Separation Logic Tutorial

(To appear in Proceedings of ICLP'08)

Peter O'Hearn[⋆]
Queen Mary, University of London

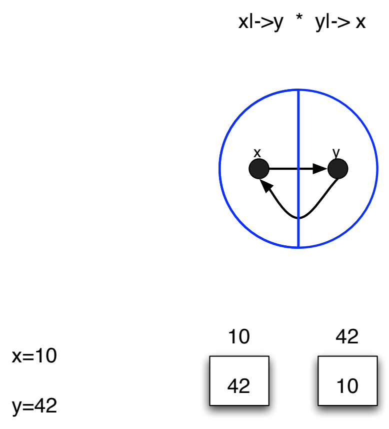
Separation logic is an extension of Hoare's logic for reasoning about programs that manipulate pointers. It is based on the *separating conjunction* $P * Q$, which asserts that P and Q hold for separate portions of computer memory.

This tutorial on separation logic has three parts.

1. *Basics.* Concentrating on highlights from the early work [1–4].
2. *Model Theory.* The model theory of separation logic evolved from the general resource models of bunched logic [5–7], and includes an account of program dynamics in terms of their interaction with resource [8, 9].
3. *Proof Theory.* I will describe those aspects of the proof theory, particularly new entailment questions (frame and anti-frame inference [10, 11]), which are important for applications in mechanized program verification.

1 Basics

The Separating Conjunction. I introduce the separating conjunction by example. Consider the following memory structure.



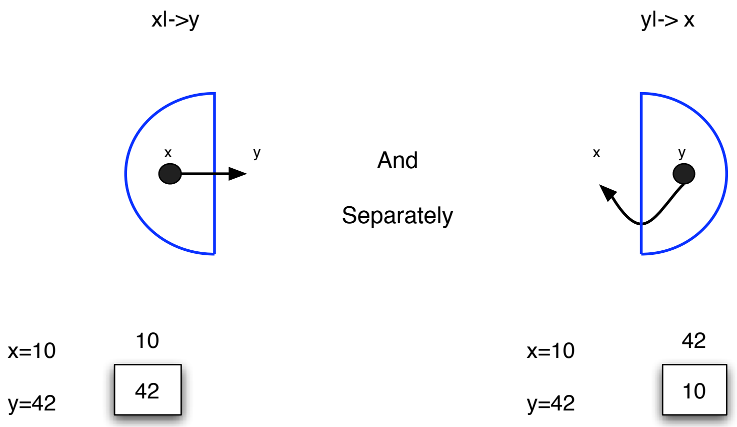
We read the formula at the top of this figure as "x points to y, *and separately* y points to x". Going down the middle of the diagram is a line which represents a heap partitioning: a separating conjunction asks for a partitioning that divides memory into parts satisfying its two conjuncts.

At the bottom of the figure we have given an example of a concrete memory description that corresponds to the diagram. There, x and y have values 10 and

[⋆] I gratefully acknowledge the support of an EPSRC Advanced Fellowship and a Royal Society Wolfson Research Merit Award.

42 (in the "environment", or "register bank"), and 10 and 42 are themselves locations with the indicated contents (in the "heap", or even "RAM"). It should be clear how the picture corresponds to the concrete structure. It is simplest to think in terms of the picture semantics of separation logic, but if confusion arises as to what diagrams mean you can always drop down to the RAM level.

The indicated separating conjunction above is true of the pictured memory because the parts satisfy the conjuncts. That is, the components



are separate sub-states that satisfy the relevant conjuncts.

It can be confusing to see a diagram like the one on the left where "x points to y and yet to nowhere". This is disambiguated in the RAM description below the diagram. In the more concrete description x and y denote values (10 and 42), x's value is an allocated memory address which contains y's value, but y's value is not allocated. Notice also that, in comparison to the first diagram, the separating conjunction splits the heap/RAM, but it does *not* split the association of variables to values: heap cells, but not variable associations, are deleted from the original situation to obtain the sub-states.

