Published in *Software Safety and Security; Tools for Analysis and Verification*. NATO Science for Peace and Security Series, vol 33, pp286-318, 2012

A Primer on Separation Logic (and Automatic Program Verification and Analysis)

Peter W. O'Hearn [1]
Queen Mary University of London

Abstract. These are the notes to accompany a course at the Marktoberdorf PhD summer school in 2011. The course consists of an introduction to separation logic, with a slant towards its use in automatic program verification and analysis.

Keywords. Program Logic, Automatic Program Verification, Abstract Interpretation, Separation Logic

1. Introduction

Separation logic, first developed in papers by John Reynolds, the author, Hongseok Yang and Samin Ishtiaq, around the turn of the millenium [73,47,61,74], is an extension of Hoare's logic for reasoning about programs that access and mutate data held in computer memory. It is based on the *separating conjunction* $P * Q$, which asserts that P and Q hold for separate portions of memory, and on program-proof rules that exploit separation to provide modular reasoning about programs.

In this course I am going to introduce the basics of separation logic, its semantics, and proof theory, in a way that is oriented towards its use in automatic program-proof tools and abstract interpreters, an area of work which has seen increasing attention in recent years. After the basics, I will describe how the ideas can be used to build a verification or program analysis tool.

The course consists of four lectures:

1. *Basics,* where the fundamental ideas of the logic are presented in a semi-formal style;
2. *Foundations,* where we get into the fomalities, including the semantics of the assertion language and axioms and inference rules for heap-mutating commands, and culminating in an account of the local dynamics which underpin some of the rules in the logic;

[1] This work was supported by funding from the Royal Society, the EPSRC and Microsoft Research.

3. *Proof Theory and Symbolic Execution*, which describes a way of reasoning about programs by 'executing' programs on formulae rather than concrete states, and which can form the basis for an automatic verifier; and
4. *Program Analysis*, where abstraction is used to infer loop invariants and other annotations, increasing the level of automation.

These course notes include two sections based on the first two lectures, followed by a section collecting ideas from the last two lectures. At this stage the notes are incomplete, and they will possibly be improved and extended in the future. I hope, though, that they will still prove useful in giving a flavour of some of the main lines of work, as well as in pointers into the literature. In particular, at the end I give references to current directions being pursued in program analysis.

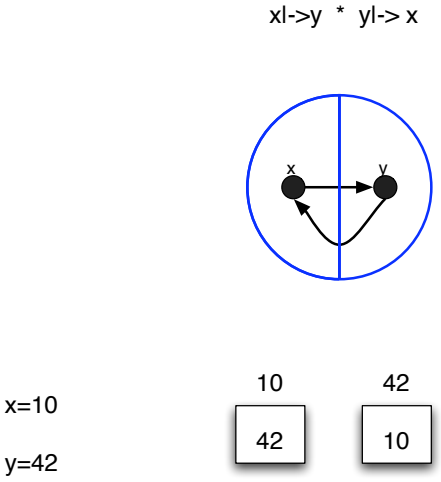
I should say that, with this slant towards automatic proof and program analysis, there are active ongoing developments related to separation logic in several other directions that I will not be able to cover, particularly in concurrency, data abstraction and refinement, object-oriented languages and scripting languages; a small sample of work in these directions includes [62,64,66,10,81,34,28,38].

2. Basics

In this section I introduce separation logic in a semi-formal way. I am hoping that some of the ideas can strike home and be seen to reflect natural reasoning that programmers might employ, even before we consider formal definitions. Of course, the informal presentation inevitably skates over some issues, issues that could very well lead to unsound conclusions if not treated correctly, and to nail things down we will get to the definitions in the next section.

2.1. The Separating Conjunction.

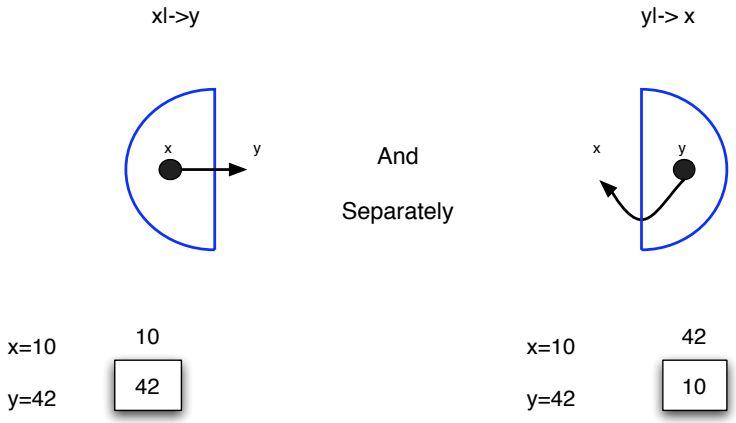
Consider the following memory structure.



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 is an example of a concrete memory description that corresponds to the diagram. There, x and y have values 10 and 42 (in the 'environment', or 'register bank'), and 10 and 42 are themselves locations with the indicated contents (in the 'heap', or even 'RAM').

The indicated separating conjunction above is true of the pictured memory because the parts satisfy the corresponding 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 nothing'. 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. It is usually simplest to think in terms of the picture semantics of separation logic, but when we get formal in the next section we will drop down to the RAM level (as we could always do when pressed).

In general, an assertion P denotes a set of states, and $P * Q$ is true of a state just if its heap/RAM component can be split into two parts, one of which satisfies P and the other of which satisfies Q.

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:

$$\begin{aligned} \mathit{tree}(E) \iff & \texttt{if isatom?}(E) \texttt{ then } \mathit{emp} \\ & \texttt{else } \exists xy.\, E \mapsto [l\colon x, r\colon y] \;*\; \mathit{tree}(x) \;*\; \mathit{tree}(y) \end{aligned}$$

In this definition we assume a boolean expression $\texttt{isatom?}(E)$ which distinguishes atomic values (e.g., characters...) from addressible locations: in the RAM model, we

could say that the locations are the non-negative integers and the atoms the negative ones. We have used a record notation $E{\mapsto}[l\colon x, r\colon y]$ for a 'points-to predicate' that describes a single record E that contains x in its l field and y in its r field. Again in the RAM model, this binary points-to can be compiled into the unary one: That is, $E{\mapsto}[l\colon x, r\colon y]$ could be an abbreviation for $(E{\mapsto}x) * (E{+}1{\mapsto}y)$ (Or, you could imagine a model where the heap consists of explicit records with field selection.) The separating conjunction between the $E{\mapsto}[l\colon x, r\colon y]$ assertion and the two recursive instances of *tree* in the definition 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 of the inductive definition describes the empty heap, the heap with no allocated cells. A consequence of this is that when *tree*(E) holds there are no extra cells, cells in the heap but 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? The mere ability to describe heaps in principle is not important in and of itself, and in this separation logic adds nothing significant to traditional predicate logic. It is when we consider the interaction between assertions and operations for mutating memory that the point of the formalism comes out.

2.2. In-place Reasoning

Proving by Executing. I am going to show you part of a program proof outline in separation logic. It might seem slightly eccentric that I do this before giving you a definition of the logic. My aim is to use a computational reading of the proof steps to motivate the inference rules, rather than starting from them.

