j}') \;*\; \mathsf{k}'\mapsto \mathsf{b},\mathsf{k},\mathsf{j}'.$$

The utility of unrestricted address arithmetic is illustrated by a variation of the doubly-linked list, in which the second and third fields of each record are replaced by a single field giving the exclusive **or** of their contents. If we write $\mathsf{xlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}')$ when α is represented by such an *xor-linked* list:

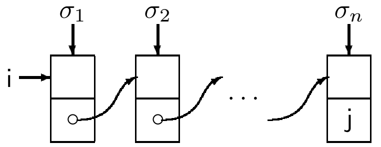


we can define this predicate by

$$\mathsf{xlist}\,\epsilon\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \overset{\mathrm{def}}{=} \mathbf{emp} \wedge \mathsf{i}=\mathsf{j} \wedge \mathsf{i}'=\mathsf{j}'$$
$$\mathsf{xlist}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \overset{\mathrm{def}}{=}$$
$$\qquad \exists \mathsf{j}.\ \mathsf{i}\mapsto \mathsf{a},(\mathsf{j}\oplus\mathsf{i}') \;*\; \mathsf{xlist}\,\alpha\,(\mathsf{j},\mathsf{i},\mathsf{k},\mathsf{k}').$$

The basic properties are analogous to those for dlist [24].

Finally, we mention an idea of Richard Bornat's [1], that instead of denoting a sequence of data items, a list (or other structure) should denote the sequence (or other collection) of addresses at which the data items are stored. In the case of simply linked lists, we write $\mathsf{listN}\,\sigma\,(\mathsf{i},\mathsf{j})$ when there is a list segment from i to j containing the sequence σ of addresses:



This view leads to the definition

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

$$\mathsf{listN}\;\mathsf{l}{\cdot}\sigma\,(\mathsf{i},\mathsf{k}) \stackrel{\mathrm{def}}{=} \mathsf{l} = \mathsf{i} \wedge \exists \mathsf{j}.\; \mathsf{i} + 1 \mapsto \mathsf{j} \,*\, \mathsf{listN}\;\sigma\,(\mathsf{j},\mathsf{k}).$$

Notice that listN is extremely local: It holds for a heap containing only the second cell of each record in the list structure.

The reader may verify that the body of the list-reversing program preserves the invariant

$$\exists \sigma, \delta.\; (\mathbf{listN}\;\sigma\,(\mathsf{i},\mathbf{nil}) \,*\, \mathbf{listN}\;\delta\,(\mathsf{j},\mathbf{nil})) \wedge \sigma_0^{\dagger} = \sigma^{\dagger}{\cdot}\delta,$$

where σ_0 is the original sequence of addresses of the data fields of the list at i. In fact, since it shows that these addresses do not change, this invariant embodies a stronger notion of "in-place algorithm" than that given before.

6. Trees and Dags

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

In this section, 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

$$\begin{aligned} &\tau \in \text{S-exps iff} \\ &\qquad\qquad \tau \in \text{Atoms} \\ &\qquad\quad \text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps}. \end{aligned}$$

(Of course, this is just a particular, and very simple, initial algebra. We could take carriers of any anarchic 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 loops "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:

$$\mathsf{tree}\;a\,(i) \text{ iff } \mathbf{emp} \wedge i = a$$

$$\begin{aligned} &\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff} \\ &\qquad \exists i_1, i_2.\; i \mapsto i_1, i_2 \,*\, \mathsf{tree}\;\tau_1\,(i_1) \,*\, \mathsf{tree}\;\tau_2\,(i_2) \end{aligned}$$

$$\mathsf{dag}\;a\,(i) \text{ iff } i = a$$

$$\begin{aligned} &\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff} \\ &\qquad \exists i_1, i_2.\; i \mapsto i_1, i_2 \,*\, (\mathsf{dag}\;\tau_1\,(i_1) \wedge \mathsf{dag}\;\tau_2\,(i_2)). \end{aligned}$$

Here, since $\mathbf{emp}$ is omitted from its definition, $\mathsf{dag}\;a\,(i)$ is pure, and therefore intuitionistic. By induction, it is easily seen that $\mathsf{dag}\;\tau\, i$ is intuitionistic for all τ. In fact, this is vital, since we want $\mathsf{dag}\;\tau_1\,(i_1) \wedge \mathsf{dag}\;\tau_2\,(i_2)$ to hold for a heap that contains the (possibly overlapping) sub-dags, but not to assert that the sub-dags are identical.

To express simple algorithms for manipulating trees, we must introduce recursive procedures. However, to avoid problems of aliased variables and interfering procedures, we will limit ourselves to first-order procedures without global variables (except, in the case of recursion, the name of procedure being defined), and we will explicitly indicate which formal parameters are modified by the procedure body. Thus a procedure definition will have the form

$$h(x_1, \cdots, x_m; y_1, \cdots, y_n) = c,$$

where $y_1, \cdots, y_n$ are the free variables modified by c, and $x_1, \cdots, x_m$ are the other free variables of c, except h.

We will not declare procedures at the beginning of blocks, but will simply assume that a program is reasoned about in the presence of procedure definitions. In this setting, an appropriate inference rule for partial correctness is

- Procedures

  When $h(x_1, \cdots, x_m; y_1, \cdots, y_n) = c$,

$$\frac{\begin{array}{c} \{p\}\; h(x_1, \cdots, x_m; y_1, \cdots, y_n)\; \{q\} \\ \vdots \\ \{p\}\; c\; \{q\} \end{array}}{\{p\}\; h(x_1, \cdots, x_m; y_1, \cdots, y_n)\; \{q\}.}$$

In essence, to prove some specification of a call of a procedure in which the actual parameters are the same as the formal parameters in the procedure definition, one proves a similar specification of the body of the definition under the *recursion hypothesis* that recursive calls satisfy the specification being proved.

