Separation Logic

Peter O'Hearn

Facebook and University College London

Dedicated to John C Reynolds

1. Introduction

A fundamental technique in reasoning about programs is the use of logical assertions to describe properties of program states. Turing used assertions to argue about the correctness of a particular program in 1949 [40], and they were incorporated into general formal systems for program proving starting with the work of Floyd [21] and Hoare [22] in the 1960s. Hoare logic, which Separation Logic builds upon, is a formal system for proving specifications of the form

$$\{precondition\}code\,\{postcondition\}$$

where the precondition and postcondition are assertions describing properties of the input and output states. For example,

$$\{x == N\}code\,\{x == N \wedge y == N!\}$$

can serve as a specification of an imperative program that computes the factorial of the value held in variable x and places it in y.

Hoare logic and related systems worked very well for programs manipulating simple primitive data types such as for integers or strings, but proofs became more complex when dealing with structured data containing embedded pointers. One of the founding papers of Separation Logic summarized the problem as follows [32].

> The main difficulty is not one of finding an in-principle adequate axiomatization of pointer operations; rather there is a mismatch between simple intuitions about the way that pointer operations work and the complexity of their axiomatic treatments. ... when there is aliasing, arising from several pointers to a given cell, an alteration to a cell may affect the values of many syntactically unrelated expressions.

Bornat provided a good description of the struggles in reasoning about mutable data structures up to the year 2000 [6].

In joint work with John Reynolds and others we developed Separation Logic to address the fundamental problem of reasoning about programs that mutate data structures. From a special logic for heaps it gradually evolved into a general theory for modular reasoning about concurrent as well as sequential programs. Efforts by many researchers established that the logic provides a basis for efficient proof search in automatic and semi-automatic proof tools, for example giving rise to the Infer static analyzer, a tool that is in deployment at Facebook where it catches thousands of bugs per month before code reaches production in products used by over a billion people daily.

Published in *Communications of the ACM, February 2019, vol. 62, no. 2.* doi:10.1145/3211968

Key Insights

- Separation Logic supports in-place updating of *facts* as we reason, in a way that mirrors in-place update of memory during execution, and this leads to logical proofs about imperative programs that match computational intuition.
- Separation Logic supports scalable reasoning by using an inference rule (the Frame Rule) that allows a proof to be localized to the resources that a program component accesses (its footprint).
- Concurrent Separation Logic shows that modular reasoning about threads that share storage and other resources is possible.

Separation logic (SL) is an extension of Hoare logic which employs novel logical operators, most importantly the separating conjunction $*$ (pronounced "and separately"), when writing assertions. For example, we might write

$$\begin{array}{l}\{x \mapsto 0 * y \mapsto 0\}\\ [x] = y;\\ [y] = x\\ \{x \mapsto y * y \mapsto x\}\end{array}$$

as a specification of code that wires together two memory locations into a cyclic linked list. Here $x \mapsto v$ says that pointer variable x holds the address of a memory location where v is stored (or more briefly, x points to v), and a command of the form $[x] = v$ updates the location referred to by x so that its contents becomes v'. The use of $*$ rather than the usual Boolean conjunction $\wedge$ ensures that x and y are not aliases – distinct names for the same location – so that we have a two-element cyclic list in the postcondition. A central principle is that a command that mutates a single location affects only one $*$-conjunct: operational in-place update is mirrored in the logic, addressing the key difficulty where "an alteration to a cell may affect the values of many syntactically unrelated expressions".

Reynolds was the first to describe a program logic including the separating conjunction; he defined an intuitionistic (constructive) logic with $*$ [37], building on earlier ideas of Burstall [10]. O'Hearn and Ishtiaq [26] realized that the assertion language could be seen as an instance of the resource logic BI of O'Hearn and Pym [31]; they independently discovered the same intuitionistic logic as Reynolds, and also saw that a more powerful Boolean (non-constructive) variant was possible in which one could reason about explicit memory management (Reynolds had assumed garbage collection). They also introduced the separating implication $-\!\!*$.

Separation Logic for sequential programs reached maturity in a further paper of O'Hearn, Reynolds and Yang [32]. In that work O'Hearn proposed the following *principle of local reasoning*, both as a way to describe what was special about SL and as a guiding principle for development of reasoning methods.