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

```
procedure DispTree(p)
local i, j;
if ¬isatom?(p) then
    i := p→l; j := p→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 as cells containing left and right pointers, with the base case corresponding to atomic, non-pointer values. (See Exercise 2 below for a fuller description.)

The specification of `DispTree` is just

$$\{\mathit{tree}(p)\}\ \texttt{DispTree}(p)\ \{\mathit{emp}\}$$

which says that if you have a tree at the beginning then you end up with the empty heap at the end. For this to make sense it is crucial that when *tree*(p) is true of a heap then that heap (or rather the heaplet, a portion of a global heap) contains all and only those

cells in the tree. So, the spec talks about as small a portion of the global program state as possible.

The crucial part of the argument for `DispTree`'s correctness, in the `then` branch, can be pictured with the following annotated program which gives a 'proof by execution' style argument.

$$
\begin{array}{l}
\{p{\mapsto}[l{:}\,x, r{:}\,y] * \mathit{tree}(x) * \mathit{tree}(y)\} \\
\quad i := p{\to}l; j := p{\to}r; \\
\{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathit{tree}(i) * \mathit{tree}(j)\} \\
\quad \texttt{DispTree}(i); \\
\{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathit{emp} * \mathit{tree}(j)\} \\
\{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathit{tree}(j)\} \\
\quad \texttt{DispTree}(j); \\
\{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathit{emp}\} \\
\quad \texttt{free}\ p \\
\{\mathit{emp}\}
\end{array}
$$

After we enter the `then` branch of the conditional we know that $\neg\texttt{isatom?}(p)$, so that (according to the inductive definition of the *tree* predicate) 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 two consecutive assertions in the middle of the proof are an application of the rule of consequence of Hoare logic. These two assertions are equivalent because *emp* is the unit of $*$. Continuing on, the second call removes the right subtree, and the final instruction frees the root pointer p. The assertions, and their mutations, follow this operational narrative.

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 code forwards, updating formulae in the usual way you do when thinking about in-place update of memory. In-place reasoning works not only for freeing a cell, but for heap mutation and allocation as well. And, it even works for larger-scale operations such as entire procedure calls: we updated the assertions in-place at each of the recursive call sites during this 'proof'.

Exercise 1 *The usual Hoare logic rules for sequencing and consequence are*

$$\frac{\{P\}\,C_1\,\{Q\} \quad \{Q\}\,C_2\,\{R\}}{\{P\}\,C_1; C_2\,\{R\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\,C\,\{Q'\} \quad Q' \Rightarrow Q}{\{P\}\,C\,\{Q\}}$$

where $\Rightarrow$ refers to implication. Assuming we know how to decide $\Rightarrow$ formulae, convert the annotated program block for the `then` *case above into a proof in the usual logical sense (that is, a tree built from instances of these rules). You can assume that the triples of pre/post for each of the individual statements in the proof outline are given as axioms.*

Local Reasoning and Frame Axioms. In the steps in the proof outline for $\texttt{DispTree}(p)$ I used the procedure spec as an assumption when reasoning about the recursive calls, as usual when reasoning about recursive procedures in Hoare logic [43]. However, there is an extra ingredient at work. For the second recursive call, for instance, the assertion at

the call site does not match the procedure specification's precondition, even after p in the spec is instantiated with j, because the assertion has an extra $*$-conjunct, $p{\mapsto}[l\colon i, r\colon j]$.

$$\begin{array}{ll} \text{Assertion at call site}: & p{\mapsto}[l\colon i, r\colon j] * \mathit{tree}(j) \\ \text{Precondition in spec}: & \mathit{tree}(j) \end{array}$$

This extra $*$-conjunct is not touched by the recursive call. It is called a 'frame axiom' in AI. The terminology 'frame axiom' comes from an analogy with animation, where the moving parts of a scene are successively aid over an unchanging frame. Indeed, the fact $p{\mapsto}[l\colon i, r\colon j]$ is left unchanged by the second call. You should be able to pick out the frame in the first call as well.

Thus, there is something slightly awry in this 'proof', unless I tell you more: The mismatch, between the call sites and the procedure precondition, needs to be taken care of if we are really to have a proof of the procedure. One way to resolve the mismatch would be to complicate the specification of the procedure, to talk explicitly about frames in one way or another (see 'back in the day' below). A better approach is to have a generic inference rule, which allows us to avoid mentioning the frames at all in our specifications, but to bring them in when needed. This generic rule is

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

and it 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.

This better way, which avoids talking about frames in specifications, corresponds to programming intuition. When reasoning about a program we should only have to talk about the resources it accesses (its 'footprint'), as all other resources will remain unchanged. This is the *principle of local reasoning* [47,61]. In the specification of `DispTree` the precondition *tree*(p) describes only those cells touched by the procedure.

Aside: back in the day... This issue of local reasoning has nothing to do with the 'power' or 'completeness' of a formal method: what is possible to do in principle. It has only to do with the simplicity and directness of the specs and proofs. To see the issue more clearly, consider how we might have written a spec for `DispTree`(p) in traditional Hoare logic, before we had the frame rule. Here is a beginning attempt:

$$\big\{\mathit{tree}(p) \wedge \texttt{reach}(p, n)\big\}\ \texttt{DispTree}(p)\ \big\{\neg\texttt{allocated}(n)\big\}$$

assuming that we have defined the predicates that say when p points to a (binary) tree in memory, when n is reachable (following l and r links) from p, and when n is allocated. This spec says that any node n which is in the tree pointed to by p is not allocated on conclusion.

While this specification says part of what we would like to say, it leaves too much unsaid. It does not say what the procedure does to nodes that are not in the tree. As a result, this specification is too weak to use at many call sites. For example, consider the first recursive call, `DispTree`(i), to dispose the left subtree. If we use the specification (instantiating p by i) as an hypothesis, then we have a problem: the specification does

not rule out the possibility that the procedure call alters the right subtree j, perhaps creating a cycle or even disposing some of its nodes. As a consequence, when we come to the second call $\texttt{DispTree}(j)$, we will not know that the required *tree*(j) part of the precondition will hold. So our reasoning will get stuck.

We can fix this 'problem' by making a stronger specification which includes frame axioms.

$$\begin{array}{l}\{\textit{tree}(p) \wedge \texttt{reach}(p,n) \wedge \neg\texttt{reach}(p,m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \\ \neg\texttt{allocated}(q)\} \\ \texttt{DispTree}(p) \\ \{\neg\texttt{allocated}(n) \wedge \neg\texttt{reach}(p,m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \\ \neg\texttt{allocated}(q)\}\end{array}$$

The additional parts of the spec say that any allocated cell not reachable from p has the same contents in memory and that any previously unallocated cell remains unallocated. The additional clauses are the frame axioms. (I am assuming that m, m', n and q are auxiliary variables, guaranteed not to be altered. The reason why, say, the predicate $\neg\texttt{allocated}(q)$ could conceivably change, even if q is constant, is that the $\texttt{allocated}$ predicate refers to a behind-the-scenes heap component. f is used in the spec as an arbitrary field name.)

Whether or not this more complicated specification is correct, I think you will agree: it is complicated! I expect that you will agree as well that it is preferable for the frame axioms to be left out of specs, and inferred when needed.

Beyond Shapes. The above shows one inductive definition, for binary trees. The definition is limited in that it does not talk about the contents of a tree. It is the kind of definition often used in automatic shape analysis, as we will describe in Section 4, where avoiding taking about the contents can make it easier to prove entailments or synthesize invariants.

To illustrate the limitation of the definition, suppose that we were to write a procedure to copy a tree rather than delete it. We could give it a specification such as

$$\{\textit{tree}(p)\}\ q := \texttt{CopyTree}(p)\ \{\textit{tree}(p) * \textit{tree}(q)\}$$

but then this specification would also be satisfied by a procedure that rotates a tree as it copies. A more precise specification would be of the form

$$\{\textit{tree}(p,\tau)\}\ q := \texttt{CopyTree}(p)\ \{\textit{tree}(p,\tau) * \textit{tree}(q,\tau)\}$$

where *tree*(p,τ) is a predicate which says that p points to a data structure in memory representing the *mathematical* tree τ. (I use the term 'mathematical' tree to distinguish if from a representation in the computer memory: the mathematical tree does not contain pointer or other such representation information.)

Exercise 2 *The notion of 'mathematical tree' appropriate to the above inductive definition of the* $\texttt{tree}$ *predicate is that of an s-expression (the terminology comes from Lisp): that is, an atom, or a pair of s-expressions. An s-expression is an element of the least set satisfying the equation*

$$\texttt{Sexp} = \texttt{Atom} + (\texttt{Sexp} \times \texttt{Sexp})$$

for some set `Atom` *of atoms, where* $\times$ *and* $+$ *are the cartesian product and disjoint union of sets. Here, then, is the inductive definition of a tree*(p, τ) *predicate, where* $\tau \in$ `Sexp`*:*

$$\begin{aligned} \mathit{tree}(E, \tau) \iff\ & \texttt{if}\ (\texttt{isatom?}(E) \wedge E = \tau)\ \texttt{then}\ \mathit{emp} \\ & \texttt{else}\ \exists xy\tau_1\tau_2.\ \tau = \langle \tau_1, \tau_2 \rangle \wedge (E{\mapsto}[l{:}\, x, r{:}\, y]\ *\ \mathit{tree}(x)\ *\ \mathit{tree}(y)) \end{aligned}$$

Define the `CopyTree` *procedure and give a proof-by-execution style argument for its correctness, where you put assertions (symbolic states) at the appropriate program points. Yes, I am asking you to do a 'proof' in a formalism that has not yet been defined (!), but give it a try.*

2.3. Perspective.

In this section I have attempted to illustrate the following points.

(i) The separating conjunction fits together with inductive definitions in a way that supports natural descriptions of mutable data structures.
(ii) The separating conjunction supports *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.
(iii) Frame axioms, which state what does not change, can be avoided when writing specifications.

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. (That is, previous approaches excepting the remarkable precursor work of Burstall [15], which provided inspiration for Reynolds's earliest work on separation logic [73]. You can see [12] for references and a good account of work on proving pointer programs before separation logic.)

However, I should stress at once that program proofs do not always go as easily as for `DispTree`. When one considers graph algorithms with significant sharing, or concurrent programs with nontrivial interaction, proofs can become complicated. Neither separation logic nor any other formalism takes programs that are difficult to understand and magically gives them easy proofs.

