A Brief Introduction to Separation Logic

Robert Frohardt

December 4, 2009

Abstract

Separation logic[5] is a modern system for reasoning about program correctness. It is an extension of Hoare logic that allows reasoning about the state of the heap. It has also been used to reason about other situations involving shared resources, e.g., concurrency with shared memory. A complete understanding of separation logic requires a background in first-order logic and Hoare logic. The purpose of this paper is to give a brief overview of this background and of separation logic without worrying about the details. Rather, my goal is to give a reader with relatively little background knowledge an idea of what separation logic is and how it is being used in current research. The target audience of this paper is a first-year Ph.D. student in computer science.

1 First-order Logic

For an introduction to logic with a focus on applications in computer science, see [3]. A more formal mathematical introduction is given in [1]. This section gives a brief and informal description of some of the key ideas from logic that are necessary for understanding separation logic.

1.1 Propositional Logic

Propositional logic describes simple *propositions*, often referred to as sentences, and the basic connectives for forming more complex sentences. We commonly use p, q, etc. to represent sentences. The basic connectives are:

$$\begin{aligned} \neg &: \text{negation (i.e., `not')} \\ \wedge &: \text{conjunction (i.e., `and')} \\ \vee &: \text{disjunction (i.e., `inclusive or')} \\ \Rightarrow &: \text{implication (i.e., `if ... then ...')} \end{aligned}$$

For example, we could define p and q as follows:

$$\begin{aligned} p &: \text{“Music is the food of love.”} \\ q &: \text{“I order you to continue playing.”} \end{aligned}$$

We can this combine these sentences (although not very poetically) using the connectives:

$$\begin{aligned}
\neg p :& \text{“Music is \textit{not} the food of love”} \\
p \wedge q :& \text{“Music is the food of love \textit{and} I order you to continue playing.”} \\
\neg p \vee q :& \text{“\textit{Either} music is not the food of love, \textit{or} I order you to continue} \\
& \text{playing, \textit{or both}.”} \\
p \Rightarrow q :& \text{“\textit{If} music is the food of love, \textit{then} I order you to continue playing.”}
\end{aligned}$$

Note that in formal logic the last two sentences are equivalent, although it is not clear whether they are equivalent in everyday speech.

1.2 First-order Logic

First-order logic extends propositional logic by adding *variables* and *quantifiers*. In the following examples, x is a variable. The quantifiers of first-order logic are $\forall$ (for all) and $\exists$ (there exists). We can also define n-ary *predicates* (i.e., relations) and functions. For example, we could define two unary relations, R and D as:

$$\begin{aligned}
R(x) :& \text{ “x is rotten.”} \\
D(x) :& \text{ “x is in the state of Denmark.”}
\end{aligned}$$

In these these two logical formulas, the variable x is said to be *free*. Combining these with the existential quantifier we have:

$$\begin{aligned}
\exists x.(R(x) \wedge D(x)) :& \text{“There exists something which is rotten} \\
& \text{and is in the state of Denmark.”}
\end{aligned}$$

In this logical formula, the variable x is *bound*. It only has local meaning within the scope of the $\exists$.

1.3 Sequent Calculus

Sequent calculus is a formal proof system introduced by Gentzen [2]. A *sequent* is an expression of the form $\Gamma \vdash \Delta$, where Γ is a set of *premises* and Δ is a set of *conclusions*. The premises and conclusions are formulas in first-order logic. The meaning of the sequent is that there is a derivation of the conclusions from the premises. This is a purely syntactic claim about derivation within the proof system. *Inference rules* define how one sequent can be derived from others in a proof. These rules are subdivided into two categories: logical rules and structural rules.

An example of a logical rule is:

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \tag{1.1}$$

This rule is sometimes called $\wedge$ introduction, and it states that if:

1. Whenever the premises Γ are true, P must be true
2. Whenever the premises Γ are true, Q must be true

Then we can conclude that whenever the premises Γ are true, $P \wedge Q$ must be true. This may appear to be a trivial observation, but combining this inference rule with the others we have a system for formally specifying the rules of mathematical proof. Furthermore, this type of system can potentially be used for automatic theorem proving (although in practice there are better systems for automation).