Of course, one must be able to deduce specifications about calls in which the actual parameters differ from the formals. For this purpose, however, the structural inference rule for substitution suffices, if one takes the variables modified by $h(x_1, \cdots, x_m; y_1, \cdots, y_n)$ to be $y_1, \cdots, y_n$. Note that the restrictions on this rule prevent the creation of aliased variables or expressions.

For example, we would expect a tree-copying procedure

$$\begin{array}{l}
\mathsf{copytree}(\mathsf{i};\mathsf{j}) = \\
\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\
\quad\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in} \\
\quad\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\
\quad\quad\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;\ \mathbf{copytree}(\mathsf{i}_2;\mathsf{j}_2)\ ; \\
\quad\quad\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1,\mathsf{j}_2)\big)
\end{array}$$

to satisfy

$$\{\mathsf{tree}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{tree}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}. \tag{5}$$

The following is a proof outline of a similar specification of the procedure body:

$$\begin{array}{l}
\{\mathsf{tree}\ \tau(\mathsf{i})\} \\
\mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then} \\
\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\} \\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{i} = \tau))\} \\
\quad \mathsf{j} := \mathsf{i} \\
\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{j} = \tau))\} \\
\mathbf{else} \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{tree}\ (\tau_1 \cdot \tau_2)(\mathsf{i})\} \\
\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in} \\
\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ;\ \mathsf{i}_2 := [\mathsf{i}+1]\ ; \\
\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 \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2))\} \\
\quad\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\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 \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1))\} \\
\quad\quad \mathbf{copytree}(\mathsf{i}_2;\mathsf{j}_2)\ ; \\
\quad\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\quad\quad\quad (\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 \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2))\} \\
\quad\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2) \\
\quad\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\quad\quad\quad (\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 \mathsf{j} \mapsto \mathsf{j}_1, \mathsf{j}_2\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2))\} \\
\quad\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\quad\quad\quad (\mathsf{tree}\ (\tau_1 {\cdot} \tau_2)\ (\mathsf{i})\ *\ \mathsf{tree}\ (\tau_1 {\cdot} \tau_2)\ (\mathsf{j}))\}\big) \\
\{\mathsf{tree}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}.
\end{array}$$

To obtain the specifications of the recursive calls in this proof outline from the recursion hypothesis (5), one must use the frame rule to move from the footprint of the recursive call to the larger footprint of the procedure body. In more detail, the specification of the first recursive call in the outline can be obtained by the inferences

$$\begin{array}{l}
\{\mathsf{tree}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{tree}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\} \\
\hline
\{\mathsf{tree}\ \tau_1(\mathsf{i}_1)\}\ \mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ \{\mathsf{tree}\ \tau_1(\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_1(\mathsf{j}_1)\} \\
\hline
\quad \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2)\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\} \\
\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\quad \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2)\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\} \\
\hline
\quad \{\tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2))\} \\
\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\quad \{\tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1))\} \\
\hline
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2))\} \\
\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\quad \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ * \\
\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1))\},
\end{array}$$

using the substitution rule, the frame rule, the rule of consequence (with the axiom that $(p \wedge q)\ *\ r \Leftrightarrow p \wedge (q\ *\ r)$ when p is pure), and auxiliary variable elimination.

What we have proved about copytree, however, is not as general as possible. In fact, this procedure is insensitive to sharing in its input, so that it also satisfies

$$\{\mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{dag}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}. \tag{6}$$

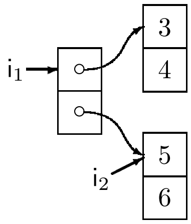
But if we try to prove this specification by mimicking the previous proof, we encounter a problem: For (say) the first recursive call, at the point where we used the frame rule, we must infer

$$\begin{array}{l}
\{(\cdots)\ *\ (\mathsf{dag}\ \tau_1(\mathsf{i}_1) \wedge \mathsf{dag}\ \tau_2(\mathsf{i}_2))\} \\
\mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1) \\
\{(\cdots)\ *\ (\mathsf{dag}\ \tau_1(\mathsf{i}_1) \wedge \mathsf{dag}\ \tau_2(\mathsf{i}_2))\ *\ \mathsf{tree}\ \tau_1(\mathsf{j}_1)\}.
\end{array}$$

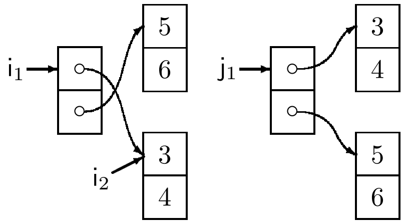
Now, however, the presence of $\wedge$ instead of $*$ prevents us from using the frame rule — and for good reason: The recursion hypothesis (6) is not strong enough to imply this specification of the recursive call, since it does not imply that $\mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)$ leaves unchanged the portion of the heap shared by the dags representing τ_1 and τ_2. For example, suppose

$$\tau_1 = ((3 \cdot 4) \cdot (5 \cdot 6)) \qquad\qquad \tau_2 = (5 \cdot 6).$$

Then (6) would permit $\mathsf{copytree}(\mathsf{i}_1;\mathsf{j}_1)$ to change the state from



into



where $\mathsf{dag}\ \tau_2\ (\mathsf{i}_2)$ is false.