A more realistic goal is to have *simple proofs for simple programs*. Whether, or to what extent, this might be achieved by any given formalism can only be decided personally by you looking at, or better by doing, proofs of a number of examples.

3. Foundations

Building on the ideas described informally in the previous section, I now give a rigorous treatment of the program logic.

3.1. Semantics of Assertions (the Heaplet Model)

The model has two components, the store and the heap. The store is a finite partial function mapping from variables to integers, and the heap is a finite function from natural numbers to integers.

$$\texttt{Stores} \triangleq \texttt{Variables} \rightharpoonup_{fin} \texttt{Ints} \qquad \texttt{Heaps} \triangleq \texttt{Nats} \rightharpoonup_{fin} \texttt{Ints}$$

($\triangleq$ abbreviates 'is defined to be equal to'.) In logic, what we are calling the store is often called the valuation, and the heap is a possible world. In programming languages, what we are calling the store is also sometimes called the environment (the association of values to variables).

We have standard integer expressions E and boolean expressions B built up from variables and constants. These are heap-independent, so determine denotations

$$[\![E]\!]s \in \texttt{Ints} \qquad [\![B]\!]s \in \{true, false\}$$

where the domain of $s \in \texttt{Stores}$ includes the free variables of E or B. We leave this semantics unspecified.

We use the following notations in the semantics of assertions.

1. *dom*(h) denotes the domain of definition of a heap $h \in \texttt{Heaps}$, and *dom*(s) is the domain of $s \in \texttt{Stores}$;
2. $h \# h'$ says that *dom*$(h) \cap$ *dom*$(h') = \emptyset$;
3. $h \bullet h'$ denotes the union of functions with disjoint domains, which is undefined if the domains overlap;
4. $(f \mid i \mapsto j)$ is the partial function like f except that i goes to j.

The satisfaction judgement $s, h \models P$ which says that an assertion holds for a given store and heap, assuming that the free variables of P are contained in the domain of s.

$s, h \models B$ iff $[\![B]\!]s = true$

$s, h \models E \mapsto F$ iff $\{[\![E]\!]s\} =$ *dom*(h) and $h([\![E]\!]s) = [\![F]\!]s$

$s, h \models false$ never

$s, h \models P \Rightarrow Q$ iff if $s, h \models P$ then $s, h \models Q$

$s, h \models \forall x.P$ iff $\forall v \in \texttt{Ints}. [s \mid x \mapsto v], h \models P$

$s, h \models emp$ iff $h = [\,]$ is the empty heap

$s, h \models P * Q$ iff $\exists h_0, h_1.\ h_0 \# h_1,\ h_0 * h_1 = h,\ s, h_0 \models P$ and $s, h_1 \models Q$

$s, h \models P \mathrel{-\!\!*} Q$ iff $\forall h'.$ if $h' \# h$ and $s, h' \models P$ then $s, h \bullet h' \models Q$

The semantics of the connectives $(\Rightarrow\ false, \forall)$ gives rise to meanings of other connectives of classical logic $(\exists, \vee, \neg, true)$ in the usual way. For example, taking $P \wedge Q$ to be $\neg(P \Rightarrow \neg Q)$, we obtain that $s, h \models P \wedge Q$ has the usual meaning of '$s, h \models P$ and $s, h \models Q$'.

The general logical context of this form of semantics is that it can be seen as a possible world model which combines:

(i) the standard semantics of classical logic $(\Rightarrow, false, \forall)$ in the complete boolean algebra of the power set of heaps; and

(ii) a semantics of 'substructural logic' $(emp, *, -\!\!*)$ in the same power set (which gives us what is known as a residuated commutative monoid, an ordered commutative monoid where $A * (-)$ has a right adjoint $A -\!\!* (-)$).

The semantics is an instance of the 'resource semantics' of bunched logic devised by David Pym [63,70,69], where one starts from a partial commutative monoid in place of heaps (with $\bullet$ and the empty heap giving partial monoid structure). The resulting mathematical structure on the powerset, of a boolean algebra with an additional commutative residuated monoid structure, is sometimes called a 'boolean BI algebra'. The model of $\bullet$ as heap partitioning, which lies at the basis of separation logic, was discovered by John Reynolds when he first described the separating conjunction [73]. The separating conjunction was connected with Pym's general resource semantics in [47].

Notice that the semantics of $E \mapsto F$ requires that E is the only active address in the current heap. Using $*$ we can build up descriptions of larger heaps. For example, $(10 \mapsto 3) * (11 \mapsto 10)$ describes two adjacent cells whose contents are 3 and 10. We can express an inexact variant of points-to as follows

$$E \hookrightarrow F \quad = \quad (true * E \mapsto F).$$

Generally, $true * P$ says that P is true of a subheap of the current one. The difference between $\hookrightarrow$ and $\mapsto$ shows up in the presence or absence of projection or Weakening for $*$.

1. $P * (x \mapsto 1) \Rightarrow (x \mapsto 1)$ is not always true.
2. $P * (x \hookrightarrow 1) \Rightarrow (x \hookrightarrow 1)$ is always true.

The different way that the two conjunctions $*$ and $\wedge$ behave is illustrated by the following examples.

1. $(x \mapsto 2) * (x \mapsto 2)$ is unsatisfiable (you can't be in two places at once).
2. $(x \mapsto 2) \wedge (x \mapsto 2)$ is equivalent to $x \mapsto 2$.
3. $(x \mapsto 1) * \neg(x \mapsto 1)$ is satisfiable (thus, we have a kind of 'paraconsistent' logic).
4. $(x \mapsto 1) \wedge \neg(x \mapsto 1)$ is unsatisfiable.

The third example drives home how separation logic assertions do not talk about the global heap: $P * \neg P$ can be consistent because P can hold of one portion of heap and $\neg P$ of another. To understand separation logic assertions you should always think locally: for this you might regard the h component in the semantics of assertions as describing a 'heaplet', a portion of heap, rather than a complete global heap in and of itself.

Exercise 3 *Define $\mapsto$ in terms of $\hookrightarrow$, $\wedge$, $*$, $\neg$ and emp.*

Aside: on $\mapsto$ versus $=$ A frequent source of confusion when first learning separation logic concerns how the $*$ separator splits the heap but not the store, and this translates into confusions reading assertions with $=$ in them. Recall the definitions above

$$s, h \models B \qquad \text{iff } [\![B]\!]s = true$$

$$s, h \models E \mapsto F \text{ iff } \{[\![E]\!]s\} = \mathit{dom}(h) \text{ and } h([\![E]\!]s) = [\![F]\!]s.$$

Notice that the rhs of the clause for $s, h \models B$ does not mention h at all, where for $s, h \models E \mapsto F$ the rhs does contain h. I said I would not give a precise semantics of

boolean expressions, but let me consider just one, the expression $x = y$ where x and y are variables:

$$[\![x = y]\!]s \triangleq (sx = sy).$$

Now, consider the assertion $(x = y) * (x = y)$. Can it ever be true? Well, yes, it is satisfiable, and in fact it has the same meaning as $x = y$ and as $(x = y) \wedge (x = y)$. On the other hand, consider the assertion $(x \mapsto y) * (x \mapsto y)$. Can it ever be true? How about $(x = y) * (x \mapsto y)$? Or $(x = y) \wedge (x \mapsto y)$? Work out the answers to these questions by expanding the semantic definitions.

3.2. Inductive Definitions, Again

Earlier we considered an inductive definition of trees representing s-expressons.

$$\begin{array}{l} \mathit{tree}(E, \tau) \iff \texttt{if } (\texttt{isatom?}(E) \wedge E = \tau) \texttt{ then } \mathit{emp} \\ \qquad\qquad\quad \texttt{else } \exists xy\tau_1\tau_2.\ \tau = \langle \tau_1, \tau_2 \rangle \wedge (E \mapsto [l\colon x, r\colon y] \ * \ \mathit{tree}(x) \ * \ \mathit{tree}(y)) \end{array}$$

Now we can be more precise about its meaning. The use of if-then-else can be desugared using boolean logic connectives in the usual way. $\texttt{if } B \texttt{ then } P \texttt{ else } Q$ is the same as $(B \wedge P) \vee (\neg B \wedge Q)$ where here B is heap-independent. Therefore, in the inductive definition we can now see that the condition $(\texttt{isatom?}(E) \wedge E = \tau)$ is completely heap-independent, and not affected by $*$: it talks only about values, and not the contents of heap cells.

It is also helpful to ponder the clause

$$\tau = \langle \tau_1, \tau_2 \rangle \wedge (E \mapsto [l\colon x, r\colon y] \ * \ \mathit{tree}(x) \ * \ \mathit{tree}(y))$$

used in the definition. In fact, we could have rewritten it using $*$ in place of $\wedge$, as

$$(\tau = \langle \tau_1, \tau_2 \rangle \wedge \mathit{emp}) * (E \mapsto [l\colon x, r\colon y] \ * \ \mathit{tree}(x) \ * \ \mathit{tree}(y)).$$

Here, we have used a general identity

$$B \wedge P \iff (B \wedge \mathit{emp}) * P$$

which holds whenever B is heap-independent. On the other hand, if we replaced one of the other occurrences of $*$ by $\wedge$, it would more dramatically alter the definition (exercise: by playing with $*$, $\wedge$ and perhaps inserting true, can you alter this definition so that it describes dags rather than trees?).

In case you missed it, to be fully formal in the interpretation of this definition we should also extend the store type to be

$$\texttt{Stores} \triangleq \texttt{Variables} \rightharpoonup_{fin} (\texttt{Ints} + \texttt{Sexp})$$

so that a variable can take on an s-expression as well as an integer value. We could also distinguish s-expression variables τ from program variables x syntactically. (In practice,

one would probably want to use a many-sorted rather than one-sorted logic as we are doing in these notes for theoretical simplicity.)

Finally, we can regard $E \mapsto [l\colon x, r\colon y]$ as sugar for $(E \mapsto x) * (E+1 \mapsto y)$ in the RAM model. Note, though, that this low-level desugaring is not part of the essence of separation logic, only this particular model. Other models can be used where heaps are represented by $L \rightharpoonup_{fin} V$ where V might be a structured type to represent records. However, that the RAM model *can* be used is appealing in a foundational way, as we know that programs of all kinds are eventually compiled to such a model (modern concerns with weak memory notwithstanding).

Generally, for any kind of data structure you will want to provide an appropriate predicate definition which will often be inductive. Linked lists are the most basic case, and illustrate some of the issues involved.

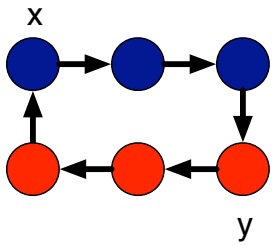
When reasoning about imperative data structures, one needs to consider not only complete linked lists (terminated with nil) but also 'partial lists' or linked-list segments. Here is an example of a list segment predicate describing lists from E to F (where F is not allocated).

$$ls(E, F) \iff \texttt{if } E = F \texttt{ then } emp \\ \texttt{else } \exists y. E \mapsto y * ls(y, F)$$

I am intending that ls is the least predicate satisfying the equation. Mathematically, it can be worked out as the least fixed-point of a monotone function on a certain lattice, by reference to the Tarski fixed-point theorem. (Exercise: what is the lattice and what is the monotone function?) It is possible as well to give an alternate definition whose formalization does not need to talk about lattices: you define a predicate $ls(E, F, n)$ describing a linked list segment from E to F of length n, and then define $ls(E, F)$ to be $\exists n.\, ls(E, F, n)$.

This list segment predicate rules out cycles. However, cycles can be described using two list predicates, or a points-to and a list segment. For example, the following assertion is validated in the pictured model.

$$ls(x, y) * ls(y, x)$$



These partial lists are sometimes used in the specifications of data structures, such as queues. In other cases, they are needed to state the internal invariants of an algorithm, even when the pre and post of a program use total lists only (total lists $list(E)$ can be regarded as abbreviations for segments $ls(E, nil)$). Here is a program from the SMALLFOOT tool [7] which exemplifies this point.