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

A proof rule, the Frame Rule, allowed to infer that cells remain unchanged when they are not mentioned in a precondition. The Frame Rule was named in homage to the *frame problem* from Artificial Intelligence, which concerns axiomatizing state changes without enumerating all of the things that don't change. The Frame Rule is the key to scalable reasoning in SL.

An influential survey article of Reynolds summarized the early developments up to 2002 [38]. At the end of this early period O'Hearn circulated a note which proposed a Concurrent Separation Logic (CSL). CSL showed efficient reasoning about threads that share access to storage, proofs which mirrored design principles espoused by Dijkstra at the birth of concurrent programming [16]. The correctness of CSL's proof rules (its 'soundness') turned out to be a formidable problem, solved eventually by Brookes. Brookes and O'Hearn were awarded the 2016 Gödel prize for their papers on CSL [30, 8], the significance of which was summed up as follows.

> For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to CSL, both of these problems have been elegantly and efficiently solved; and they have the same solution. *2016 Gödel Prize citation*[1]

It is worth remarking that the first part of this citation, about pointer manipulation, applies to sequential and not just concurrent SL.

After the early papers, research on SL expanded rapidly. Starting from a special logic for heaps SL has evolved into a general theory for modular reasoning. Non-standard models of SL based on an abstract model theory due to Pym provided many potential avenues for wider application, and Gardner and others realized that there exist non-standard models that support modular reasoning about intertwined structures *as if* they were separate. SL has even been applied to interfering processes using fine-grained concurrency, a situation far removed from the original intuitions of the logic.

SL is the basis of numerous automated proof tools, and it has been used in significant verification efforts. It has been used to provide the first verification of a crash-proof file system [14], and to provide the first verification of a commercial, preemptive OS microkernel [41]. These verification efforts are semi-automatic, done by a human together with a proof assistant (in these cases, the Coq proof assistant). SL has also been used in static program analysis, where weaker properties than full correctness are targeted but with higher automation, so that the tool can scale better both in the sizes of codebases covered and the number of programmers served. Static analysis with SL has matured to the point where it has been applied industrially in the Facebook Infer program analyzer, an open-source tool which is used at Facebook, Mozilla, Spotify, Amazon Web Services and other companies (www.fbinfer.com).

The purpose of this paper is to describe the basic ideas of SL as well as these and other developments.

2. Separating Conjunction and Implication

Mathematical semantics has been critical to the discovery and further development of Separation Logic, but many of the main points can be gleaned from a "picture semantics". Consider the first picture in Figure 1. 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

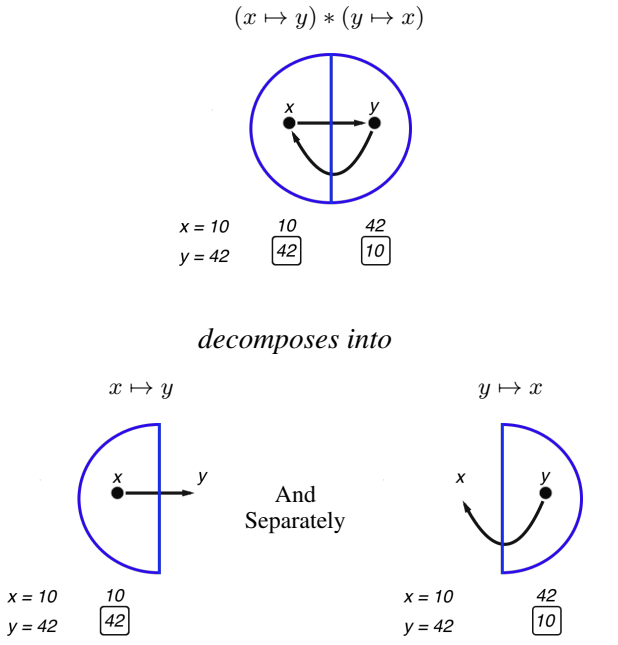


Figure 1. Picture Semantics

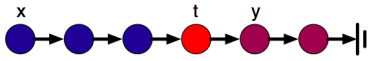
separating conjunction asks for a partitioning that divides the heap into parts, *heaplets*, satisfying its two conjuncts. At the bottom of the first picture is an example of a concrete memory description that corresponds to the diagram. There, x and y have values 10 and 42 (in the "environment", or "register bank"), and 10 and 42 are themselves locations with the indicated contents (in the "heaplet", or even "RAM").