One way of surmounting this problem is to extend assertions to contain *assertion variables*, and to extend the substitution rule so that the v_i's include assertion variables as well as ordinary variables, with assertions being substituted for these assertion variables.

Then we can use an assertion variable p to specify that every property of the heap that is active before executing $\mathsf{copytree}(\mathsf{i};\mathsf{j})$ remains true after execution:

$$\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p} * \mathsf{tree}\ \tau(\mathsf{j})\}. \qquad (7)$$

We leave the proof outline to the reader, but show the inference of the specification of the first recursive call from the recursion hypothesis, using the substitution rule and auxiliary variable elimination:

$$\begin{array}{l}
\qquad \{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p} * \mathsf{tree}\ \tau(\mathsf{j})\} \\
\hline
\quad \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)) \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)\} \\
\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\quad \{(\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)) * \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\} \\
\hline
\{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)) \wedge \mathsf{dag}\ \tau_1\ (\mathsf{i}_1)\} \\
\mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\{\exists \tau_1, \tau_2.\ \ (\tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{p} \wedge \mathsf{dag}\ \tau_2\ (\mathsf{i}_2)) * \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\}.
\end{array}$$

7. Arrays and 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

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

with the operational semantics

$$\begin{array}{l}
\langle v := \mathbf{allocate}\ e, (s, h)\rangle \rightsquigarrow ([\, s \mid v{:}\, \ell\,], h \cdot h') \\
\quad \text{where } h \perp h' \text{ and } \operatorname{dom} h' = \{\, i \mid \ell \leq i < \ell + [\![e]\!]_{\text{exp}} s \,\}.
\end{array}$$

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

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

with the meaning

$$\begin{array}{l}
[\![\bigodot_{v=e}^{e'} p]\!]_{\text{asrt}}(s, h) = \\
\qquad \mathbf{let}\ m = [\![e]\!]_{\text{exp}} s,\ \ n = [\![e']\!]_{\text{exp}} s, \\
\qquad\qquad\qquad I = \{\, i \mid m \leq i \leq n \,\}\ \mathbf{in} \\
\qquad\quad \exists H \in I \to \text{Heaps}. \\
\qquad\qquad\quad \forall i, j \in I.\ i \neq j \text{ implies } Hi \perp Hj \\
\qquad\qquad\quad \text{and } h = \bigcup\{\, Hi \mid i \in I \,\} \\
\qquad\qquad\quad \text{and } \forall i \in I.\ [\![p]\!]_{\text{asrt}}([\, s \mid v{:}\, i\,], Hi).
\end{array}$$

The following axiom schemata are useful:

$$m > n \Rightarrow \left(\bigodot_{i=m}^{n} p(i) \Leftrightarrow \mathbf{emp}\right)$$

$$m = n \Rightarrow \left(\bigodot_{i=m}^{n} p(i) \Leftrightarrow p(m)\right)$$

$$\begin{array}{l}
k \leq m \leq n + 1 \Rightarrow \\
\qquad \left(\bigodot_{i=k}^{n} p(i) \Leftrightarrow \left(\bigodot_{i=k}^{m-1} p(i) * \bigodot_{i=m}^{n} p(i)\right)\right)
\end{array}$$

$$\bigodot_{i=m}^{n} p(i) \Leftrightarrow \bigodot_{i=m-k}^{n-k} p(i + k)$$

$$\begin{array}{l}
m \leq n \Rightarrow \\
\qquad \left(\left(\bigodot_{i=m}^{n} p(i)\right) * q \Leftrightarrow \bigodot_{i=m}^{n} (p(i) * q)\right) \\
\qquad\qquad\qquad \text{when } q \text{ is pure}
\end{array}$$

$$m \leq j \leq n \Rightarrow \left(\left(\bigodot_{i=m}^{n} p(i)\right) \Rightarrow (p(j) * \mathbf{true})\right).$$

A simple example 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 $\mathsf{l} := \mathbf{allocate}\ \mathsf{n}$, 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

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

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

Somewhat surprisingly, iterated separating conjunction is useful for making assertions about structures that do not involve arrays. For example, the connection between our list and Bornat's listN is given by

$$\begin{aligned}&\mathsf{list}\,\alpha\,(\mathsf{i},\mathsf{j}) \Leftrightarrow \\ &\qquad \exists\sigma.\ \#\sigma = \#\alpha \wedge (\mathsf{listN}\,\sigma\,(\mathsf{i},\mathsf{j}) \ * \ \textstyle\bigodot_{\mathsf{k}=1}^{\#\alpha} \sigma_k \mapsto \alpha_k).\end{aligned}$$

A more elaborate example is provided by a program that accepts a list i representing a sequence α of integers, and produces a list j of lists representing all (not necessarily contiguous) subsequences of α. This program is a variant of a venerable LISP example that has often been used to illustrate the storage economy that can be obtained in LISP by a straightforward use of sharing. Our present interest is in using iterated separating conjunction to specify the sharing pattern in sufficient detail to show the amount of storage that is used.