Two examples of structural rules are:

$$\frac{\Gamma \vdash Q}{\Gamma, P \vdash Q}\ \textit{Weakening} \qquad \frac{\Gamma, P, P \vdash Q}{\Gamma, P \vdash Q}\ \textit{Contraction} \tag{1.2}$$

These two rules make intuitive sense. For example, contraction states that if Γ, P and P imply Q then we did not have to write the premise P down twice, we could have written it just once. However, while these rules may seem obvious we will see in Section 3 that they do not hold in Separation Logic.

2 Hoare Logic

For an introduction to Hoare logic see [3, 6]. This paper follows the notation in [6], which is more standard. *Hoare logic* (also known as Floyd-Hoare logic) is a system for proving the correctnesss of computer programs. Hoare logic works with *Hoare triples* which are expressions of the form:

$$\{P\}\mathbf{c}\{Q\}$$

This is a statement about *partial correctness.* It states that if P is true and the program **c** runs, then *if the program halts*, Q will be true in the final state. There is a different notation for reasoning about *total correctness* (i.e., that the program does halt and that Q is true in the final state), but total correctness is not addressed here.

Hoare logic gives inference rules, similar to those described in Section 1.3, for creating proofs of program correctness. One very simple example of a *Hoare rule* is:

$$\frac{}{\{P\}\mathbf{skip}\{P\}}$$

Note that writing nothing above the line indicates that the statement below the line is always true. This rule states that running a **skip** statement that does nothing has no effect on P. Another straightforward rule is the *sequencing rule*, which states what happens if two commands run sequentially:

$$\frac{\{P\}\mathbf{c_0}\{R\} \quad \{R\}\mathbf{c_1}\{Q\}}{\{P\}\mathbf{c_0};\mathbf{c_1}\{Q\}}$$

One rule that we will focus on for comparison with separation logic is the rule for assignments. The rule is:

$$\frac{}{\{P[e/x]\}\mathbf{x} := \mathbf{e}\{P\}} \tag{2.1}$$

Where $P[e/x]$ is the formula P with all free occurrences of x replaced by the expression e. This rule may appear confusing at first, but a simple example shows its meaning. Suppose that after running the command $\mathbf{x} := \mathbf{3}$ we want to conclude $x == 3$. The assignment rule tells us how to find a precondition that leads to this conclusion. In our desired conclusion, we substitute 3 for x, which yields $3 == 3$ which is equivalent to *true*. Therefore we conclude that the following is a valid Hoare triple:

$$\{true\}\mathbf{x} := \mathbf{3}\{x == 3\}$$

Similarly, our program may have another variable y and we wish to conclude the apparently obvious fact that the above assignment does not affect the value of y. In this case, we can use the assignment rule to derive:

$$\{y == 4\}\mathbf{x} := \mathbf{3}\{x == 3 \wedge y == 4\}$$

Unfortunately, this reasoning becomes problematic when we introduce the heap. Using the notation of [5], we write:

- $[x]$ denotes the contents at address x (i.e., dereferencing)
- $x \mapsto e$ means that x points to e
- $x \mapsto -$ means that x has been allocated, but does not specify the value pointed to by x.

Now we wish to look at the assignment $[\mathbf{x}] := \mathbf{3}$ and see what we can conclude. If we wanted to use the old assignment rule, we might try rewriting it with one small change as:

$$\frac{}{\{x \mapsto - \wedge P[e/[x]]\}[\mathbf{x}] := \mathbf{e}\{P\}} \tag{2.2}$$

That is, we take the old assignment rule (2.1) and add the requirement that an address is active before we try to set the value there. This rule is unsound, although for the postcondition $[x] == 3$ it appears to work correctly because it gives us the triple:

$$\{x \mapsto -\}[\mathbf{x}] := \mathbf{3}\{[x] == 3\}$$

However, suppose we now want to conclude that this assignment does not affect the value stored at address y. Using our new ad hoc assignment rule (2.2), we would conclude:

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

Of course, this is not valid because it does not consider the possibility that $x == y$, in which case after the assignment statement we would have $[y] == 3$. We could create a valid triple here by adding another precondition:

$$\{x \mapsto - \wedge [y] == 4 \wedge \neg(x == y)\}[\mathbf{x}] := \mathbf{3}\{[x] == 3 \wedge [y] == 4\}$$

The problem is that this quickly becomes unwieldy in a large program where we would have to specify in each precondition that some list of addresses has no overlap with the addresses affected by a command.