The indicated separating conjunction above is true of the pictured memory because the parts satisfy the conjuncts, as indicated in the second picture. The meaning of "x points to y and yet to nothing" is precisely disambiguated in the RAM description below the diagram: 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. The separating conjunction splits the heap/RAM, but it does not split the association of variables to values.

Generally speaking, the separating conjunction $P * Q$ is true of a heap if it can be split into two heaplets, one of which makes P true and the other of which makes Q true. A distinction between $*$ and Boolean conjunction $\wedge$ is that $P * P \neq P$ where $P \wedge P = P$. In particular, $x \mapsto v * x \mapsto v$ is always false: there is no way to divide any heap in such a way that a cell x goes to both partitions.

$*$ is often used with linked structures. If $list(x, y)$ describes an acyclic linked list running from x to y, then we can describes a structure with a list segment, followed by a single pointer, followed by a further list running up to 0 (null), as follows:

$$list(x, t) * t \mapsto y * list(y, 0)$$



This is the kind of structure you might need to consider when deleting an element from a list, or inserting one into it.

There is a further connective, the separating implication or "magic wand". $P \mathrel{-\!\!*} Q$ says that whenever the current heaplet is extended with a separate heaplet satisfying P, the resulting combined heaplet will satisfy Q. For example, $(x \mapsto -) * ((x \mapsto 3) \mathrel{-\!\!*} Q)$

[1] http://eatcs.org/index.php/component/content/article/1-news/2280-2016-godel-prize

- Assume a partial commutative monoid $(H, \circ, e)$, where $\circ : H \times H \rightharpoonup H$ and $e \in H$. Pre/post assertions denote elements of the powerset $\mathcal{P}(H)$.
- $*$ lifts $\circ$ to the powerset $\mathcal{P}(H)$: $P * Q$ is

  $\{h_P \circ h_Q \mid h_P \circ h_Q$ defined and $h_P \in P$ and $h_Q \in Q\}$
- `emp` denotes the singleton set of the empty heaplet: $\{e\}$.
- $-\!\!*$ is an implication quantifying over separate heaps: $P -\!\!* Q$ is

  $\{h \mid \forall h_P.\ h \circ h_P$ defined and $h_P \in P$ implies $h \circ h_P \in Q\}$
- In the RAM model H is the set of finite partial functions from positive integers (addressible locations) to integers, $h \circ h'$ is the union of functions with disjoint domain, and undefined when h and h' overlap. e is the empty partial function. The assertion $n \mapsto m$ denotes the singleton set $\{f\}$ where f maps n to m and is undefined elsewhere.
- To deal with variables and also quantifiers consider functions s from variables to integers, and extend the above semantics pointwise to pairs (s, h).

Figure 2. Mathematical Semantics

says that x is allocated in the current heap, and that if you mutate its contents to 3 then Q will hold. This describes the "weakest precondition" for the mutation $[x] = 3$ with postcondition Q [26].

Finally, there is an assertion `emp` which says "the heaplet is empty". `emp` is the unit of $*$, so that $P = \texttt{emp} * P = P * \texttt{emp}$. Also, $-\!\!*$ and $*$ fit together is a way similarly to how implication $\Rightarrow$ and conjunction $\wedge$ do in standard logic. For example, the entailment

$$A * (A -\!\!* B) \vdash B$$

(where $\vdash$ reads "entails") is a SL relative of "modus ponens".

Although we will concentrate on the informal picture semantics in this paper, for the theoretically inclined we have included a glimpse of the formal semantics in Figure 2.

3. Rules for Program Proof

Figure 3 contains a selection of proof rules of SL. The rules are divided into axioms for basic mutation commands (the "small axioms") and inference rules for modular reasoning. An inference rule says "if you can derive what is above the line, then so can you what is below", and the axioms are derivable true statements that are given. The small axioms are for a programming language with load and store instructions similar to an assembly language. If we vary the programming language the small axioms change. The Concurrency Rule uses a composition operator $\|$ for running two processes in parallel, derived from Dijkstra's parbegin/parend [16].

