An Overview of Separation Logic

John C. Reynolds*

Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract. After some general remarks about program verification, we introduce *separation logic*, a novel extension of Hoare logic that can strengthen the applicability and scalability of program verification for imperative programs that use shared mutable data structures or shared-memory concurrency.

1 Introduction

Proving programs is not like proving mathematical theorems. A mathematical conjecture often gives no hint as to why it might be true, but a program is written by a programmer who inevitably has at least an informal understanding of why the program might behave as it should. The goal of program verification is not to search some huge proof space, but to formalize the programmer's own reasoning to the point where any flaws become evident.

Thus, I believe that programs should be proved as they are written, and that the programmer must be intimately involved in the proof process.

At Syracuse University, from 1972 to 1986, I taught a course on structured programmming and Hoare-style program proving to at least a thousand students, mostly master's candidates in computer science. This experience convinced me that good programmers can annotate and prove their programs rigorously, and in doing so achieve vast improvements in the quality of these programs.

No mechanical aids were used — not even a compiler. Indeed, much of the effectiveness of the course came from forcing the students to produce logically correct programs without debugging.

To scale from classroom examples to modern software, however, effective mechanical assistance will be vital. Since proofs are as prone to errors as programs are, they must be checked by machine. Moreover, to avoid drowning in a sea of minutiae, the programmer must be freed from trivial arguments that are amenable to efficient decision procedures.

On the other hand, interaction with the programmer is vital. The ultimate goal of creating error-free software will not be met by filtering the results of

* Portions of the author's own research described in this position paper were supported by National Science Foundation Grants CCR-9804014 and CCR-0204242, by the Basic Research in Computer Science (http://www.brics.dk/) Centre of the Danish National Research Foundation, and by EPSRC Visiting Fellowships at Queen Mary, University of London, and Edinburgh University.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 460–469, 2008.

conventional programming through any form of post-hoc verification. Instead specification and verification must be tightly interwoven with program construction — and thus require a logic that is concise and readable.

2 Separation Logic

Around the turn of the millenium, Peter O'Hearn and I devised an extension of Hoare logic called "separation logic" [1,2,3,4]. Our original goal was to facilitate reasoning about low-level imperative programs that use shared mutable data structure. Extensions of the logic, however, have proven applicable to a wider conceptual range, where access to memory is replaced by permission to exercise capabilities, or by knowledge of structure. In a few years, the logic has become a significant research area, with a growing literature produced by a variety of researchers.

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

For example, consider the following program, which performs an in-place reversal of a list:

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

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

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

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

(Here the predicate $\mathsf{list}\,\alpha\,\mathsf{i}$ can be read "i is a list representing the sequence α.")

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

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

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

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

A second, more general, advantage of separation logic is the support it gives to *local reasoning*, which underlies the scalability of the logic. For example, using

the invariant given above, one can prove the following specification of the list-reversal program:

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

The semantics of this specification implies that the addressable storage described by the precondition $\{\mathsf{list}\ \alpha\ \mathsf{i}\}$, which is the storage containing the list i representing α, is the *only* addressable storage touched by the execution of $LREV$ (often called the *footprint* of $LREV$). If $LREV$ is a part of a larger program that also manipulates some separate storage, say containing a list k representing a sequence γ, then one can use an inference rule due to O'Hearn, called the *frame rule* [3]:

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}}$$

(where c does not assign to the free variables of r), to infer directly that the additional storage is unaffected by $LREV$:

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

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

There is little need for local reasoning in proving toy examples. But it has been critical in proving more complex programs, such as the Schorr-Waite marking algorithm [5,6] or the Cheney copying garbage collector [7].

It should also be mentioned that the assertion $\mathsf{list}\ \alpha\ \mathsf{i}$ is only true for an addressible storage containing the relevant list and nothing else. Thus either of the above specifications of $LREV$ indicates that $LREV$ does not cause a memory leak.

3 Semantics of the Logic

Separation logic describes programs in an extension of the simple imperative language with commands for allocating, accessing, mutating, and deallocating data structures, but without garbage collection. A critical feature of this language is that any attempt to dereference a dangling pointer causes program execution to abort.

In Hoare logic, the state of a computation is a mapping from variables to integers, which we will henceforth call a *store*. Thus:

$$\text{Stores}_V = V \to \text{Integers}\ ,$$

where V is a set of variables. In separation logic, however, there is a second component of the state, called a *heap*, which formalizes the addressable storage

where mutable structures reside. Specifically, the heap maps some finite set of *active* addresses into integers, where *addresses* are a proper subset of the integers:

$$\text{Addresses} \subset \text{Integers}$$

$$\text{Heaps} = \bigcup_{A \subseteq_{\text{fin}} \text{Addresses}} (A \to \text{Integers})$$

$$\text{States}_V = \text{Stores}_V \times \text{Heaps}\,.$$

The nature of the novel commands in our programming language can be illustrated by a sequence of state transitions:

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

The indeterminacy of allocation must be emphasized; for example, $\mathsf{x}{:=}\mathbf{cons}(1, 2)$ could augment the domain of the heap with any two consecutive addresses that are not already in that domain.