First, given a sequence σ of sequences, we define $\mathsf{ext_a}\sigma$ to be the sequence of sequences obtained by prefixing the integer a to each sequence in σ:

$$\begin{aligned}\#\mathsf{ext_a}\sigma &\stackrel{\text{def}}{=} \#\sigma \\ (\mathsf{ext_a}\sigma)_i &\stackrel{\text{def}}{=} \mathsf{a}\cdot\sigma_i.\end{aligned}$$

Then we define $\mathsf{ss}(\alpha,\sigma)$ to assert that σ is a sequence of the subsequences of α (in the particular order that is produced by an iterative program):

$$\begin{aligned}\mathsf{ss}(\epsilon,\sigma) &\stackrel{\text{def}}{=} \sigma = [\epsilon] \\ \mathsf{ss}(\mathsf{a}\cdot\alpha,\sigma) &\stackrel{\text{def}}{=} \exists\sigma'.\ \mathsf{ss}(\alpha,\sigma') \wedge \sigma = (\mathsf{ext_a}\sigma')^\dagger\cdot\sigma'.\end{aligned}$$

(To obtain the different order produced by a simple recursive program, we would remove the reflection (†) operator here and later.) Next, we define $Q(\sigma,\beta)$ to assert that β is a sequence of lists whose components represent the components of σ:

$$Q(\sigma,\beta) \stackrel{\text{def}}{=} \#\beta = \#\sigma \wedge \forall_{\mathsf{i}=1}^{\#\beta}(\mathsf{list}\,\sigma_\mathsf{i}\,(\beta_\mathsf{i},\mathbf{nil}) \ * \ \mathbf{true}).$$

At this stage, we can specify the abstract behavior of the subsequence program subseq by

$$\begin{aligned}&\{\mathsf{list}\,\alpha\,(\mathsf{i},\mathbf{nil})\} \\ &\mathsf{subseq} \\ &\{\exists\sigma,\beta.\ \ \mathsf{ss}(\alpha^\dagger,\sigma) \wedge (\mathsf{list}\,\beta\,(\mathsf{j},\mathbf{nil}) \ * \ (Q(\sigma,\beta)))\}.\end{aligned}$$

To capture the sharing structure, however, we use iterated separating conjunction to define $R(\beta)$ to assert that the last element of β is the empty list, while every previous element consists of a single integer cons'ed onto some later element of β:

$$\begin{aligned}R(\beta) \stackrel{\text{def}}{=} &(\beta_{\#\beta} = \mathbf{nil} \wedge \mathbf{emp}) \ * \\ &\qquad \textstyle\bigodot_{\mathsf{i}=1}^{\#\beta-1}(\exists\mathsf{a},\mathsf{k}.\ \ \mathsf{i} < \mathsf{k} \le \#\beta \wedge \beta_\mathsf{i} \mapsto \mathsf{a},\beta_\mathsf{k}).\end{aligned}$$

Here the heap described by each component of the iteration contains a single cons-pair, so that the heap described by $R(\beta)$ contains $\#\beta - 1$ cons-pairs. Finally, the full specification of c is

$$\begin{aligned}&\{\mathsf{list}\,\alpha\,(\mathsf{i},\mathbf{nil})\} \\ &\mathsf{subseq} \\ &\{\exists\sigma,\beta.\ \mathsf{ss}(\alpha^\dagger,\sigma) \wedge (\mathsf{list}\,\beta\,(\mathsf{j},\mathbf{nil}) * (Q(\sigma,\beta) \wedge R(\beta)))\}.\end{aligned}$$

Here the heap described by $\mathsf{list}\,\beta\,(\mathsf{j},\mathbf{nil})$ contains $\#\beta$ cons-pairs, so that the entire heap described by the postcondition contains $(2 \times \#\beta) - 1 = (2 \times 2^{\#\alpha}) - 1$ cons-pairs.

8. Proving the Schorr-Waite Algorithm

The most ambitious application of separation logic has been Yang's proof of the Schorr-Waite algorithm for marking structures that contain sharing and cycles [33, 34]. This proof uses the older form of classical separation logic [19] 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.

Several significant features of the proof are evident from its main invariant:

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

The heap described by this invariant is the footprint of the entire algorithm, which is exactly the structure that is reachable from the address root. Since addresses now refer to entire records, it is easy to assert that the record at x has been allocated:

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

that all active records are marked:

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

that x is not a dangling address:

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

or that no active record contains a dangling address:

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

(Yang uses the helpful convention that the predicates with names ending in "R" are those that are not intuitionistic.)

In the second line of the invariant, the subassertion $\mathsf{listMarkedNodesR}(\mathsf{stack}, \mathsf{p})$ holds for a part of the heap called the *spine*, which is a linked list of records from root to the current address t in which the links have been reversed; the variable stack is a sequence of four-tuples determining the contents of the spine. In the next line, $\mathsf{restoredListR}(\mathsf{stack}, \mathsf{t})$ describes what the contents of the spine should be after the links have been restored, and $\mathsf{spansR}(\mathsf{STree}, \mathsf{root})$ asserts that the abstract structure STree is a spanning tree of the heap.

The assertion $\mathsf{spansR}(\mathsf{STree}, \mathsf{root})$ also appears in the precondition of the algorithm. 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 $\mathsf{spansR}(\mathsf{STree}, \mathsf{root})$ is any predicate that is independent of the boolean fields in the records; spanning trees are used only because they are enough to determined the heap, except for the boolean fields.) To the author's knowledge, this part of the invariant is the only conceptual use of separating implication in a real proof (as opposed to its formal use in expressing 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 [27, 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.

9. Computability and Complexity Results

The existence of weakest preconditions for each of our new commands assures us that a central property of Hoare logic is preserved by our extension: that a program specification annotated with loop invariants and recursion hypotheses (and, for total correctness, variants) can be reduced to a collection of assertions, called verification conditions, whose validity will insure the original specification. Thus the central question for computability and complexity is to decide the validity of assertions in separation logic.