The first small axiom just says that if x points to something beforehand, then it points to v afterwards, and it says this for a small portion of the state in which x is the only active cell.

The second axiom says that if x points to v and we read x into y, then y will have value v. Here, we distinguish between the value in a variable or register (x and y) and the r-value in a heap cell whose l-value is the value held in x. The second axiom assumes that x does not appear in syntactic expression v (see [32] for a precise description of this and other variable side conditions).

The Allocation axiom says: if you start with no heap, then you end with a heap of size 1. Conversely the De-Allocation axiom starts with a hap of size 1 and ends with the empty heap. The Application axiom assumes that allocation always succeeds. To model a case where allocation might fail we could use a disjunctive

SMALL AXIOMS

Pointer Write (Store)

$\{x \mapsto -\}[x] = v\ \{x \mapsto v\}$

Pointer Read (Load)

$\{x \mapsto v\} y = [x]\ \{y == v \wedge x \mapsto v\}$

Allocation

$\{emp\} x = \texttt{alloc}()\ \{x \mapsto -\}$

De-Allocation

$\{x \mapsto -\}\texttt{free}(x)\ \{emp\}$

LOCAL REASONING RULES

Frame Rule

$$\frac{\{pre\}code\ \{post\}}{\{pre * frame\}code\ \{post * frame\}}$$

Concurrency Rule

$$\frac{\{pre_1\}process_1\ \{post_1\} \quad \{pre_2\}process_2\ \{post_2\}}{\{pre_1 * pre_2\}process_1 \parallel process_2\ \{post_1 * post_2\}}$$

Figure 3. Separation Logic Proof System (a Selection)

postcondition, like $x \mapsto - \vee x == 0$; this is what tools such as SpaceInvader and Infer, discussed later, do for `malloc()` in C.

The small axioms are so named because each mentions a small amount of memory: a single memory cell. When people first see the axioms they can come as a shock: aren't they *too simple*? Previous approaches had complex descriptions accounting for the effect of mutations on global properties of graph-like structures [6].

In actuality, there is a sense in which the small axioms capture all that is needed to know about the statements they describe. In intuitive terms we can say that imperative computation proceeds by in-place update, where these primitive statements update or access a single memory cell at a time; describing what happens to only that cell should be enough. The small axioms are thus an extreme illustration of the principle of local reasoning.

The Frame Rule in Figure 3 provides logical support for this intuition. It allows us to extend reasoning from one to multiple cells; so the seeming restriction to one cell in the small axioms is not a restriction at all, but rather a pleasantly succinct description. For instance, if we choose $x \mapsto y$ as our frame then the first instance in Figure 4 gives the reasoning for the second step of the code to wire up a cyclic linked list described at the start of the paper.

The ultimate theoretical support for the small axioms came from a completeness theorem in Yang's PhD thesis [42]. He showed that the small axioms and Frame Rule and several other inference rules (particularly Hoare's rules for strengthening preconditions and weakening postconditions, and a rule for existential quantifiers) can be used to derive all true Hoare triples for these statements. Locality properties of program behaviour, and their connection to logic [44, 13], are critical for these results:

> An assertion talks about a heaplet rather than the global heap, and a spec $\{P\}C\ \{Q\}$ says that if C is given a heaplet satisfying P then it will never try to access heap outside of P (other than cells allocated during execution) and it will deliver a heaplet satisfying Q if it terminates. [2]

In-place reasoning as with the two-element cyclic list has been applied to many imperative programs. As an example, consider the

$$\frac{\{y \mapsto 0\}\ [y] = x\ \{y \mapsto x\}}{\{(y \mapsto 0) * (x \mapsto y)\}\ [y] = x\ \{(y \mapsto x) * (x \mapsto y)\}}$$

$$\frac{\{x \mapsto 0\}\ [x] = y\ \{x \mapsto y\} \qquad \{y \mapsto 0\}\ [y] = x\ \{y \mapsto x\}}{\{(y \mapsto 0) * (x \mapsto 0)\}\ [x] = y \parallel [y] = x\ \{(y \mapsto x) * (x \mapsto y)\}}$$