```
list_append(x,y) PRE: [list(x) * list(y)] {
  local t;
  if (x == NULL) {
    x = y;
  } else {
    t = x; n = t->tl;
    while (n != NULL) [ls(x,t) * t |->  n * list(n)] {
      t = n;
      n = t->tl;
    }
    t->tl = y;
  } /* ls(x,t) * t |-> y * list(y) */
} POST: [list(x)]
```

This program, which appends two lists by walking down one and then swinging its last pointer to the other, uses a partial list in its loop invariant, even though partial lists are not needed in the overall procedure spec. In proving this program an important point is how one gets from the last statement to the postcondition. A comment near the end of the program shows an assertion describing what is known at that program point, and we need to show that it implies the post to verify the program. That is, we need to show an implication

$$ls(x,t) * t{\mapsto}y * list(y) \implies list(x).$$

This implication may seem unremarkable, but it is at this point that automatic tools must begin to do something clever. For, consider how you, the human, would convince yourself of the truth of this implication. If it were me, I would look at the semantics and prove this fact by induction on the length of the list from x to t. But if we were to include such reasoning in an automatic tool, we had better try to do so in an inductionless way, else our tool will need to search for induction hypotheses (which is hard to make automatic).

Exercise 4 *There are other definitions of list segments that have been used. Here is one, the 'imprecise list segment'.*

$$\begin{aligned} ils(E,F) \iff & (E = F \ \wedge \ emp) \\ & \vee \ \exists y. E{\mapsto}y \ * \ ils(y,F) \end{aligned}$$

Q1. *What is a heap that distinguishes* $ls(10,10)$ *and* $ils(10,10)$ *?*
Q2. *What distinguishes* $ls(10,11)$ *and* $ils(10,11)$ *?*
Q3. *Prove or disprove the following laws (do your proof by working in the semantics)*

$$ls(x,y) * ls(y,z) \implies ls(x,z) \quad ???$$
$$ils(x,y) * ils(y,z) \implies ils(x,z) \; ???$$

Q4. *Suppose we want to write a procedure that frees all the cells in a list segment. For which of* ils *or* ls *can you do it? If you cannot do it for one of them, why not? That is, we are asking for terminating programs satisfying*

$$\{ls(x,y)\}\ \texttt{delete\_ls}(x,y)\ \{emp\}$$
$$\{ils(x,y)\}\ \texttt{delete\_ils}(x,y)\ \{emp\}$$

(I have not given you the definition of the truth of pre/post specs yet, but you should be able to answer this question anyhow.)

Exercise 5 *Give a definition $ls(E,F,\sigma)$ of a predicate describing a linked list from E to F that contains the sequence σ in data fields. Write specification of programs that insert and delete elements from* sorted *linked lists, where σ is sorted according to an ordering. Give at least the loop invariants for these programs (write iterative versions). Attempt a proof-by-execution type argument as well.*

Exercise 6 *The predicate tree(E,τ) we used above considers τ as an s-expression, where the values are only at the leaves of the tree and not at internal nodes. Often, one wants to use a data structure for mathematical trees including data at internal nodes, and one way to describe these is with the set equation*

$$\texttt{Mtree} \;=\; \{nil\} + (\texttt{Mtree} \times \texttt{Atom} \times \texttt{Mtree})$$

In this sort of tree, nil is the empty tree and the leaves of a non-empty tree are those 3-tuples that have nil in their first and third components.

Give an inductive definition of a predicate tree(E,τ), for $\tau \in$ Mtree. Hint: use a points-to assertion of the form $E{\mapsto}[l\colon x, d\colon y, r\colon z]$ where d refers to the data, or atom, field. Define the CopyTree *and* DispTree *procedures for this sort of tree, and give proof-by-execution style arguments for their correctness.*

3.3. Proof Rules for Programs.

The proof rules for procedure calls, sequencing, conditionals and loops are the same as in standard Hoare logic [42,43]. Here I concentrate on the rules for primitive commands for accessing the heap, and the surrounding rules, called the 'structural rules'. (If you are unfamiliar with Hoare logic probably the best way to learn is to go directly to the early sources, such as [42,44,43,37,27], which are pleasantly simple and easy to read.)

We will use the following abbreviations:

$$\begin{array}{lcll} E{\mapsto}F_0,...,F_n & \triangleq & (E{\mapsto}F_0) * \cdots * (E+n{\mapsto}F_n) & \\ E \doteq F & \triangleq & (E=F) \wedge emp & \\ E{\mapsto}- & \triangleq & \exists y.E{\mapsto}y & (y \notin \text{Free}(E)) \end{array}$$

where Free(E) is the set of free variables in E.

We have axioms for each of four atomic commands. In the axioms x, m, n are assumed to be distinct variables.

THE SMALL AXIOMS

$$\{E{\mapsto}{-}\}\,[E] := F\,\{E{\mapsto}F\}$$

$$\{E{\mapsto}{-}\}\,\texttt{free}(E)\,\{emp\}$$

$$\{x \doteq m\}x := \texttt{cons}(E_1, ..., E_k)\{x{\mapsto}E_1[m/x], ..., E_k[m/x]\,\}$$

$$\{x \doteq n\}\,x := E\,\{x \doteq (E[n/x])\}$$

$$\{E{\mapsto}n \wedge x = m\}\,x := [E]\,\{x = n \wedge E[m/x]{\mapsto}n\}$$

The first small axiom just says that if E points to something beforehand (so it is in the domain of the heaplet), then it points to F afterwards, and it says this for a small portion of the state (heaplet) in which E is the only active cell. This corresponds to the operational idea of $[E] := F$ as a command that stores the value of F at address E in the heap. The other commands have similar explanations. Notice that each axiom mentions only the cells accessed or allocated: the axioms talk only about footprints, and not the entire global program state. We only get fixed-length allocation from $x := \texttt{cons}(E_1, ..., E_k)$. but it is also possible to axiomatize a command $x := \texttt{alloc}(E)$ that allocates a block of length E.

Notice that our axioms allow us to free any cell that is allocated, even from the middle of a block given by `cons`. This is different from the situation in the C programming language, where you are only supposed to free an entire block that has been allocated by `malloc`(). An elegant treatment of this problem has been given using predicate variables in [66].

The assignment statement $x := E$ is for a variable x and heap-independent arithmetic expression E. Thus, this statement accesses and alters the store, but not the heap. It is the assignment statement considered by Hoare in his original system [42]. In contrast, the form $[E] := F$ alters the heap but not the store.

To go along with the small axioms we have additional surrounding rules.

THE STRUCTURAL RULES

Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \text{Modifies}(C) \cap \text{Free}(R) = \emptyset$$

Auxiliary Variable Elimination

$$\frac{\{P\}\,C\,\{Q\}}{\{\exists x.P\}\,C\,\{\exists x.Q\}}\ x \notin \text{Free}(C)$$

Variable Substitution

$$\frac{\{P\}\,C\,\{Q\}}{(\{P\}\,C\,\{Q\})[E_1/x_1, ..., E_k/x_k]}\ \begin{array}{l}\{x_1, ..., x_k\} \supseteq \text{Free}(P, C, Q), \text{ and}\\ x_i \in \text{Modifies}(C) \text{ implies}\\ E_i \text{ is a variable not free in any other } E_j\end{array}$$

Rule of Consequence

$$\frac{P' \Rightarrow P \qquad \{P\}\,C\,\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\,C\,\{Q'\}}$$

Modifies(C) here is the set of variables that are assigned to within C. The Modifies set of each of $x := \mathtt{cons}(E_1, ..., E_k)$, $x := E$ and $x := [E]$ is $\{x\}$, while for $\mathtt{free}(E)$ and $[E] := F$ it is empty. Note that the Modifies set only tracks potential alterations to the store, and says nothing about the heap cells that might be modified.

Two of these rules we have already seen: the frame and consequence rules. The others are rules that have been considered in the Hoare logic literature. This collection of axioms and rules is complete in the sense that all true Hoare triples for the basic statements can be derived them (assuming an oracle for implication in the consequence rule). A proof of this fact is contained in Hongseok Yang's thesis [84] (in fact, Yang chose the existential and substitution structural rules precisely in order to make the small axioms complete).

This presentation of the proof system above is from [61]. In his LICS'02 paper [74] Reynolds gives a comprehensive description of a variety of axioms, in local (small) and global and backwards forms, for the various atomic commands. The additional laws are important because one prefers to have derived laws that can be applied at once in common situations without going back to the small axioms every time and invoking the structural rules extensively.

For example, it follows from Yang's results that Hoare's assignment axiom

$$\{P[E/x]\}\, x := E\, \{P\}$$

can be derived, where $x := E$ is the assignment statement that is heap independent. One can also derive Floyd's forwards-running axiom [36]

$$\{P\}\, x := E\, \{\exists x'.\, x = E[x'/x] \wedge P[x'/x]\}$$

where the existentially quantified variable x' (which must be fresh) provides a way to talk about x's value in the pre-state. The symbolic execution rules in SMALLFOOT and related tools use forwards-running rules of this variety (Section 4.2).

As an example derived rule for a heap-accessing command, with the frame rule and auxiliary variable elimination one can obtain an axiom from [73]

$$\{\exists x_1, \cdots, x_n.\, (E{\mapsto}{-}) \ *\ R\}\, [E] := F\, \{\exists x_1, \cdots, x_n.\, (E{\mapsto}F) \ *\ R\}$$
(where $x_1, ..., x_n \notin \mathrm{Free}(E, F)$.)

that will be useful when defining symbolic execution later.

The $-\!\!*$ connective has not often been used in proofs of particular programs (some examples are in [84,66,32]). But it is a handy thing to have when doing metatheoretic reasoning about a system [47,86,67,17]. The de Morgan dual $\neg(P -\!\!* \neg Q)$ (called 'septraction' in [81]) has played a central role in the formulation of a logic marrying separation logic and the rely-guarantee method for concurrent programs, and it is used in an automated tool based on the marriage logic [19].

An example metatheoretic use of $-\!\!*$ is in proving completeness results. For example, the following derivation

$$\dfrac{\dfrac{\{E{\mapsto}{-}\}\, [E] := F\, \{E{\mapsto}F\}}{\{(E{\mapsto}{-}) * ((E{\mapsto}F) -\!\!* Q)\}\, [E] := F\, \{(E{\mapsto}F) * ((E{\mapsto}F) -\!\!* Q)\}}\ \text{Frame}}{\{(E{\mapsto}{-}) * ((E{\mapsto}F) -\!\!* Q)\}\, [E] := F\, \{Q\}}\ \text{Consequence}$$

gives us general precondition for any postcondition Q, and this is key to showing that the small axiom for mutation is not missing anything.

Exercise 7 *Go back over the proof-by-execution style arguments you gave in the previous exercises, and convince yourself that you can formalize them in the proof system given in this section. You will probably want to use derived laws for each of the basic program statements. In such proofs you get to use the semantics as an oracle when deciding the implication statements in the rule of consequence.*

Exercise 8 *Formulate an operational semantics of $[E] := F$ in terms of stores and heaps. I.e., say when $[E] := F, s, h$ evaluates to s, h'. Don't use separation logic in this formulation.*

For a given set of states Q, say what it means to be the weakest precondition of $[E] := F$ with postcondition Q.

Finally, prove (in math, not in logic) that $(E \mapsto -) * ((E \mapsto F) -\!\!* Q)$ expresses the weakest precondtion.

Aside: On Variable Conditions and store-vs-environment. In Section 2 I skated over the issue of Modifies sets, not mentioning them when introducing the frame rule. Conditions involving Modifies sets are inelegant, and are all the more irritating because they arise from a deliberate punning in Hoare logic between store and environment, which is uncommon in programming languages.