Yang [8, 34] has examined the decidability of classical separation logic without arithmetic (where expressions are variables, values are addresses or **nil**, and the heap maps addresses into two-field records). He showed that, even when the characteristic operations of separation logic (**emp**, $\mapsto$, $*$, and $-\!\!*$, but not $\hookrightarrow$) are prohibited, deciding the validity of an assertion is not recursively enumerable. (As a consequence, we cannot hope to find an axiomatic description of $\mapsto$.) On the other hand, Yang and Calcagno showed that if the characteristic operations are permitted but quantifiers are prohibited, then the validity of assertions is algorithmically decidable.

For the latter case, Calcagno and Yang [8] have investigated complexity. Specifically, for the languages tabulated below they considered

MC The model checking problem: Does $[\![p]\!]_{\mathrm{asrt}} sh$ hold for a specified state (s, h)?

VAL The validity problem: Does $[\![p]\!]_{\mathrm{asrt}} sh$ hold for all states (s, h)?

In each case, they determined that the problem was complete for the indicated complexity class:

	Language	**MC**	**VAL**
$\mathcal{L}$	$P ::= E \mapsto E, E \mid \neg\, E \hookrightarrow -, -$ $\mid E = E \mid E \neq E \mid \mathbf{false}$ $\mid P \wedge P \mid P \vee P \mid \mathbf{emp}$	P	coNP
$\mathcal{L}^{*}$	$P ::= \mathcal{L} \mid P * P$	NP	Π^{P}_{2}
$\mathcal{L}^{\neg *}$	$P ::= \mathcal{L} \mid \neg\, P \mid P * P$	PSPACE	
$\mathcal{L}^{-\!*}$	$P ::= \mathcal{L} \mid P -\!\!* P$	PSPACE	
$\mathcal{L}^{\neg * -\!*}$	$P ::= \mathcal{L} \mid \neg\, P \mid P * P \mid P -\!\!* P$	PSPACE	

10. Garbage Collection

Since our logic permits programs to use unrestricted address arithmetic, there is little hope of constructing any general-purpose garbage collector. On the other hand, the situation for the older logic, in which addresses are disjoint from integers, is more hopeful. However, it is clear that this logic permits one to make assertions, such as "The heap contains two elements" that might be falsified by the execution of a garbage collector, even though, in any realistic sense, such an execution is unobservable.

The present author [28] has given the following example (shown here as a proof outline):

$$\begin{aligned}
&\{\mathbf{true}\}\\
&\mathsf{x} := \mathbf{cons}(3,4)\,;\\
&\{\mathsf{x} \hookrightarrow 3,4\}\\
&\{\exists \mathsf{x}.\ \mathsf{x} \hookrightarrow 3,4\}\\
&\mathsf{x} := \mathbf{nil}\\
&\{\exists \mathsf{x}.\ \mathsf{x} \hookrightarrow 3,4\},
\end{aligned}$$

where the final assertion describes a disconnected piece of structure, and would be falsified if the disconnected piece were reclaimed by a garbage collector. In this case, the assertions are all intuitionistic, and indeed it is hard to concoct a reasonable program logic that would prohibit such a derivation.

Calcagno, O'Hearn, and Bornat [7, 6, 5, 4] have explored ways of avoiding this problem by defining the existential quantifier in a nonstandard way, and have defined a rich variety of logics that are insensitive to garbage collection. Unfortunately, there is no brief way of relating these logics to those discussed in this paper.

11. Future Directions

11.1. New Forms of Inference

In Yang's proof of the Schorr-Waite algorithm, there are thirteen assertions that have been semantically validated but do not seem to be consequences of known inference rules, and are far too specific to be considered axioms. This surprising state of affairs is likely a consequence of the novel character of the proof itself — especially the quantification over all allocated addresses, and the crucial use of separating implication — as well as the fact that the algorithm deals with sharing in a more fundamental way than others that have been studied with separation logic.

The generalization of such assertions may be a fertile source of new inference rules. For example, suppose $\hat{p}$ is intuitionistic, and let p be $\forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{p}$. Then

$$\begin{aligned}
\mathbf{emp} &\Rightarrow p\\
(\exists x.\ (x \mapsto -,-,-,-) \wedge \hat{p}) &\Rightarrow p\\
(p \ast p) &\Rightarrow p.
\end{aligned}$$

For a more elaborate example, suppose we say that two assertions are *immiscible* if they cannot both hold for overlapping heaps. More precisely, p and q are immiscible iff, for all stores s and heaps h and h' such that $h \cup h'$ is a function, if $[\![p]\!]_{\mathrm{asrt}} sh$ and $[\![q]\!]_{\mathrm{asrt}} sh'$ then $h \perp h'$.

On the one hand, it is easy to find immiscible assertions: If $\hat{p}$ and $\hat{q}$ are intuitionistic and $\hat{p} \wedge \hat{q} \Rightarrow \mathbf{false}$ is valid, then

$$\forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{p} \qquad \text{and} \qquad \forall x.\ \mathsf{allocated}(x) \Rightarrow \hat{q}$$

are immiscible. Moreover, if p' and q' are immiscible and $p \Rightarrow p'$ and $q \Rightarrow q'$ are valid, then p and q are immiscible.

On the other hand, if p and q are immiscible, then

$$(p \ast \mathbf{true}) \wedge (q \ast r) \Rightarrow q \ast ((p \ast \mathbf{true}) \wedge r)$$

is valid.