When reasoning about programs that manipulate data structures, one normally wants to use inductively-defined predicates that describe such structures. Here is a definition for a predicate that describes binary trees:

$$\begin{aligned}\mathsf{tree}(E) \iff & \mathit{if}\ E = \mathsf{nil}\ \mathit{then}\ \mathsf{emp} \\ & \mathit{else}\ \exists x, y.\, (E \mapsto l\colon x, r\colon y) \ast \mathsf{tree}(x) \ast \mathsf{tree}(y)\end{aligned}$$

In this definition we have used a record notation $(E \mapsto l\colon x, r\colon y)$ for a "points-to predicate" that describes a single[1] record E that contains x in its l field and y in its r field. nil can be taken to be any non-addressible number[2]. The separating conjunction between this assertion and the two recursive instances of tree ensures that there are no cycles, and the separating conjunction between the two subtrees ensures that we have a tree and not a dag. The emp predicate in the base case

[1] It denotes a *singleton heap* , a heaplet wth only one cell.
[2] You can map these notions to the RAM model, or just imagine a record model.

of the inductive definition describes the empty heap (or portion of heap). A consequence of this is that when $\mathsf{tree}(E)$ holds there are no extra cells, not in the tree, in a state satisfying the predicate. This is a key specification pattern often employed in separation logic proofs: we use assertions that describe only as much state as is needed, and nothing else.

At this point you might think that I have described an exotic-looking formalism for writing assertions about heaps and you might wonder: why bother? In fact, the mere ability to describe heaps is not at all important in and of itself, and in this separation logic adds nothing significant to traditional predicate logic. It is only when we consider the interaction between assertions and operations for mutating memory that the point of the formalism begins to come out.

In-place Reasoning. I am going to try something that might seem eccentric: I am going to give you a program proof, without telling you the inference rules it uses. I am hoping that you will find the reasoning steps I show to be intuitively understandable, prior to becoming embroiled in too many formalities. Whether I succeed in my aim is, of course, for you to judge.

Consider the following procedure for disposing the elements in a tree.

```
procedure DispTree($p$)
local $i, j$;
if  $p \neq \mathsf{nil}$ then
    $i = p \to l$ ; $j := p \to r$; DispTree($i$); DispTree($j$); free($p$)
```

This is the expected procedure that walks a tree, recursively disposing left and right subtrees and then the root pointer. It uses a representation of tree nodes with left, right and data fields, and the empty tree is represented by nil.

The specification of DispTree is just

$$\{\mathsf{tree}(p)\}\ \mathtt{DispTree}(p)\ \{\mathsf{emp}\}$$

which says that if you have a tree at the beginning then you end up with the empty heap at the end. The crucial part of the proof, in the `if` branch, is:

```
{$p \mapsto [l\colon x, r\colon y] * \mathsf{tree}(x) * \mathsf{tree}(y)$}
   $i$ := $p \to l$; $j$ := $p \to r$;
{$p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$}
  DispTree($i$);
{$p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j)$}
  DispTree($j$);
{$p \mapsto [l\colon i, r\colon j]$}
  free $p$
{emp}
```

After we enter the conditional statement we know that $p \neq \mathsf{nil}$, so that (according to the inductive definition) p points to left and right subtrees occupying separate storage. Then the roots of the two subtrees are loaded into i and j. The first recursive call operates in-place on the left subtree, removing it, the second call removes the right subtree, and the final instruction frees the root pointer p. This verification uses the procedure specification as a recursive assumption.

I am leading to a more general suggestion: try thinking about reasoning in separation logic as if you are an interpreter. The formulae are like states, *symbolic* states. Execute the procedure forwards, updating formulae in the usual way you do when thinking about in-place update of memory. In-place reasoning works not only for disposal, but for heap mutation and allocation as well [1, 2].

One thing at work in the "proof" above is a rule

$$\frac{\{P\}\ C\ \{Q\}}{\{R * P\}\ C\ \{R * Q\}}\ \text{Frame Rule}$$

that lets us tack on additional assertions "for free", as it were. For instance, in the second recursive call the frame axiom R selected is $p\mapsto[l\colon i, r\colon j]$ and $\{P\}C\{Q\}$ is a substitution instance of the procedure spec: this captures that the recursive call does not alter the root pointer. Generally, the frame rule that lets us use "small specifications" that only talk about the cells that a program touches [3].

Perspective. The essential points that I have tried to illustrate are the following.

(i) The separating conjunction fits together with inductive definitions in a way that supports natural descriptions of mutable data structures [1].
(ii) Axiomatizations of pointer operations support *in-place reasoning*, where a portion of a formula is updated in place when passing from precondition to postcondition, mirroring the operational locality of heap update [1, 2].
(iii) Frame axioms, which state what does not change, can be avoided when writing specifications [2, 3].