Figure 4. Frame and Concurrency Examples

insertion of a node y into a linked list after position x. We can do this in two steps: first we swing x's pointer so that it points to y, and then we swing y to point to z (the node after x).

$$\begin{array}{l}\{x \mapsto z * list(z) * y \mapsto -\}\\ [x] = y\\ \{x \mapsto y * list(z) * y \mapsto -\}\\ [y] = z\\ \{x \mapsto y * list(z) * y \mapsto z\}\end{array}$$

Here, in the precondition for each step we write the frame in red; it is the blue that is updated in place. The reader can see how, using the small axiom for `free` together with the Frame Rule, we could reason about the converse case of removing an element from a list.

This example generalizes to many other list and tree algorithms: insertion, deletion, reversal, etc. The proofs in SL resemble the box-and-pointer arguments which have long been used informally in describing data structure mutation.

These ideas extend to concurrent programs; for example, the second rule instance in Figure 4 uses the Concurrency Rule to reasons about our two-element cyclic list, but wired up concurrently rather than sequentially. The $*$ in the precondition in this instance ensures that x and y are not aliases, so that there is no data race in the parallel program.

The Concurrency Rule is the main rule of CSL. In applying CSL to languages with dynamic thread creation instead of parbegin/-parend different rules are needed, but the basic point that separation allows independent reasoning about processes carries over.

SL's Concurrency Rule took inspiration from the "disjoint concurrency rule" of Hoare [23]. Hoare's rule used $\wedge$ in place of $*$ together with side conditions to rule out interference[2]. $*$ allows us to extend its applicability to pointer structures. But even without pointers, the CSL rule is more powerful. Indeed, upon seeing CSL Hoare immediately exclaimed to the author: "we can prove parallel quicksort!". A direct proof can be given using $*$ to recognize and unite disjoint array partitions [30].

4. Frames, Footprints and Local Reasoning

The previous section describes how the separating conjunction leads to simple proofs of the individual steps of heap mutations, and how the Frame Rule embeds reasoning about small chunks of memory within larger memories. In this section we explain the more fundamental role of the rule as a basis for scalable reasoning.

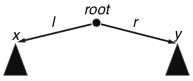
We illustrate by reasoning about a recursive program for deleting the nodes in a binary tree. Consider the C program in (1) of Figure 5. This program satisfies the specification in (2) of the figure, where the $tree$ predicate says that its argument points to a binary tree in memory. The predicate is defined recursively in (3), with a picture below depicting what is described by the `else` part of the definition. Note that here we are using a "points-to" predicate $root \mapsto [l : x, r : y]$ for describing records with l and r fields.

[2] In some presentations of SL there are variable conditions, which can technically be done away with by using a version of $*$ that separates variables as well as heap [34]. In this paper we will gloss over this issue.

(1)
```
void deletetree(struct node *root)
{ if( root != 0 )
  { struct node *left = root->l, *right = root->r;
    deletetree(left);
    deletetree(right);
    free( root );
} }
```

(2) Spec: $\{tree(root)\}$ `deletetree(root)` $\{$`emp`$\}$

(3) $tree(root) =$ `if` $root == 0$ `then emp`
`else` $\exists xy.\ root \mapsto [l : x, r : y] * tree\ (x) * tree(y).$



(4)
$\{root \mapsto [l : left, r : right] * tree(left) * tree\ (right)\}$
`deletetree(left);`
$\{root \mapsto [l : left, r : right]$ `*` `emp` $* tree(right)\}$
`deletetree(right);`
$\{root \mapsto [l : left, r : right] *$ `emp` $*$ `emp`$\}$
`free( root );`
$\{$`emp` $*$ `emp` $*$ `emp`$\}$
$\{$`emp`$\}$

Figure 5. `deletetree` Example

The use of `emp` in the `if` branch of the definition means that $tree(r)$ is true of a heaplet which contains all and only the cells in the tree; there are no additional cells. Thus, the specification of `deletetree(r)` does not mention nodes not in in the tree. This is analogous to what we did with the small axioms for basic statements in Figure 3, and is a typical pattern in SL reasoning: "small specifications" are used which mention only the cells touched by the program component (its footprint).