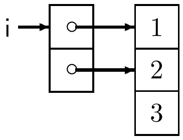
Both of these examples are sound, and useful for at least one proof of a specification of a real program. But they may well be special cases of more generally useful rules or other inference mechanisms. O'Hearn suspects that there are good prospects to find a useful proof theory for separation logic using "labeled deduction" [13, 14].

It should also be noted that Yang's proof depends critically on the fact that the Schorr-Waite algorithm performs an in-place computation. New problems may arise in trying to prove, say, a program that copies a reachable structure while preserving its sharing patterns — somehow, one must express the isomorphism between the structure and its copy.

Beyond these particulars, in its present state separation logic is not only theoretically incomplete, but *pragmatically* incomplete: As it is applied in new ways, there will be a need for new kinds of inference.

11.2. Taming Address Arithmetic

When we first realized that separation logic could be generalized and simplified by permitting address arithmetic, it seemed an unalloyed benefit. But there are problems. For example, consider the definition of dag given in Section 6: The assertion $\mathsf{dag}\ ((1 \cdot 2) \cdot (2 \cdot 3))\ (\mathsf{i})$ holds in states where two distinct records overlap, e.g.



Similarly, iff one were to try to recast the Schorr-Waite proof in the logic with address arithmetic, it would be difficult (but necessary) to assert that distinct records do not overlap, and that all address values in the program denote the first fields of records.

These problems, of what might be called *skewed sharing*, become even more difficult when there are several types of records, with differing lengths and field types.

A possible solution may be to augment states with a mapping from the domain of the heap into "attributes" that could be set by the program, described by assertions, and perhaps tested to determine whether to abort. These attributes would be similar to auxiliary variables (in the sense of Owicki and Gries [25]), in that their values could not influence the values of nonauxiliary variables or heap cells, nor the flow of control.

To redefine $\mathbf{dag}$ to avoid skewed sharing, one could, at each allocation $x := \mathbf{cons}(e_1, e_2)$ that creates a record in a dag, give x (but not $x+1$) the attribute **dag-record**. Then the definition of a non-atomic dag would be changed to

$$\begin{aligned}&\mathsf{dag}\ (\tau_1 \cdot \tau_2)\ (i) \text{ iff } \textbf{is-dag-record}(i) \wedge \\ &\qquad \exists i_1, i_2.\ i \mapsto i_1, i_2 \ast (\mathsf{dag}\ \tau_1\ (i_1) \wedge \mathsf{dag}\ \tau_2\ (i_2)).\end{aligned}$$

In a version of the Schorr-Waite proof using address arithmetic, one might give the attribute **record** to the address of the beginning of each record. Then one would add

$$\begin{aligned}&\forall \mathsf{x}.\ \textbf{is-record}(\mathsf{x}) \Rightarrow \exists \mathsf{l}, \mathsf{r}, \mathsf{c}, \mathsf{m}.\ \mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, \mathsf{c}, \mathsf{m} \\ &\quad \wedge (\textbf{is-record}(\mathsf{l}) \vee \mathsf{l} = \mathbf{nil}) \wedge (\textbf{is-record}(\mathsf{r}) \vee \mathsf{r} = \mathbf{nil}) \\ &\quad \wedge \textbf{is-boolean}(\mathsf{c}) \wedge \textbf{is-boolean}(\mathsf{m})\end{aligned}$$

to the invariant, and use the assertion $\textbf{is-record}(x)$ in place of $\mathsf{allocated}(x)$.

There is a distinct flavor of types in this use of attributes: The attributes record information, for purposes of proof, that in a typed programming language would be checked by the compiler and discarded before runtime. An alternative, of course, would be to move to a typed programming language, but our goal is to keep the programming language low-level and, for simplicity and flexibility, to use the proof system to replace types rather than supplement them.

11.3. Concurrency

In the 1970's, Hoare and Brinch Hansen argued persuasively that, to prevent data races where two processes attempt to access the same storage without synchronization, concurrent programs should be syntactically restricted to limit process interference to explicitly indicated *critical regions*. In the presence of shared mutable structures, however, processes can interfere in ways too subtle to be detected syntactically.

On the other hand, when one turns to program verification, it is clear that separation logic can specify the absence of interference. In the simplest situation, the concurrent execution $c_1 \parallel c_2$ of two processes that do not interfere with one another is described by the inference rule

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

where no variable free in p_1 or q_1 is modified by c_2, or vice-versa.

Going beyond this trivial case, O'Hearn [22, 21] has extended the Hoare-like logic for concurrency devised by Owicki and Gries [25] (which is in turn an extension of work by Hoare [18]), to treat critical regions in the framework of separation logic.

The basic idea is that, just as program variables are syntactically partitioned into groups owned by different processes and resources, so the heap should be similarly partitioned by separating conjunctions in the proof of the program. The most interesting aspect is that the partition of the heap can be changed by executing critical regions associated with resources, so that ownership of a particular address can move from a process to a resource and from there to another process.

Thus, for example, one process may allocate addresses and place them in a buffer, while another process removes the addresses from the buffer and deallocates them. Similarly, in a concurrent version of quicksort, an array segment might be divided between two concurrent recursive calls, and reunited afterwards.

Unfortunately, at this writing there is no proof that O'Hearn's inference rules are sound. The difficulty is that, in O'Hearn's words, "Ownership is in the eye of the asserter", i.e., the changing partitions of the heap are not determined by the program itself, but only by the assertions in its proof.