3 Separation Logic

Separation logic improves this situation by adding two new logical connectives:

$$\begin{aligned} * &: \text{separating conjunction} \\ -\!\!* &: \text{separating implication} \end{aligned}$$

We also introduce a new value *emp* which follows the rule $P * emp \iff P$. In the context of a programming language with a heap we interpret these new symbols as follows:

- *emp* : The heap is empty.
- $P * Q$: The heap contains disjoint parts such that P holds in one and Q holds in the other.
- $P -\!\!* Q$: If the heap were extended with a disjoint part such that P holds, then Q holds for the new larger heap.

We can see that this simplifies the problem of specifying preconditions. For example, now we can write:

$$\{x \mapsto - * y \mapsto 4\}[\mathbf{x}] := \mathbf{3}\{x \mapsto 3 * y \mapsto 4\} \tag{3.1}$$

The separating conjunction has the fact that $\neg(x == y)$ built into it because it asserts that $x \mapsto -$ and $y \mapsto 4$ hold for disjoint parts of the heap. For the complete statement of the assignment rule and its use to prove (3.1) see Appendix A.

In fact, assuming that the semantics of the programming language satisfy certain requirements, we have an inference rule called the *frame rule*:

$$\frac{\{P\}\mathbf{c}\{Q\}}{\{P * R\}\mathbf{c}\{Q * R\}}$$

Which is valid as long as no free variable in R is modified by **c**. The power of this rule is that it allows local reasoning. Using the frame rule we can prove statements about the local effects of a sequence of commands, and then immediately conclude that statements about unaffected regions of the heap remain valid.

The above description of separation logic is specific to programming with a heap, but separation logic more generally can be applied to other situations where we are concerned with analyzing access to shared resources. One way of looking at the differences between first-order logic and separation logic is to look at how some of the inference rules change in separation logic. For example, the rule for $*$ introduction is given by (cf. (1.1)):

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1, \Gamma_2 \vdash P * Q}$$

One way of thinking about this rule is that it says that if we want to prove that some set of premises Γ implies $P * Q$ then we need to split Γ into two sets Γ_1 and Γ_2 which we can use separately to prove P and Q. Another important observation is that the structural rules from first-order logic do not all hold in separation logic. For this reason, separation logic is called a *substructural logic.* Both of the weakening and contraction rules (1.2) are invalid in separation logic. For more information on the relationship between separation logic and substructural logics see [4].

4 Acknowledgements

Special thanks to Evan Chang for providing many references about this material and for very useful discussions both about separation logic and the motivation of this material. Thanks to Andrzej Ehrenfeucht for suggesting the structure of this paper. Thanks to Sriram Sankaranarayanan for some corrections to the first draft. This paper was presented as the final project for CSCI 5444, Introduction to the Theory of Computation, taught by Andrzej Ehrenfeucht.

References

[1] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, Inc., 1972.

[2] Gerhard Gentzen. Untersuchungen ueber das logische schliessen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[3] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, second edition, 2004.

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

[5] John C. Reynolds. An introduction to separation logic. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15818As2009/cs818A3-09.html`, 2009. Lecture notes from course in separation logic.

[6] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

A Assignments in separation logic

Similarly to the analogous rule in Hoare logic, the inference rule for assignments in separation logic is confusing at first. However, an example of this rule's application helps clarify it. The rule is:

$$\frac{}{\{(x \mapsto -) * ((x \mapsto e) -\!* P)\}[\mathbf{x}] := \mathbf{e}\{P\}} \tag{A.1}$$

Suppose we want to prove (3.1) using (A.1). That is, we would like our precondition to be the simpler one given in (3.1), but the inference rule for assignments requires a more complicated precondition. The precondition that (A.1) tells us we need in order to conclude our desired postcondition is given by:

$$\{(x \mapsto -) * ((x \mapsto 3) -\!*(x \mapsto 3 * y \mapsto 4))\}[\mathbf{x}] := \mathbf{3}\{x \mapsto 3 * y \mapsto 4\} \tag{A.2}$$

Comparing (3.1) and (A.2) we see that we wish to show that:

$$y \mapsto 4 \Rightarrow x \mapsto 3 -\!*(x \mapsto 3 * y \mapsto 4)$$

In fact, this is true due to one of the inference rules of separation logic:

$$\frac{P * Q \Rightarrow R}{P \Rightarrow (Q -\!* R)}$$