It should also be noted that **cons** and the square brackets are part of the syntax of command forms, rather than parts of expressions. As a consequence, expressions do not depend on the heap and do not have side-effects.

A second example shows the effect of an attempt to mutate a dangling pointer:

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

Similar faults are also caused by out-of-range lookup or deallocation.

The assertions of separation logic go beyond the predicate calculus used in Hoare logic by providing four new forms for describing heaps:

- **emp**
  The heap is empty.
- $e \mapsto e'$
  The heap contains a single cell, at address e with contents e'.
- $p_1 * p_2$
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
- $p_1 -\!\!* p_2$
  If the current heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

The separating conjunction $*$ is associative and commutative, with **emp** as its neutral element. The operation $-\!\!*$, called the *separating implication*, is adjoint to $*$. Neither the law of contraction nor the law of weakening hold for $*$.

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

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

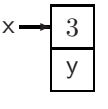
$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$$

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

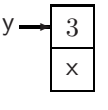
$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e+n-1 \hookrightarrow e_n$$
$$\text{iff } e \mapsto e_1, \ldots, e_n * \mathbf{true}\,.$$

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

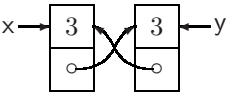
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β, α is an address, and the heap maps α into 3 and $\alpha + 1$ into β).



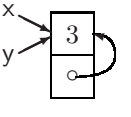
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x.



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

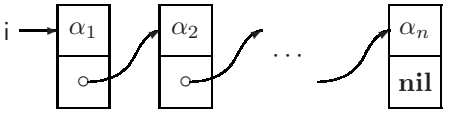


4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



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

It is also possible (and, except in trivial cases, necessary) to define predicates by structural induction over abstract data types. For example, the predicate $\mathsf{list}\,\alpha\,\mathsf{i}$:

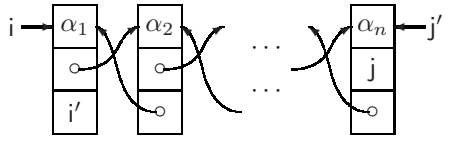


can be defined by structural induction on the sequence α:

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

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

A more elaborate representation of sequences is provided by *doubly-linked* list segments. 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 (again on sequences) is:

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

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

It is straightforward to use inductive definitions over any initial algebra without laws. An obvious example is the set S-exps of S-expressions in the sense of LISP, which is the least set satisfying

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

(where atoms are integers that are not addresses).

Suppose we call the obvious representation of S-expressions without sharing a *tree*, and the analogous representation with possible sharing (but without cycles) a *dag*. Then we can define

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

$$\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(i) \stackrel{\text{def}}{=} \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ *\ \mathsf{tree}\,\tau_1\,(i_1) \ *\ \mathsf{tree}\,\tau_2\,(i_2)$$

and

$$\mathsf{dag}\,a\,(i) \stackrel{\text{def}}{=} i = a$$

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

(Notice that $\mathsf{tree}\,\tau\,(i)$ describes a heap containing a tree-representation of τ and nothing else, while $\mathsf{dag}\,\tau\,(i)$ describes a heap that may properly contain a dag-representation of τ.)

4 Extending Hoare Logic

The meaning of the triples used to specify commands is roughly similar to that of Hoare logic. More precisely, however, the partial-correctness triple $\{p\}\ c\ \{q\}$ holds iff, starting in any state in which p holds:

- No execution of c aborts.
- If any execution of c terminates in a final state, then q holds in the final state.

Notice that the universal quantification in this definition extends over both store and heap components of states, and also over the multiple possible executions that arise from the indeterminacy of allocation.

Also notice that, even for partial correctness, specifications preclude abortion. This is a fundamental characteristic of the logic — that (to paraphrase Milner) well-specified programs do not go wrong (when started in states satisfying their precondition). As a consequence, one can implement well-specified programs without runtime memory-fault checking.

An obviously analogous treatment of total correctness is straightforward.

Except for the so-called rule of constancy, which is replaced by the frame rule, all the inference rules of Hoare logic remain sound in separation logic. In addition, there are rules for the new heap-manipulating commands. For instance, for mutation there is a *local* rule:

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

from which one can use the frame rule to infer a *global* rule:

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

There is also a *backward reasoning* form of the rule that uses separating implication:

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

These three rules are interderivable, and the last describes weakest preconditions.

A similar situation holds for allocation, lookup, and deallocation, although the first two cases are complicated by the need to use quantifiers to describe variable overwriting.