At the birth of program semantics, in one of the founding papers of the field, Strachey advised to distinguish the environment (association of variables to values), which can be altered by variable binding in a way that obeys a stack discipline, from the store (association of values to locations), which can be mutated by assignment [78]. Programming languages from C to ML to Java observe Strachey's distinction. The benefit from conflating the ideas is that one gets beautifully simple specifications and proofs of simple example programs in Hoare logic, or in Dijkstra's wp calculus: it leads to neater (shorter) examples to illustrate ideas, so in that sense the pun was worth it. I persist with the pun in these lectures for the same reason. But, researchers are more and more avoiding conflating store and environment in working out their theories, and proof tools for C and Java do not need to worry about Modifies sets. See [65] for further discussion.

3.4. Tight Specifications

The issues related to frame axioms that we discussed in Section 2.2 go a long way back, to the beginning work on logic in AI [57]. Fundamentally, the reason why AI issues are relevant to program logic is just that programmers describe their code in a way that corresponds to a commonsense reading of specifications, where much is left unsaid. Practically, if we do not employ *some kind* of solution to the AI problems, then specifications quickly become extremely complicated [11].

Some people think that the real problem is in a way negative in nature, is to avoid writing nasty frame axioms like we did in the 'back in the day' discussion in Section 2.2. Other people think the problem is just to have succinct specs, however one gets them. I have always thought both of these, succinct specs and avoiding writing frame axioms, should be a consequence of a solution, but are not themselves the problem. My

approach to this issue has always been to embrace the 'commonsense reasoning' aspect first, and for this the idea of a 'tight specification' is crucial: the idea is that if you don't say that something changes, then it doesn't. For example, if you say that a robot moves block A from position 1 to position 2, then the commonsense reading is that you are implicitly saying as well that this action does not change the position of a separate block B (unless, perhaps, block B is on top of block A). Programmers' informal descriptions of their code are similar. In the java.util List interface the description of the copy method is just that it 'copies all of the elements from one list into another'. There is no mention of frames in the description: the description carries the understanding that the frame remains unchanged. The need to describe the frames explicitly in some formalisms is just an artefact, which programmers do not find necessary when talking about their code (because of this commonsense reasoning that they employ).

Be that as it may, formalization of the notion of tight specification proved to be surprisingly difficult, and in AI there have been many elaborate theories advanced to try to capture this notion – circumscription, default logic, nonmonotonic logic, and more – far too many to give a proper account of here. Without claiming to be able to solve the general AI problem, this section explains how an old idea in program logic, when connected to the principle of local reasoning (that you only need to talk about the cells a program touches), gives a powerful and yet very simple approach to tight specifications.

The old idea is of *fault-avoiding specifications*. To formulate this, let us suppose that we have a semantics of commands where $C, \sigma \rightsquigarrow^* \sigma'$ indicates that there is a terminating computation of command C from state σ to state σ'. In the RAM model σ can be a pair of a store and a heap, but the notion can be formulated at a more general level than this particular model. Additionally, we require a judgement form $C, \sigma \rightsquigarrow^* \texttt{fault}$. In the RAM model, `fault` can be taken to indicate a memory fault: a dereferencing of a dangling pointer or a double-free. Again, more generally, `fault` can be used for other sorts of errors.

Here, then, is a fault-avoiding semantics of triples, where for generality we are viewing the preconditions and postconditions as sets of states rather than as formulae written in some particular assertion language.

Faut-Avoiding Partial Correctness

$\{A\}\, C\, \{B\}$ holds iff $\forall \sigma \in A$

1. no faults: $C, \sigma \not\rightsquigarrow^* \texttt{fault}$
2. partial correctness: $C, \sigma \rightsquigarrow^* \sigma'$ implies $\sigma' \in B$.

The 'no faults' clause is a reasonable thing to have as a way for proven programs to avoid errors, and was used as far back as Hoare and Wirth's axiomatic semantics of Pascal in 1973 [45]. Notice that the small axioms given above are already in a form compatible with the fault-avoiding semantics. For instance, in the axiom

$$\{E \mapsto -\}\, [E] := F\, \{E \mapsto F\}$$

the $E \mapsto -$ in the precondition ensures that E is not a dangling pointer, and so $[E] := F$ will not memory fault.

Remarkably, besides ensuring that well-specified programs avoid certain errors, it was realized much later [47] that the fault-avoiding interpretation gives us an approach to tight specifications. The key point is a consequence of the 'no faults' clause: touching

any cells not known to be allocated in the precondition falsifies the triple, so *any cells not 'mentioned' (known to be allocated) in the pre will remain unchanged*. To see why, suppose I tell you

$$\{10 \mapsto -\}\, C\, \{10 \mapsto 25\}$$

but I don't tell you what C is. Then I claim C cannot change location 11 if it happens to be allocated in the pre-state (when 10 is also allocated). For, if C changed location 11, it would have to access location 11, and this would lead to `fault` when starting in a state where 10 is allocated and 11 is not. That would falsify the triple (no error clause). As a consequence we obtain that

$$\{10{\mapsto} - {*} 11{\mapsto}4\}\, C\, \{10{\mapsto}25 * 11{\mapsto}4\}$$

should hold.

This reasoning is the basis for the frame rule. But the semantic fact that location 11 doesn't change is completely independent of separation logic. In fact, we could state a similar conclusion without mentioning $*$ at all

$$\{10{\hookrightarrow} - \wedge 11{\hookrightarrow}4\}\, C\, \{10{\hookrightarrow}25 \wedge 11{\hookrightarrow}4\}$$

Separation logic, and the frame rule, only give you a *convenient* way to exploit the tightness (that things don't change if you don't mention them) in the fault-avoiding interpretation. This tightness phenomenon is in a sense at a more fundamental level, prior to logic.

It is useful to consider that for this approach to tight specifications to work `fault` does not literally need to indicate memory fault, and it is not necessary to use a low-level memory model. For instance, we can put a notion of 'accesses' or 'ownership' in a model, and then when the program strays beyond what is owned we declare a specification false: then, the same argument as above lets us conclude that certain cells do not change. This is the idea used in implicit dynamic frames [77], and in separation logics for garbage-collected languages like Java where there are no memory faults (e.g., [66]). Alternate approaches may be found in [4,3,49].

I have tried to explain the basis for tight specifications above in a semi-formal way. But, the reader might have noticed that there were some unstated assumptions behind my arguments. One can imagine mathematical relations on states and `fault` that contradict our conclusion that 11 will remain unchanged. One such relation is as follows: if the input heap is a singleton, it sets the contents of the only allocated location to be 25, and otherwise sets all allocated locations in the input heap to have contents 50. This is not a program that you can write in C, but it shows that that there are *locality properties* of the semantics of programs at work behind the tight interpretation of triples, and it is important theoretically to set these conditions down precisely; see [86,18,72].

Exercise 9 *Without saying what the commands C are, and ignoring the store component (i.e., think about heap only), formulate sufficient conditions on the relations $C, \sigma \rightsquigarrow^* \sigma'$ and $C, \sigma \rightsquigarrow^*$* `fault` *which make the frame rule valid according to fault-avoiding partial correctness. Give a proof of the validity of the frame rule from these conditions.*

Are your conditions necessary as well as sufficient?

4. Symbolic Heaps, Symbolic Execution and Abstract Interpretation

In the previous sections I emphasized an informal view of program proof as a form of symbolic execution. That is the view implemented in a number of verification and analysis tools based on separation logic, beginning with SMALLFOOT [7]. In this section I describe the foundations of this approach, and give a short introduction to its extension to program analysis (where abstraction is used to calculate loop invariants).

4.1. Symbolic Heaps

When designing an automatic program verification tool there are almost always compromises to be made, forced by the constraints of recursive undecidability of so many questions about logics and programs. The first tools based on separation logic chose to restrict attention to a certain format of assertions which made three tasks easier than they might otherwise have been: symbolic execution, entailment checking, and frame inference.

Symbolic heaps [6,30] are formulae of the form

$$\exists \vec{X}.(P_1 \wedge \cdots P_n) \wedge (S_1 * \cdots * S_m)$$

where the P_i and S_j are primitive pure and spatial predicates, and $\vec{X}$ is a vector of logical variables (variables not used in programs). We understand the nullary conjunction of P_i's as $true$ and the nullary $*$-conjunction of S_i's as emp. The special form of symbolic heaps does not allow, for instance, nesting of $*$ and $\wedge$, or boolean negation $\neg$ around $*$, or the separating implication $-\!\!*$. This special form was chosen, originally, to match the usage of separation logic in a number of by-hand proofs that had been done. The form does not cover all proofs, such as Yang's proof of the Schorr-Waite algorithm [84], so there are immediately-known limitations.

The grammar for symbolic heaps can be instantiated with different sets of basic pure and spatial predicates. Pure formulae are heap-independent, and describe properties of variables only, where the spatial formulae specify properties of the heap. One instantiation is as follows

SIMPLE LISTS INSTANTIATION

$$\begin{array}{l} P ::= E{=}E \mid E{\neq}E \mid \\ S ::= E{\mapsto}E \mid lsne(E,E) \mid true \end{array}$$

Expressions E include program variables x, logical variables X, or constants κ (e.g., nil). Here, the *points-to* predicate $x{\mapsto}y$ denotes a heap with a single allocated cell at address x with content y, and $lsne(x,y)$ denotes a nonempty list segment from x to y. This is the list segment predicate used in the paper [30] on program analysis, which described the analysis that we call BABY SPACEINVADER (the grown up version is represented in [5,85]). In contrast, the 'possibly empty list segments' predicate ls we described before was used in SMALLFOOT. It turns out that there is no one best predicate. In a practical program analysis tool, it is helpful to keep both forms of segment ls and $lsne$ in the assertion language, even though they can be expressed in terms of one another and disjunction: keeping distinct predicates for empty and nonempty list segments in a language provides a means to help limit the number of disjuncts that need to be considered by the

program analysis, a key issue in dealing with state-space explosion [85]. Some tools even prefer the imprecise list segment predicate ils from Exercise 4, to make the abstraction or widening step in an abstract interpreter easier to design.