The critical part of the proof of the program is presented in (4), where the precondition at the beginning is obtained by unwinding the recursive definition using the if condition $root \mathrel{!}= 0$. The proof steps then follow the intuitive description of the algorithm: the first recursive call deletes the left subtree, the second call deletes the right subtree, and the final statement deletes the root node. In the pictured reasoning, the overall specification of the procedure is applied as an induction hypothesis at each call site, together with the Frame Rule for showing that the parts not touched by recursive calls are left unchanged. For instance, the assertions for the second recursive call are an instance of the Frame Rule with the triple $\{tree(right)\}$ `deletetree(right)` $\{$`emp`$\}$ as the premise.

The simplicity of this proof comes about because of the principle of local reasoning. The Frame Rule allows in-place reasoning for larger-scale operations (entire procedures) than individual heap mutations. And it allows the specification to concentrate on the footprint of a procedure instead of the global state. Put contrapositively, the `deletetree` procedure could not be verified without the Frame Rule, unless we were to complicate the initial specification by including some representation of frame axioms (saying what does not change) to enable the proofs at the recursive call sites.

The above reasoning uses a tree predicate that is suitable for reasoning about memory safety; it mentions that we have a tree, but not what data it holds. For functional correctness reasoning it is typical to use inductive predicates that connect memory structures to mathematical entities. In place of $tree(root)$ we could have a predicate $tree(\tau, root)$ which says that $root$ points to an area of memory representing the mathematical binary tree τ, where a

mathematical tree is either empty or an atom or a pair of trees. We could then specify a procedure for copying a tree using a postcondition of the form

$$\mathit{tree}\ (\tau,\ \mathit{oldroot}) \ * \ \mathit{tree}\ (\tau,\ \mathit{newroot})$$

which says that we have two structures in memory representing the same mathematical tree. An assertion like this would tell us that we could mutate one of the trees without affecting the other (at which point they would cease to represent the same tree).

For data structures without much sharing, such as variations on lists and trees, reasoning in SL is reminiscent of reasoning about purely functional programs: you unroll an inductive definition, then mutate, then roll it back up.Inductive definitions using $*$ and mutation go well together. The first SL proof to address complex sharing was done by Yang in his PhD thesis, where he provided a verification of the classic Schorr-Waite graph marking algorithm. The algorithm works by reversing links during search, and then restoring them later: a space-saving representation of the stack of a recursive algorithm. Part of the main invariant in Yang's proof is

$$\big(\mathit{listMarkedNodesR(stack,\ p)} \\ * (\mathit{restoredListR(stack,t)} \mathrel{-\!\!*} \mathit{spansR(STree,\ root)})\big)$$

capturing the idea that if you replace the list of marked nodes by a restored list, then you get a spanning tree. Yang's proof reflected the intuition that the algorithm works by a series of *local surgeries* that mutate small parts of the structure: the proof decomposed into verifications of the surgeries, and ways of combining them.

The idiomatic use of $\mathrel{-\!\!*}$ in assertions of the form $A * (B \mathrel{-\!\!*} C)$ to describe generalized update was elevated to a general principle in work of Hobor and Villard [25]. They give proofs of a number of programs with significant sharing, including graphs, dags, overlaid structures (e.g., a list overlaying a tree), and culminating in the copying algorithm in Cheney's garbage collector.

Many papers on SL have avoided $\mathrel{-\!\!*}$, often on the grounds that it complicates automation and is only needed for programs with significant sharing. However, $\mathrel{-\!\!*}$ is recently making something of a comeback. For example, it is used routinely as a basic tool in the Iris higher-order logic [29].

5. Concurrency, Ownership and Separation

The Concurrency Rule in Figure 3 says: To prove a parallel composition we give each process a separate piece of state, and separately combine the postconditions for each process. The rule supports completely independent reasoning about processes. This rule can be used to provide straightforward proofs of processes that don't share access to storage. We mentioned parallel quicksort before, and `deletetree()` provides another illustration: we can run the two recursive calls in parallel rather than sequentially, as presented in the proof outline (1) in Figure 6.