5 Present Accomplishments

At present, separation logic has been used to verify manually a variety of small programs, as well as a few that are large enough to demonstrate the potential of local reasoning for scalability [5,6,7]. In addition:

1. It has been shown that deciding the validity of an assertion in separation logic is not recursively enumerable, even when address arithmetic and the characteristic operation **emp**, $\mapsto$, $*$, and $-\!\!*$, but not $\hookrightarrow$ are prohibited [8,6]. On the other hand, it has also been shown that, if the characteristic operations are permitted but quantifiers are prohibited, then the validity of assertions is algorithmically decidable within the complexity class PSPACE [8].
2. A decision procedure has been devised for a restricted form of the logic that is capable of shape analysis of lists [9].
3. An iterated form of separating conjunction has been introduced to reason about arrays [4].
4. The logic has been extended to procedures with global variables, where a "hypothetical frame rule" permits reasoning with information hiding [10]. Recently, a further extension to higher-order procedures (in the sense of Algol-like languages) has been developed [11].
5. Separation logic itself has been extended to a higher-order logic [12].
6. The logic has been integrated with data refinement [13,14].
7. The logic has been extended to shared-variable concurrency with conditional critical regions, where one can reason about the transfer of ownership of storage from one process to another [15,16].
8. Fractional permissions (in the sense of Boyland) and counting permissions have been introduced so that one can permit several concurrent processes to have read-only access to an area of the heap [17].
9. In the context of proof-carrying code, separation logic has inspired work on proving run-time library code for dynamic allocation [18].

6 The Future

As with Hoare logic, it is difficult in separation logic to assert relations between states at different points in a program; one must use ghost variables, but now their values may need to be finite functions or relations. Moreover, the usual notation for such functions or relations is distractingly different from assertions about the current local heap. (These complications are evident in the proof of the Cheney algorithm [7].)

It is likely that the extension to higher-order logic mentioned above will alleviate this problem; for example, it should be possible to use ghost variables to denote the past values of heaps.

Recent advances in generalizing the logic have outstripped our experience in proving actual programs. We particularly need further experience proving programs where the sharing patterns of data structures convey semantic information — for example, when a cyclic graph is used to represent a network. And we have just begun to explore concurrent programs.

If separation logic is going to have an impact on verification, we must be able to automate proof-checking, at a high enough level to avoid trivial details. Judging by recent preliminary research, the likely route here is to express separation logic in a system such as Coq (as in [18]) or Isabelle, so that the inference rules of separation logic become theorems in the underlying logic.

Moreover, the logic must be adapted to real-world languages. This is likely to be relatively straightforward for languages such as C or assembly languages, but a number of issues arise for higher-level languages. These include the treatment of complex type systems (including typed values in the heap), garbage collection [19] (perhaps coexisting with explicitly allocated storage or some form of regions), and the presence of code pointers or closures in the heap [4,20].

A final goal is an integration of logic and type systems. One would like to have a type system for shared mutable data structures, with at least the expressiveness of Walker and Morrisett's alias types [21], that is a checkable sublanguage of separation logic.

References

1. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millennial Perspectives in Computer Science, Houndsmill, Hampshire, Palgrave, pp. 303–321 (2000)
2. Ishtiaq, S., O'Hearn, P.W.: 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, pp. 14–26. ACM, New York (2001)
3. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, Springer, Heidelberg (2001)
4. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science, Los Alamitos, California, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
5. Yang, H.: An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In: Henglein, F., Hughes, J., Makholm, H., Niss, H. (eds.) SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, IT University of Copenhagen, pp. 41–68 (2001)
6. Yang, H.: Local Reasoning for Stateful Programs. Ph. D. dissertation, University of Illinois, Urbana-Champaign, Illinois (2001)
7. Birkedal, L., Torp-Smith, N., Reynolds, J.C.: Local reasoning about a copying garbage collector. In: Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 220–231. ACM Press, New York (2004)
8. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 108–119. Springer, Heidelberg (2001)
9. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
10. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 268–280. ACM Press, New York (2004)

11. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules. In: Proceedings Twentieth Annual IEEE Symposium on Logic in Computer Science, Los Alamitos, California, IEEE Computer Society, Los Alamitos (2005)
12. Biering, B., Birkedal, L., Torp-Smith, N.: Bi-hyperdoctrines and higher order separation logic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 233–247. Springer, Heidelberg (2005)
13. Mijajlović, I., Torp-Smith, N.: Refinement in a separation context. In: SPACE 2004: Informal Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (2004)
14. Mijajlović, I., Torp-Smith, N., O'Hearn, P.W.: Refinement and separation contexts. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 421–433. Springer, Heidelberg (2004)
15. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
16. Brookes, S.D.: A semantics for concurrent separation logic. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 16–34. Springer, Heidelberg (2004)
17. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.: Permission accounting in separation logic. In: Conference Record of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 259–270. ACM Press, New York (2005)
18. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for PCC: Dynamic storage allocation. Science of Computer Programming 50, 101–127 (2004)
19. Calcagno, C., O'Hearn, P.W., Bornat, R.: Program logic and equivalence in the presence of garbage collection. Theoretical Computer Science 298, 557–581 (2003)
20. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: Research Report YALEU/CS/TR-1294, Yale University, New Haven, Connecticut (2005), `http://flint.cs.yale.edu/flint/publications/xcap.html`
21. Walker, D., Morrisett, G.: Alias types for recursive data structures. In: Harper, R. (ed.) TIC 2000. LNCS, vol. 2071, pp. 177–206. Springer, Heidelberg (2001)