There are many other instantiations that one can consider. One instantiation keeps P the same and replaces simple linked-lists by a higher-order variant which allows lists to be nested [5]. Varieties of trees, possibly with back pointers, have been considered [20]. As have predicates that track arithmetic information or the contents of data structures [8,51,80]. Very complicated abstract domains are needed to cope with the complicated data structures occurring in real-world programs. But in these lectures we will stick with the simple lists, for simplicity of presentation.

CONVENTIONS. We observe the following conventions. In writing a symbolic heap we omit the leading $\exists \vec{X}$, understanding that the logical variables X are implicitly existentially quantified. Also, we overload the $*$ operator, so that it also works for entire symbolic heaps H and not only the components.

$$\begin{array}{l} \big((P_1 \wedge \cdots P_n) \wedge (S_1 * \cdots * S_m)\big) * \big((P'_1 \wedge \cdots P'_{n'}) \wedge (S'_1 * \cdots * S'_{m'})\big) \\ \triangleq (P_1 \wedge \cdots P_n \wedge P'_1 \wedge \cdots P'_{n'}) \wedge (S_1 * \cdots * S_m * S'_1 * \cdots * S'_{m'}) \end{array}$$

4.2. Symbolic Execution

The symbolic execution semantics $H, A \Longrightarrow H'$ takes a symbolic heap H and an atomic command, and transforms it into an output symbolic heap or `fault`. In these rules we require that the logical variables X, Y be fresh.

SYMBOLIC EXECUTION RULES

$$\begin{array}{lll} H & x := E & \Longrightarrow x = E[X/x] \wedge H[X/x] \\ H * E{\mapsto}F & x := [E] & \Longrightarrow x = F[X/x] \wedge H * E{\mapsto}F)[X/x] \\ H * E{\mapsto}F & [E] := G & \Longrightarrow H * E{\mapsto}G \\ H & x := \mathtt{cons}(-) & \Longrightarrow H[X/x] * x{\mapsto}X \\ H * E{\mapsto}F & \mathbf{free}(E) & \Longrightarrow H \end{array}$$

With the convention that the logical variables are implicitly existentially quantified, the first rule is just a restating of Floyd's axiom for assignment. The other rules can be obtained from the small axioms of Section 3 by applications of the structural rules.

The rules for $x := [E]$, $[E] := G$ and $[E] := G$ all assume that we have $E{\mapsto}F$ explicitly in the precondition. In some cases, this knowledge that E points to something will be somewhat less explicit, as in the symbolic heap $E = E' \wedge E'{\mapsto}F$. Then, a simple amount of logical reasoning can convert this formula to the equivalent form $E = E' \wedge E{\mapsto}F$, which is now ready for an execution step. In another case, $lsne(E, F)$, we might have to unroll the inductive definition to reveal the $\mapsto$. In general, for any of these heap-accessing forms, we need to massage a symbolic heap to 'make $E{\mapsto}$ explicit'. Here are sample rules for doing this massaging.

REARRANGEMENT RULES

$$A(E) \quad ::= [E] := G \mid [E] := G \mid [E] := G$$
$$P(E,F) ::= E{\mapsto}F \mid lsne(E,F)$$

$$\frac{H_0 * P(E,G),\ A(E) \implies H_1}{H_0 * P(F,G),\ A(E) \implies H_1}\ H_0 \vdash E = F$$

$$\frac{H_0 * E{\mapsto}X * lsne(X,G),\ A(E) \implies H_1}{H_0 * lsne(E,G),\ A(E) \implies H_1}$$

$$\frac{H_0 * E{\mapsto}F,\ A(E) \implies H_1}{H_0 * lsne(E,F),\ A(E) \implies H_1}$$

$$\frac{H \not\vdash \texttt{Allocated}(E)}{H,\ A(E) \implies \texttt{fault}}$$

In these rules we referred to a notion of entailment $\vdash$ that will be discussed in Section 4.4. $\texttt{Allocated}(E)$ can be represented by the assertion $E{\mapsto}X * true$ where X is fresh.

[Aside: This rearrangement notion is related to the *partial concretization* operation used in shape analysis [75,76], where one concretizes just enough of an abstract value so that the concrete program semantics can be applied. Rearrangement is also a special case of the concept of frame inference discussed later in Section 4.5.]

It is a good idea, and good for practice, for you to become familiar with the different variations on list segments.

Exercise 10 *Without looking at any of the papers referenced in this section...*

1. *Give an inductive definition of the predicate for necessarily non-empty list segments $lsne$, corresponding to the rearrangement rules above.*
2. *Give rearrangement rules that would be appropriate for the earlier definition of possibly empty list segments, ls.*
3. *Consider the formulae*

   $$ls(x,y) * ls(x,z) \qquad \textit{and} \qquad lsne(x,y) * lsne(x,z)$$

   Is either formula satisfiable? Might this affect any of the steps in symbolic execution?
4. *(Advanced) Write an inductive definition for a predicate that describes doubly-linked list segments. It should have four arguments. Be careful about the base case.*
   Write rearrangement rules for this doubly-linked list predicate.

4.3. Recipe for Cooking a Verifier

Using symbolic execution, it is possible to construct an automatic verification tool as follows. The input to the tool is a while program with heap-manipulating primitives as in the previous section. The program must be annotated with loop invariants and a pre-

condition and a postcondition. The SMALLFOOT `list_append` program from Section 3.2 is of this form. The usual rules of Hoare logic for loops and conditionals then enable us to chop up the correctness of such a program into a number of questions of the form $\{H\}\, c_1; ...; c_n\, \{H'\}$ for atomic commands c_i. If we can verify that each of these straightline specifications $\{H\}\, c_1; ...; c_n\, \{H'\}$ is true then we can conclude that the beginning program satisfies its pre/post spec.

In many verification tools the straightline specs $\{H\}\, c_1; ...; c_n\, \{H'\}$ are decided by using a weakest preconditon calculation to obtain a formula $wp(c_1; ...; c_n, H')$ and then asking a theorem prover if $H \Rightarrow wp(c_1; ...; c_n, H')$. Or, a strongest postcondition could be used.

The approach used often in separation logic tools is something like the strongest post calculation, except that lots of subsidiary calls are made to a theorem prover along the way. To decide $\{H\}\, c_1; ...; c_n\, \{H'\}$ we first ask the theorem prover if H is inconsistent. If it is, we are done (the spec is true). Second, if $c_1, ..., c_n$ is the empty sequence we ask a prover if $H \vdash H'$. Otherwise, we apply symbolic execution the first statement c_1, and this gives us `fault` or several symbolic heaps (several because there is nondeterminism in rearrangement, and since some basic commands in a real language might have disjunctions in their postconditions. (Why might `malloc()` have a disjunction as its post?). A theorem prover is consulted in the rearrangement phase here. If `fault` resulted from symbolic execution then were are done (the spec is false). If, instead, execution yields several heaps $H_1, ..., H_m$ then we return the conjunction of the smaller questions $\{H_i\}\, ..; c_n\, \{H'\}$. This last case essentially relies on the rule

$$\frac{\{P_1\}\, C\, \{Q\} \quad \cdots \quad \{P_n\}\, C\, \{Q\}}{\{P_1 \vee \cdots \vee P_n\}\, C\, \{Q\}}$$

of Hoare logic.

This verification strategy relies on having a theorem prover to answer entailment questions $H \vdash H'$. A straightforward embedding of separation logic into a classical logic, where one writes the semantics in the target logic (e.g., '$\exists \sigma_1 \sigma_2 . \sigma = \sigma_1 \bullet \sigma_2 ...$'), has not yet yielded an effective prover, because it introduces existential quantifiers to give the semantics of $*$. Therefore, proof tools for separation logic has used dedicated proof procedures, built from the proof rules of the logic. (Work is underway on more nuanced interpretations into existing provers that do more than a direct semantic embedding.)

4.4. Proof Procedures for Entailment

An approach to proving symbolic heaps was pioneered by Josh Berdine and Cristiano Calcagno [6]. Their approach revolves around proof rules for abstraction and subtraction. A sample abstraction rule is

$$ls(x, t) * list(t) \;\vdash\; list(x)$$

where the subtraction rule is

$$\frac{Q_1 \vdash Q_2}{Q_1 * S \vdash Q_2 * S}$$

Their basic idea is to try to reduce an entailment to an axiom $B \wedge emp \vdash true \wedge emp$ by successively applying abstraction rules, and Subtracting when possible. The basic idea can be appreciated by considering two examples.

First, a successful example:

$\overset{\cdot\cdot}{\smile}$	
$emp \ \vdash \ emp$	Axiom!
$list(x) \ \vdash \ list(x)$	Subtract
$ls(x,t) * list(t) \ \vdash \ list(x)$	Abstract (Inductive)
$ls(x,t) * t{\mapsto}y * list(y) \ \vdash \ list(x)$	Abstract (Roll)

The entailment on the bottom is the one we needed to prove at the end of the `list_append` procedure from Section 3.2. The first step, going upward, is a simple rolling up of an inductive definition. The second step is more serious: it is one that we would use induction in the metalanguage to justify. We then get to a position where we can apply the subtraction rule, and this gets us back to a basic axiom.

For an unsuccessful example

$\overset{\cdot\cdot}{\frown}$	
$list(y) \ \vdash \ emp$	Junk: Not Axiom!
$list(x) * list(y) \ \vdash \ list(x)$	Subtract
$ls(x,t) * t{\mapsto}nil * list(y) \ \vdash \ list(x)$	Abstract (Inductive)

The last line is an entailment that SMALLFOOT would attempt to prove if the statement `t->tl = y` at the end of the `list_append` program were replaced by `t->tl = nil`. There we do an abstraction followed by a subtraction and we get to a position where we cannot reduce further. Rightly, we cannot prove this entailment.

The detailed design and theoretical analysis of a proof theory based on these ideas is nontrivial. For the specific case of singly-linked list segments, Berdine and Calcagno were able to formulate a complete and terminating proof theory. There is no space to go into all the details of their theory, but it is worth listing their abstraction rules, presented here as entailments.

Rolling

$$emp \vdash ls(E,E)$$

$$E_1 \neq E_3 \wedge E_1{\mapsto}E_2 * ls(E_2,E_3) \vdash ls(E_1,E_3)$$

Induction Avoidance

$$ls(E_1,E_2) * ls(E_2,nil) \vdash ls(E_1,nil)$$

$$ls(E_1,E_2) * E_2{\mapsto}nil \vdash ls(E_1,nil)$$

$$ls(E_1,E_2) * ls(E_2,E_3) * E_3{\mapsto}E_4 \vdash ls(E_1,E_3) * E_3{\mapsto}E_4$$

$$E_3 \neq E_4 \wedge ls(E_1,E_2) * ls(E_2,E_3) * ls(E_3,E_4) \vdash \quad ls(E_1,E_3) * ls(E_3,E_4)$$

The remarkable thing about these abstraction rules is not that they are sound, but in a sense complete: any true fact about list segments and points-to facts that can be expressed in symbolic heap form can be proven using these axioms, without appealing to an explicit induction axiom or rule. The Berdine/Calcagno proof theory works by using these rules on the left (in effect employing a special case of the Cut rule of sequent calculus). It has other rules as well, such as for inferring $x \neq nil$ from $x \mapsto -$: at every stage, their decision procedure records as many pure disequalities as possible on the left, and it substitutes out all equalities, getting to a kind of normal form. It is this normal form that makes the subtraction rule complete (a two-way inference rule).

Note: in this subsection we have gone back to the ls rather than $lsne$ predicate, as Berdine and Calcagno formulated their rules for ls. In fact, it is easier to design a complete proof theory for $lsne$ rather than ls. It is also relatively easy (and was folklore knowledge) to see that entailment for the $lsne$ symbolic heaps can be decided in polytime, but the question for the ls remained open until a recent paper which showed the entailment is indeed in polytime in the original problem [23]. (The reason for subtlety in this question is related to question 3 of Exercise 10; you might go back there and wonder about it.)

I like to call this approach of combining abstraction and subtraction rules the 'crunch, crunch' method. It works by taking a sequent $H \vdash H'$ and applying abstraction and subtraction rules to crunch it down to a smaller size by removing $*$-conjuncts, until you get emp as the spatial part on one side or the other of $\vdash$. If you have emp on only one side, you have a failed proof. If you have other pure facts, of the form $\Pi \wedge emp \vdash \Pi' \wedge emp$ you can then ask a straight classical logic question $\Pi \vdash \Pi'$. The final check, $\Pi \vdash \Pi'$, is a place where one could call an external theorem prover, say for a decidable theory such as linear arithmetic, and that is all the more useful when the pure part can contain a richer variety of assertions than in the simple fragment considered in this section. Indeed, there have been a number of provers for separation logic developed that use variations on this 'crunch, crunch' approach together with an external classical logic solver, including [13,68] and the provers inside VERIFAST [48], JSTAR [31], HIP [60] and SLAYER [9].