In concurrent programming, it is common to permit several processes to read the same variable, as long as no process can modify its value simultaneously. It would be natural and useful to extend this notion of *passivity* to heap cells, so that the specification of a process might indicate that a portion of the heap is evaluated but never mutated, allocated or deallocated by the process. (This capability would also provide an alternative way to specify the action of copytree on dags discussed in Section 6.) Semantically, this would likely require activity bits to take on a third, intermediate value of "read-only".

Concurrency often changes the focus from terminating programs to programs that usefully run on forever — for which Hoare logic is of limited usefulness. For such programs, it might be helpful to extend temporal logic with separating operators.

11.4. Storage Allocation

Since separation logic describes a programming language with explicit allocation and deallocation of storage, without any behind-the-scenes garbage collector, it should be suitable for reasoning about allocation methods themselves.

A challenging example is the use of *regions*, in the sense of Tofte et al [31]. To provided an explicitly programmable region facility, one must program an allocator and deallocator for regions of storage, each of which is equipped with its own allocator and deallocator. Then one must prove that the program using these routines never deallocates a region unless it is safe to deallocate all storage that has been allocated from that region.

Although separation logic is incompatible with general-purpose garbage collection (at least when unrestricted address arithmetic is permitted), it should be possible to construct and verify a garbage collector for a specific program. In this situation, one should be able to be far more aggressive than would be possible for a general-purpose garbage collector, both in recovering storage, and in minimizing the extra data needed to guide the traversal of active structure. For example, for the representation described by dag in Section 6, it would be permissible for the collector to increase the amount of sharing.

The key here is that a correct garbage collector need only maintain the assertion that must hold at the point where the collector is called, while preserving the value of certain input variables that determine the abstract values being computed. (In addition, for total correctness, the collector must not increase the value of the loop variants that insure termination.) For instance, in the list-reversal example in Section 5, a collector called at the end of the **while** body would be required to maintain the invariant

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

while preserving the abstract input α_0, and not increasing the variant, which is the length of α.

It is interesting to note that the garbage collector is required to respect the partial equivalence relation determined by the invariant, in which two states are equivalent when they both satisfy the invariant and specify the same values of the input variables. Considering the use of partial equivalence relations to give semantics to types, this reinforces the view that types and assertions are semantically similar.

11.5. The Relationship to Type Systems

Although types and assertions may be semantically similar, the actual development of type systems for programming languages has been quite separate from the development of approaches to specification such as Hoare logic, refinement calculi, or model checking. In particular, the idea that states have types, and that executing a command may change the type of a state, has only taken hold recently, in the study of type systems for low-level languages, such as the *alias types* devised by Walker and Morrisett [32].

Separation logic is closely related to such type systems. Indeed, their commonality can be captured roughly by a simple type system:

$\langle\text{value type}\rangle ::= \mathbf{integer} \mid \langle\text{address variable}\rangle$

$\langle\text{store type}\rangle ::=$

$\quad\langle\text{variable}\rangle\!: \langle\text{value type}\rangle, \ldots, \langle\text{variable}\rangle\!: \langle\text{value type}\rangle$

$\langle\text{heap type}\rangle ::= \mathbf{emp}$

$\quad\mid \langle\text{address variable}\rangle \mapsto \langle\text{value type}\rangle, \ldots, \langle\text{value type}\rangle$

$\quad\mid \langle\text{heap type}\rangle\ *\ \langle\text{heap type}\rangle$

$\langle\text{state type}\rangle ::= \langle\text{store type}\rangle\ ;\ \langle\text{heap type}\rangle$

It is straightforward to translate state types in this system into assertions of classical separation logic (in the formulation without address arithmetic) or into alias types.

It may well be possible to devise a richer type system that accommodates a limited degree of address arithmetic, permitting, for example, addresses that point into the interior of records, or relative addresses.

Basically, however, the real question is whether the dividing line between types and assertions can be erased.

11.6. Embedded Code Pointers

Even as a low-level language, the simple imperative language axiomatized by Hoare is deficient in making no provision for the occurrence in data structures of addresses that refer to machine code. Such code pointers appear in the compiled translation of programs in higher-order languages such as Scheme or SML, or object-oriented languages such as Java or C♯. Moreover, they also appear in low-level programs that use the techniques of higher-order or object-oriented programming.

Yet they are difficult to describe in the first-order world of Hoare logic; to deal with embedded code pointers, we must free separation logic from both Hoare logic and the simple imperative language. Evidence of where to go comes from the recent success of type theorists in extending types to machine language (in particular to the output of compilers) [20, 32]. They have found that a higher-order functional language (with side-effects) comes to resemble a low-order machine language when two restrictions are imposed:

- Continuation-passing style (CPS) is used, so that functions receive their return addresses in the same way as they receive other parameters.
- Free variables are prohibited in expressions that denote functions, so that functions can be represented by code pointers, rather than by closures that pair code pointers with data.

In fact, this restricted language is formally similar to an imperative language in which programs are "flat", i.e., control paths never come together except by jumping to a common label.

This raises the question of how to marry separation logic with CPS (with or without the prohibition of free variables in function expressions, which appears to be irrelevant from a logical point of view).

A simple example of CPS is provided by the function $\mathsf{append}(\mathsf{x}, \mathsf{y}, \mathsf{r})$, which appends the list x to the list y (without mutation, so that the input lists are unchanged), and passes the resulting list on to the continuation r:

```
letrec append(x, y, r) = if x = nil then r(y) else
    let a = [x], b = [x + 1],
        k(z) = let w = cons(a, z) in r(w)
    in append(b, y, k).
```

We believe that the key to specifying such a CPS program is to introduce a *reflection* operator # that allows CPS terms to occur within assertions (in a manner reminiscent of dynamic logic [15]). Specifically, if t is a CPS term then $\#\, t$ is an assertion that holds for a state if t never aborts when executed in that state — in this situation we say that it is *safe* to execute t in that state.