In work on CSL, proof outlines are often presented in a spatial fashion like this: this outline shows the premises of the concurrency rule in the left and right Hoare triples, the overall precondition (the $pre1 * pre2$) at the beginning, and the post at the end.

While this reasoning is simple, if CSL had only been able to reason about disjoint concurrency, where there is no inter-process interaction, then it would have rightly been considered rather restrictive. An important early example done with CSL was a pointer transferring buffer, where one thread allocates a pointer and puts it into a buffer while the other thread reads it out and frees it. Crucially, not only is the pointer deemed to transfer from one process to another, but the "knowledge that it is allocated" transfers with the proof. The proof establishing absence of memory errors is shown in (2) of Figure 6. A way to implement the buffer code for `put` and `get` is to use locks to synchronize access to a shared variable and a Boolean to signal when the buffer is full. We will not delve into the

(1)

$\{\mathit{tree(left)} * \mathit{tree(right)}\}$

$\{\mathit{tree(left)}\}$		$\{\mathit{tree(right)}\}$
`deletetree(left)`	$\|\|$	`deletetree(right)`
$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$

$\{\mathtt{emp} * \mathtt{emp}\}$

(2)

$\{\mathtt{emp} * \mathtt{emp}\}$

$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$
$x = \mathtt{alloc}();$	$\|\|$	$\mathtt{get}(y);$
$\{x \mapsto -\}$		$\{y \mapsto -\}$
$\mathtt{put}(x);$		$\mathtt{free}(y);$
$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$

$\{\mathtt{emp} * \mathtt{emp}\}$

Figure 6. Concurrency Proofs

subproofs of buffer operations here – for that, consult [30] – but we want to talk about a shift in perspective on the meanings of logical assertions that the proof (2) led to.

Notice the assertion `emp` after the $\mathtt{put}(x)$ statement in the left process. We could not prove a mutation were we to place it there, because `emp` is not a sufficient precondition for any mutation; that is fortunate as such a mutation could lead to a race condition. But it is not the case that we know that the global heap is empty, because the pointer x could still persist. Rather, the knowledge that it points to something has been forgotten, transferred to the second process where it materializes as $y \mapsto -$. A reading of assertions began to form based on the "right to dereference" or "ownership" (taken as synonymous with right to dereference). On this reading `emp` says "I don't have permission to dereference any heap", or "I own nothing", rather than "the heap is empty". Similarly, $x \mapsto -$ says "I own x" (where "I" is the process from which the assertion is made).

The ownership transfer example made it clear that quite a few concurrent programs would have much simpler proofs than before. Modular proofs were provided of semaphore programs, of a toy memory manager, and programs with interacting resources. It seemed as if the proofs mirrored design principles used to simplify reasoning about concurrent processes, such as in Dijkstra's idea of *loosely connected processes*:

> "apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other. [16]"

However, the very feature that gave rise to the unexpected power, ownership transfer, made soundness (whether the rules prove only true statements) nonobvious. O'Hearn worked on soundness during 2001 and 2002, without success. In May of 2002 he turned to Brookes who eventually (with important input from Reynolds), in 2004, proved the theorem which justified the logic

6. Abstraction and the Fiction of Separation

After the early papers there was considerable work on extending SL. Some of it concentrated on different programming paradigms, such as object-oriented programming or scripting languages, or on additional programming primitives such as message passing, reentrant lock and fork/join concurrency. Besides extensions to cover an ever greater variety of programming, two conceptual developments opened up major new directions.

- In his PhD thesis Parkinson showed how abstract predicates (predicate variables) fit together nearly with $*$ in the description of classes and other stateful data abstractions [33].

- Gardner and others emphasized a concept of fictional separation, where strong separation properties could be assumed of data abstractions, even for implementations relying on sharing.

These ideas were first described in a sequential setting. Dinsdale-Young, Gardner and Wheelhouse described an implementation of a module of sequences in terms of linked lists and noted a mismatch: at the abstract level an operation might affect a small part of a sequence, where at the implementation level its *footprint* could involve the entire list; conversely, *locality can increase with abstraction* [19]. Meanwhile, Parkinson initially targeted a sequential subset of Java. Subsequent work showed how abstract predicates could be understood using higher-order versions of SL [5].