4.5. Frame Inference

Entailment is a standard problem for verifiers to face. In work applying separation logic, a pivotal development has been identification of the notion of *frame inference*, which is an extension of the entailment question:

> In a frame inference question of the form
>
> $$A \vdash B * \texttt{?frame}$$
>
> the task is, given A and B, to find a formula `?frame` which makes the entailment valid.

Frame inference gives a way to find the 'leftover' portions of heap needed to automatically apply the frame rule in program proofs. 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.

A first solution to frame inference was sketched in [6] and implemented in the SMALLFOOT tool. The SMALLFOOT approach works by using information from failed proofs of the standard entailment question $A \vdash B$. Essentially, a failed proof of the form

$$\begin{array}{c} F \vdash \mathit{emp} \\ \vdots \\ A \vdash B \end{array}$$

tells us that F is a frame. For, from such a failed proof we can form a proof

$$\begin{array}{c} F \vdash F \\ F \vdash \mathit{emp} * F \\ \vdots \\ A \vdash B * F \end{array}$$

by tacking $*F$ on the right everywhere in the failed proof. So, the frame inferring procedure is to go upwards using the 'crunch, crunch' proof search method until you can go no further: if your attempted proof is of the form indicated above, it can tell you a frame. (Dealing with multiple branches in proofs requires some more subtlety than this description indicates.)

Frame inference is a workhorse of separation logic verification tools. As you can imagine from the discussion surrounding `Disptree` in Section 2, it is used at procedure call sites to identify the part of a symbolic heap that is not not touched by a procedure. Interprocedural program analysis tools typically use (often incomplete) implementations of frame inference for reasoning with 'procedure summaries' [39,59]. In SMALLFOOT, proof rules for critical regions in concurrent programs are verified using little phantom procedures (called 'specification statements') with specs of the form $\{emp\} - \{R\}$ and $\{R\} - \{emp\}$ for materializing and annihilating portions of storage protected by a lock. Indeed, if one has a good enough frame inference capacity, then symbolic execution can be seen to be a special case of a more general scheme, where basic commands are treated as specification statements (the small axioms), and frame inference is used in place of the special concept of rearrangement. SMALLFOOT and SPACEINVADER did not follow this idealistic approach, preferring to optimize for the common case of the basic statements, but the more recent JSTAR and SLAYER are examples of tools that call a frame inferring theorem prover at every step of symbolic execution [31,9].

4.6. A Taste of Abstract Interpretation

Beginning in 2006 [30,52], a significant amount of work has been done on the use of separation logic in automatic program analysis. There are a number of academic tools, including SPACEINVADER [5,85], THOR [53], XISA [20], FORRESTER [41], PREDATOR [33], SMALLFOOTRG [19], HEAP-HOP [83] and JSTAR [31], and the industrial tools INFER from Monoidics [16] and SLAYER from Microsoft [9]. The tools in this area are a new breed of *shape analysis*, which attempt to discover the shapes of data structures (e.g., whether a list is cyclic or acyclic) in a program [75]. These tools cannot prove functional correctness, but can be applied to code in the thousands or even millions of LOC [85,17].

The general context for this use of separation logic concerns the relation between program logic and program analysis. It has been known since the work of the Cousots in the 1970s [24,25] that concepts from Hoare logic and static program analysis are related. In principle, static analysis can be used to calculate loop invariants and procedure specifications via fixed-point computations, thereby lessening annotation burden. There is a price to pay in that trying to be completely automatic in this way almost forces one to step away from the ideal of proving full functional correctness.

While the relation between analysis and verification has been long known in principle, the last decade has seen a surge of interest in verification-by-static-analysis, with practical demonstrations of its potential such as in SLAM's application of proof technology to Microsoft device drivers [2] and ASTRÉE's proof of the absence of run-time errors in Airbus code [26]. Separation logic enters the picture because these practical tools for verification-oriented static analysis ignore pointer-based data structures, or use coarse models that are insufficient to prove basic properties of them; e.g., SLAM *assumes* memory safety, and ASTRÉE works only on input programs that do not use dynamic allocation. Similar remarks apply to other tools such as BLAST, Magic and others. Data structures present a significant problem in verification-oriented program analysis, and that is the point that the separation logic program analyses are trying to address.

This section illustrates the ideas in the abstractions used in separation logic program analyzers. To begin, suppose you were to continually symbolically execute a program with a while loop. You collect sets of formulae (abstract states) at program points, and generate new ones by symbolically executing program statements. The immediate problem is that you would go on generating symbolic heaps on loop iterations, and the process could diverge: you would never stop generating new symbolic heaps. The most basic idea of program analysis is to use abstraction, the losing of information, to ensure that such a process terminates.

Consider the following program that creates a linked list of indeterminate length.

$\{Pre : emp\}$
x := *nil*;
while $(nondet())\{$
    y := `cons`(-);
    y→tl := x;
    x := y;
$\}$

Suppose we start symbolically executing the program from pre-state emp. On the first iteration, at the program point immediately inside the loop, we will certainly have that $x = \mathit{nil} \wedge emp$ is true, so let us record this in

Loop Invariant so far (1st iteration)
$x = nil \wedge emp.$

Now, if we go around the loop once more, then it is clear that $x \mapsto nil$ will be true at the same program point, so let us add that to our calculation.

Loop Invariant so far (2nd iteration)
$(x = nil \wedge emp) \vee (x \mapsto nil).$

At the next step we get

Loop Invariant so far (3rd iteration)
$(x = nil \wedge emp) \vee (x \mapsto nil) \vee (x{\mapsto}X * X{\mapsto}nil)$

because we put another element on the front of the list. If we keep going this way, we will get lists of length 3, 4 and so on: infinite regress. However, before we go around the loop again, we might employ abstraction, to conclude that we have a list segment. That is, we use the entailment

$$x{\mapsto}X * X{\mapsto}nil \ \vdash \ lsne(x, nil)$$

on the third disjunct, giving us

Loop Invariant so far (3rd iteration after abstraction)
$(x = nil \wedge emp) \vee (x \mapsto nil) \vee lsne(x, nil)$

Loss of information has occurred because in the third disjunct we have forgotten that the list from x is of length precisely 2. And, by this step we have taken ourselves to a situation where the assertion describes finitely many heaps (up to isomorphism), to an assertion that describes infinitely many concrete heaps: combining abstraction with symbolic execution allows us to cover a great many more heaps.

Now, if we go around the loop again, we obtain

Loop Invariant so far (4th iteration before abstraction)
$(x = nil \wedge emp) \vee (x \mapsto nil) \vee (x{\mapsto}X * lsne(X, nil))$

and we can apply a Berdine/Calcagno abstraction rule

$$x{\mapsto}X * lsne(X, nil) \ \vdash \ lsne(x, nil)$$

to obtain

Loop Invariant so far (4th iteration after abstraction)
$(x = nil \wedge emp) \vee (x \mapsto nil) \vee lsne(x, nil)$

Lo and behold, what we have obtained on the 4th iteration after abstraction is the same as the 3rd. We might as well stop now, as further execution of this variety will not give us anything new: we have reached a fixed-point. As it happens, this loop invariant is also the postcondition of the procedure in this example.

In this narrative the human (me) was the abstract interpreter, choosing when and how to do abstraction. To implement a tool we need to make it systematic. In the SPACEINVADER breed of tools, this is done using rewrite rules that correspond to Berdine/Calcagno abstraction rules for entailment described in Section 4.4. The abstract interpreter is sound automatically because applying those rules to simplify formulae is just using the Hoare rule of consequence on the right. The art is in not applying the rules *too often*, which would make one lose too much information, sometimes resulting in `fault` coming out of your abstract interpreter for perfectly safe problems (a 'false alarm').

The way you set up a proof-theoretic abstract interpreter is as follows. In addition to symbolic execution rules, there are abstraction rules which you apply periodically (say, when going through loop iterations); this allows the exection process to saturate (find a fixed-point). Here are some of the rules used in (baby) SPACEINVADER [30].

$$(\rho, \rho' \text{ range over } lsne, \mapsto)$$

$$\exists \vec{X}.H * \rho(x,Y) * \rho'(Y,Z) \longrightarrow \exists \vec{X}.H * lsne(x,Z) \quad \text{where } Y \text{ not free in } H$$

$$\exists \vec{X}.H * \rho(Y,Z) \longrightarrow \exists \vec{X}.H * true \quad \text{where } Y \text{ not provably reachable from program vars}$$

The first rule says to forget about the length of uninterrupted list segments, where there are no outside pointers (from H) into the internal point. The abstraction 'gobbles up' logical variables appearing in internal points of lists, by swallowing them into list segments, as long as these internal points are unshared. This is true of either free or bound logical variables. The requirement that they not be shared is an accuracy rather than soundness consideration; we stop the rules from firing too often, so as not to lose too much information.

[A remark on terminology: What I am simply calling the 'abstraction' step is a special case of what is called Widening in the abstract interpretation literature [24], and a direct analogue of the 'canonical abstraction' used in 3-valued shape analysis [76].]

Exercise 11 *Define a program that never faults, but for which the abstract semantics just sketched returns* `fault`.

Although I have not given a fully formal specification of the abstract interpreter above, thinking about the nature of the list segment predicates and the restricted syntax of symbolic heaps is one way to find such a program (e.g., if the program needs a loop invariant that is not expressible with finitely many symbolic heaps).

This exercise actually concerns a general point in program analysis. If you have a terminating analysis that is trying to solve an undecidable problem, it must necessarily be possible to trick the analysis. Since most interesting questions about programs are undecidable, we must accept that any program analysis for these questions will have an heuristic aspect in its design.

4.7. Contextual Remarks