These points together enable specifications and proofs for pointer programs that are dramatically simpler than was possible previously, in many (not all) cases approaching the simplicity associated with proofs of pure functional programs.

2 Model Theory and Proof Theory

Above I have concentrated on the basics of separation logic, emphasizing that to "think like an interpreter" is a good approximation to program proving. The model-theoretic underpinnings of this point of view rest on a number of theorems about the semantics of imperative programs, and their interaction with the semantics of Hoare triples [8, 9].

The most significant developments in proof theory have stemmed from an inference procedure of Berdine and Calcagno in their work on the Smallfoot tool [12]. Special versions of their inference rules have been used to enable loop-invariant discovery in abstract interpreters [13, 14], which have been extended to ever-more-expressive abstract domains (e.g., [15–19]).

A pivotal development has been identification of the notion of *frame inference*, which gives a way to find the "leftover" portions of heap needed to automatically apply the frame rule in program proofs. Technically, this is done by solving an extension to the usual entailment question

$$A \vdash B * \texttt{?frame}$$

where the task is, given A and B, to find a formula `?frame` which makes the entailment valid. This extended entailment capability is used at procedure call sites, where A is an assertion at the call site and B a precondition from a procedure's specification. Frame inference was first solved by Berdine and Calcagno by using information from failed proofs of the standard entailment question $A \vdash B$ (related ideas were developed in [20]). It is used in several automatic verification and analysis tools based on separation logic [21, 16, 22–24].

More recently, there has been work on an, in a sense, inverse problem

$$A * \texttt{?anti-frame} \vdash B$$

where the task is to find a description of the missing or needed portion of heap `?anti-frame` that makes the entailment valid. This is a separation-logic cousin of the classic abductive inference question. It has been used in [11] to synthesize preconditions of procedures, by attempting to infer descriptions of just the portions of heap that they need to run without producing a memory fault. The joint abduction/frame inference question, termed "bi-abduction" in [11], forms the basis of a compositional program analysis, where Hoare triples for a procedure are generated without knowing the procedure's calling context.

I have concentrated on the basics of separation logic, on its semantics, and on proof theory as it is relevant to automatic proof tools and abstract interpreters. There have been significant developments in several other directions.

– Iteractive proof, where the semantics of the logic is embedded in a higher-order logic (e.g., [25–27]).
– Web data structures, using non-symmetric separation (context with hole)[28].
– Object-oriented programing, where the logic is used to address longstanding aliasing problems (e.g., [29, 24]).
– Concurrency, where the logic is used to control sharing of memory between concurrent threads (starting with [30, 31]).

Space prevents more comprehensive references here: The reader may consult the page www.dcs.qmul.ac.uk/∼ohearn/localreasoning.html for further pointers.

References

1. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare.
2. S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.
3. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1-19, 2001.
4. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp55-74, 2002.
5. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
6. D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Applied Logic Series. Kluwer Academic Publishers, 2002.

7. D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.
8. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, 2002. pp402-416.
9. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd LICS*, pp366-378, 2007.
10. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, volume 3780 of *LNCS*, 2005.
11. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis. Imperial College DOC Tech Report 2008/12.
12. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.
13. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *12th TACAS*. pp287-302, 2006.
14. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. *3rd SPACE Workshop*, 2006.
15. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV*. pp386-400, 2006.
16. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. *PLDI 2007*.
17. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
18. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. *19th CAV, 2007*.
19. S. Magill, M.-S. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. *20th CAV, 2008*.
20. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd POPL*, pp296–309, 2005.
21. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *13th SAS*. pp240-260, 2006.
22. H.H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. *20th CAV, 2008*.
23. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. *20th CAV, 2008*.
24. D. Distefano and M. Parkinson. jStar: Towards Practical Verification for Java. *OOPSLA, 2008*.
25. N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. *3rd SPACE Workshop*, 2006.
26. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *34th POPL*, pages 97–108, 2007.
27. M.O. Myreen and M.J.C. Gordon. Hoare logic for realistically modelled machine code. 13th TACAS, 2007.
28. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local Hoare reasoning about DOM. In *27th PODS*, pages 261–270, 2008.
29. M. Parkinson and G. Bierman. Separation logic and abstraction. In *32nd POPL*, pp59–70, 2005.
30. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science (Reynolds Festschrift)*, 375(1-3):271–307, 2007.
31. S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science (Reynolds Festschrift)*, 375(1-3):227–270, 2007.