Suppose c is a command satisfying the Hoare specification $\{p\}\ c\ \{q\}$, and t is a CPS term such that executing t has the same effect as executing c and then calling the continuation $r(x_1, ...x_n)$. Then we can specify t by the assertion

$$(p \,*\, (\forall x_1, \ldots, x_m.\ q \mathrel{-\!\!*} \#\, r(x_1, \ldots, x_n))) \Rightarrow \#\, t$$

(where $x_1, \ldots, x_m$ is a subset of the variables $x_1, \ldots, x_n$). If we ignore the difference between the separating operators and ordinary conjunction and implication, this asserts that it is safe to execute t in any state satisfying p and mapping r into a procedure such that it is safe to execute $r(x_1, ..., x_n)$ in a state satisfying q. Taking into account the separative nature of $*$ and $\mathrel{-\!\!*}$, it asserts that it is safe to execute t in any state where part of the heap satisfies p, and r is mapped into a procedure such that it is safe to execute $r(x_1, ..., x_n)$ when the part of the heap that satisfied p is replaced by a part that satisfies q. Finally, the universal quantifier indicates the variables whose value may be changed by t before it executes $r(x_1, ..., x_n)$.

For example, the CPS form of append can be specified by

$$\begin{aligned}&\Big((\mathsf{list}\,\alpha\,(\mathsf{x}, \mathbf{nil}) * \mathsf{list}\,\beta\,(\mathsf{y}, \mathbf{nil}))\\ &\quad * \Big(\forall \mathsf{z}.\ (\mathsf{list}\,\alpha\,(\mathsf{x}, \mathbf{nil}) * \mathsf{list}\,\alpha\,(\mathsf{z}, \mathsf{y}) * \mathsf{list}\,\beta\,(\mathsf{y}, \mathbf{nil}))\\ &\qquad \mathrel{-\!\!*} \#\, \mathsf{r}(\mathsf{z})\Big)\Big)\\ &\ \Rightarrow \#\, \mathsf{append}(\mathsf{x}, \mathsf{y}, \mathsf{r}).\end{aligned}$$

It can be discouraging to go back and read the "future directions" sections of old papers, and to realize how few of the future directions have been successfully pursued. It will be fortunate if half of the ideas and suggestions in this section bear fruit. In the meantime, however, the field is young, the game's afoot, and the possibilities are tantalizing.

Acknowledgements

For encouragement and numerous suggestions, I am indebted to many researchers in separation logic, as well as the students in courses I have taught on the subject. I'm particularly grateful to Peter O'Hearn for many helpful comments after reading a preliminary draft of this paper.

However, most of the opinions herein, and all of the errors, are my own.

References

[1] R. Bornat. Explicit description in BI pointer logic. Unpublished, June 2001.

[2] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[3] L. 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.

[4] C. Calcagno. Program logics in the presence of garbage collection (abstract). In F. Henglein, J. Hughes, H. Makholm, and H. Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, page 11. IT University of Copenhagen, 2001.

[5] C. Calcagno. *Semantic and Logical Properties of Stateful Programmming*. Ph. D. dissertation, Università di Genova, Genova, Italy, 2002.

[6] C. Calcagno and P. W. O'Hearn. On garbage and program logic. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, 2001. Springer-Verlag.

[7] C. Calcagno, P. W. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. Submitted July 2001, 2001.

[8] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In R. Hariharan, M. 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.

[9] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In M. Hennessy and P. Widmayer, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, Berlin, 2002. Springer-Verlag.

[10] L. Cardelli and G. 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.

[11] L. Cardelli and A. 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.

[12] G. Conforti, G. 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.

[13] D. Galmiche and D. Méry. Proof-search and countermodel generation in propositional BI logic. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 263–282, Berlin, 2001. Springer-Verlag.

[14] D. Galmiche, D. Méry, and D. J. Pym. Resource tableaux. Submitted, 2002.

[15] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, Massachusetts, 2000.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[17] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

[18] 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.

[19] S. Ishtiaq and P. 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.

[20] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In X. Leroy and A. Ohori, editors, *Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Berlin, 1998. Springer-Verlag.

[21] P. W. O'Hearn. Notes on conditional critical regions in spatial pointer logic. Unpublished, August 31, 2001.

[22] P. W. O'Hearn. Notes on separation logic for shared-variable concurrency. Unpublished, January 3, 2002.

[23] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2001. Springer-Verlag.

[25] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[26] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, Boston, Massachusetts, 2002. (to appear).

[27] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.

[28] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[29] J. C. Reynolds. Lectures on reasoning about shared mutable data structure. IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6-13, 2000.

[30] J. C. Reynolds and P. W. O'Hearn. Reasoning about shared mutable data structure (abstract of invited lecture). In F. Henglein, J. Hughes, H. Makholm, and H. 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.

[31] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, New York, 1994. ACM Press.

[32] D. Walker and G. Morrisett. Alias types for recursive data structures. In R. W. Harper, editor, *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, Berlin, 2001. Springer-Verlag.

[33] H. Yang. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In F. Henglein, J. Hughes, H. Makholm, and H. 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.

[34] H. Yang. *Local Reasoning for Stateful Programs*. Ph. D. dissertation, University of Illinois, Urbana-Champaign, Illinois, July 2001.

[35] H. Yang and P. 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.