This specific abstraction idea in the illustration in this section, to forget about the length of uninterrupted list segments, is sometimes called 'the Distefano abstraction': it was defined by Distefano in his PhD thesis [29]. The idea does not depend on separation logic, and similar ideas have been used in other abstract domains, such as based on 3-valued logic [54] or on graphs [55]. Once SMALLFOOT's symbolic execution appeared, it simply was relatively easy to port Distefano's abstraction to separation logic, when defining BABY SPACEINVADER [30]. Around the same time, very similar ideas were independently discovered by Magill et. al. [52].

These first abstract interpreters did not achieve a lot practically, but opened up the possibility of exploring the use of separation logic in program analysis. A growing amount of work has been going forward in a number of directions, an incomplete list of which includes the following, which are good places to start for further reading.

1. The use of the frame rule and frame inference $A \vdash B * \texttt{?frame}$ in interprocedural analysis [39];

2. The use of abductive inference $A * \texttt{?antiframe} \vdash B$ to approximate footprints, leading to a compositional analysis, and a boost to the level of automation and scalability [17];
3. The use of a higher-order list segment notion to attack complicated data structures in device drivers [5,85];
4. Analyses for concurrent programs [40];
5. Automatic parallelization [71,46];
6. Program-termination proving [8,51,14];
7. Analysis of data structures with sharing [21,50].

This last section has been but a sketch, and I have left out a lot of details. I will possibly extend these notes in future to put more formalities into this last section. For now I just point you to [30] for a mathematically thorough description of one abstract interpreter based on separation logic.

The leading edge, as of 2011, of what can be achieved practically on real-world code by these tools is probably represented by SLAYER [9] and INFER [16], and can be glimpsed by academic papers that fed into them [85,17]. However, there are some areas (sharing, trees) where academic prototypes outperform them precision-wise, and the leading edge in any case is moving quickly at this moment.

I have talked about automatic verification and analysis in these notes, but many of the ideas – such as frame inference, symbolic execution, abstraction/subtraction-based proof theory – are relevant as well in interactive proving. There have been several embeddings of separation logic in higher-order logics used in interactive proof assistants (e.g., [56, 35,82,79,58,1]), where the proof theoretic or symbolic execution rules are derived as lemmas. A recent paper [22] gives a good account of the state of the art and references to the literature, as well as an explanation of expressivity limitations of approaches to program verification based on automatic theorem proving for first-order logic.

It should be mentioned that there is no conflict here of having several logics (separation, first-order, higher-order, etc.): there is no need to search for 'the one true logic'. In particular, even though they can be embedded in foundational higher-order logics, special-purpose formalisms like separation and modal and temporal logics are useful for identifying specification and reasoning idioms that make specifications and proofs easier to find, for either the human or the machine.

References

[1] A.W. Appel. VeriSmall: Verified Smallfoot shape analysis. CPP 2011: First International Conference on Certified Programs and Proofs, 2011.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference*, pages 73–85, 2006.

[3] A. Banerjee, D.A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *22nd ECOOP, Springer LNCS 5142*, pages 387–411, 2008.

[4] M. Barnett, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[5] 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*.

[6] 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.

[7] J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.

[8] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV, Springer LNCS 4144*, pages pp386–400, 2006.

[9] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *23rd CAV, Springer LNCS 6806*, pages 178–183, 2011.

[10] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 5(29), 2007.

[11] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions of Software Engineering*, 21:809–838, 1995.

[12] R. Bornat. Proving pointer programs in Hoare logic. *Mathematics of Program Construction*, 2000.

[13] M. Botincan, M.J. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *Electr. Notes Theor. Comput. Sci.*, 254:5–23, 2009.

[14] J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *35th POPL*, pages 101–112, 2008.

[15] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[16] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium, Springer LNCS 6617*, pages 459–465, 2011.

[17] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. Journal of the ACM 58(6). (Preliminary version appeared in POPL'09.), 2011.

[18] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd LICS*, pp366-378, 2007.

[19] C. Calcagno, M.J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *14th SAS, Springer LNCS 4634*, pages 233–248, 2007.

[20] B. Chang and X. Rival. Relational inductive shape analysis. In *36th POPL*, pages 247–260. ACM, 2008.

[21] R. Cherini, L. Rearte, and J.O. Blanco. A shape analysis for non-linear data structures. In *17th SAS, Springer LNCS 6337*, pages 201–217, 2010.

[22] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd PLDI*, pages 234–245, 2011.

[23] B. Cook, C. Haase, J. Ouaknine, M.J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *22nd CONCUR, Springer LNCS 6901*, pages 235–249, 2011.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pp238-252, 1977.

[25] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. 6th POPL, pp269-282, 1979.

[26] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. 14th ESOP, pp21-30, 2005.

[27] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[28] T. Dinsdale-Young, P. Gardner, and M.J. Wheelhouse. Abstraction and refinement for local reasoning. In *3rd VSTTE, Springer LNCS 6217*, pages 199–215, 2010.

[29] D. Distefano. *On model checking the dynamics of object-based software: a foundational approach*. PhD thesis, University of Twente, 2003.

[30] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *12th TACAS*, 2006. pp287-302.

[31] D. Distefano and M. Parkinson. jStar: Towards Practical Verification for Java. In *23rd OOPSLA*, pages 213–226, 2008.

[32] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. Modular reasoning for deterministic parallelism. In *38th POPL*, pages 259–270, 2011.

[33] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *23rd CAV, Springer LNCS 6806*, pages 372–378, 2011.

[34] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *16th ESOP, Springer LNCS 4421*, 2007.

[35] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify

complete software systems. In *2nd VSTTE, Springer LNCS 5295*, pages 54–69, 2008.

[36] R.W. Floyd. Assigning meaning to programs. *Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., pp. 19Ð32*, 1967.

[37] M. Foley and C.A.R. Hoare. Proof of a recursive program: Quicksort. *Computer Journal*, 14:391–395, 1971.

[38] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for Javascript. In *40th POPL*. ACM, 2012.

[39] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *13th SAS,Springer LNCS 4134*, pages 240–260, 2006.

[40] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Mostly-automated verification of low-level programs in computational separation logic. In *28th PLDI*, pages 266–277, 2007.

[41] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar. Forest automata for verification of heap manipulation. In *23rd CAV, Springer LNCS 6806*, 2011.

[42] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580 and 583, 1969.

[43] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engler, editor, *Symposium on the Semantics of Algebraic Languages*, pages 102–116. Springer, 1971. Lecture Notes in Math. 188.

[44] C.A.R. Hoare. Proof of a Program: FIND. *Comm. ACM*, 14(1):39–45, 1971.

[45] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[46] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *16th SAS, Springer LNCS 5673*, pages 52–68, 2009.

[47] S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.

[48] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium, Springer LNCS 6617*, pages 41–55, 2011.

[49] I.T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.

[50] O. Lee, H.Yang, and R. Petersen. Program analysis for overlaid data structures. In *23rd CAV, Springer LNCS 6808*, pages 592–608, 2011.

[51] S. Magill, J. Berdine, E.M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *14th SAS, Springer LNCS 4634*, pages 419–436, 2007.

[52] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. *3rd SPACE Workshop*, 2006.

[53] S. Magill, M.-S. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. *20th CAV, Springer LNCS 5123.* pp 428-432, 2008.

[54] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *6th VMCAI*, pages pp181–198, 2005.

[55] M. Marron, M.V. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *17th CC, Springer LNCS 4959*, pages 245–259, 2008.

[56] N. Marti and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software, 25(3):135-147*, 2008.

[57] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[58] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *37th POPL*, pages 261–274, 2010.

[59] H.H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. *20th CAV, Springer LNCS 5123.* pp 355-369, 2008.

[60] H.H. Nguyen, C. David, S. Qin, and W.-Ngan Chin. Automated verification of shape and size properties via separation logic. In *8th VMCAI, Springer LNCS 4349*, pages 251–266, 2007.

[61] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1-19, 2001.

[62] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp49-67).

[63] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[64] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3), 2009.
[65] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *21st LICS*, 2006.
[66] M. J. Parkinson. *Local Reasoning for Java*. Ph.D. thesis, University of Cambridge, 2005.
[67] M.J. Parkinson and A.J. Summers. The relationship between separation logic and implicit dynamic frames. In *20th ESOP, Springer LNCS 6602*, pages 439–458, 2011.
[68] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *32nd PLDI*, pages 556–566, 2011.
[69] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.
[70] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, 2002.
[71] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *18th ESOP, Springer LNCS 5502*, pages 348–362, 2009.
[72] M. Raza and P. Gardner. Footprints in local reasoning. *Logical Methods in Computer Science*, 5(2), 2009.
[73] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.
[74] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp55-74, 2002.
[75] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
[76] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
[77] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *23rd ECOOP, LNCS 5653*, pages 148–172, 2009.
[78] C. Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference*, pages 198–220, Baden bei Wien, Austria, September 1964. North-Holland, Amsterdam, 1966.
[79] T. Tuerk. A formalisation of Smallfoot in HOL. In *TPHOLs, 22nd International Conference, LNCS 5674*, pages 469–484, 2009.
[80] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *10th VMCAI, LNCS 5403*, pages 335–348, 2009.
[81] V. Vafeiadis and M.J. Parkinson. A marriage of rely/guarantee and separation logic. In *18th CONCUR, Springer LNCS 4703*, pages 256–271, 2007.
[82] C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *24th MFPS*, 2008.
[83] J. Villard, É. Lozes, and C. Calcagno. Tracking heaps that hop with Heap-Hop. In *16th TACAS, Springer LNCS 6015*, pages 275–279, 2010.
[84] H. Yang. *Local Reasoning for Stateful Programs*. Ph.D. thesis, University of Illinois, Urbana-Champaign, 2001.
[85] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. *20th CAV, Springer LNCS 5123.* pp 385-398, 2008.
[86] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, 2002. Springer LNCS 2303 , pp402-416.