While they could be expressed in a sequential setting, the ideas took flight when transported to concurrency. The CAP logic [18] combined insights on abstract predicates and fiction, along with those of CSL, to reason about data abstractions with interference in their implementations. The Views theory [17] provided a foundation where separation does not appear in the normal execution semantics of programs, but only in an abstraction of it. Views showed that a simple version of CSL can embed many other techniques including even the classic rely-guarantee method [27]; this is surprising because rely-guarantee was invented for reasoning about interference, almost the opposite of the basis of original SL.

Nowadays, advanced logics are often formulated as variations on the theme of "higher-order concurrent separation logic". One of these, Verifiable C, is the foundation of Appel's Verified Software Toolchain [1], and includes an expressive higher-order logic supporting recursive predicates. Another, Iris [29], encompasses reasoning about fine-grained concurrency and even relaxed memory, based on different instantiations of a single generic model. Iris has been used to provide a foundation of the type system of the Rust programming language [28], which is very natural when you consider that ownership transfer is one of the central ideas in Rust.

Technically, these works are based on "non-standard models" of SL, different from the heaplet model but instances of Pym's resource semantics as in Figure 2; see [36]. There are many such models, including ones incorporating read and other permissions [7], auxiliary state [39], time [39], protocols [29] and others. Abstract SL [13] showed how a general program logic could be defined based on these models, and the works just mentioned and others showed that some of them had surprising ramifications.

Fictional separation and Views re-imagined fundamental concepts. The programs being proven go beyond the loosely connected processes that CSL was originally designed for. Significant new theoretical insights and soundness arguments were needed to justify the program-proof rules supporting the fine-grained concurrency examples [17]. This led to a flowering of interest and new ideas which is still in progress. A recent survey on CSL provides many more references in addition to those mentioned here [9].

7. Directions in Mechanized Reasoning

SL spawned new approaches to verification tools. In order to provide a taste of where the field has gotten to we present a sampling of practical achievements; that is, we focus on the end points rather than the (important) advancements along the way that helped get there. Further references to the literature, including discussion on intermediate advances, may be found in the appendix.

Mostly Automatic Verification *Smallfoot* [2], due to Calcagno, Berdine and O'Hearn, was the first SL verification tool. Given procedure pre/post specs, loop invariants and invariants governing lock usage, Smallfoot attempts to construct a proof. For the pointer-transferring buffer, given a buffer invariant and pre/post specs for `put` and `get` it can verify memory safety and race freedom.

Smallfoot used a decidable fragment of SL dubbed "symbolic heaps", formulae of the form $B \wedge H$ where H is a separating conjunction of heap facts and B is a Boolean assertion over non-heap data. The format was chosen to make in-place symbolic execution efficient. Smallfoot's heap facts were restricted to points-to assertions, linked lists and trees. Subsequent works extended symbolic heaps in numerous directions, covering more inductive definitions as well as arrays and arithmetic; see appendix.

Some of the most substantial automatic verifications done with SL have been carried out with the VeriFast tool of Jacobs and colleagues. VeriFast employs a symbolic execution engine like Smallfoot, but integrates a dedicated SL theorem prover with a classical SMT solver for non-heap data. A paper reports on the verification of several industrial case studies, including Java Card programs and device drivers written in C [35]; see VeriFast's GitHub site for these and many other examples (https://github.com/verifast/verifast).

Interactive Verification In an automatic verifier like Smallfoot the proof construction is automatic, given the pre/post annotations plus invariants. In interactive verification the human helps guide the proof search, commonly using a proof assistant such as Coq, HOL4 or Isabelle. Interactive verification can often prove stronger properties than automatic verifiers, but the cost is higher.

Interactive verifiers have been used to prove small, intricate algorithms. A recent paper reports on the verification of low-level concurrent algorithms including a CAS-lock, a Ticketed lock, a GC allocator, and a non-blocking stack [39]. An emphasis is placed on reusability; for instance, the stack uses the GC allocator, which in turn uses a lock, but the stack uses the spec of the allocator and the allocator uses the spec rather than the implementation of a lock.

The Verifiable C logic [1] has been used to prove crypto code. For example, OpenSSL's HMAC authentication code, comprising 134 lines of C, was proven using 2832 lines of Coq [4].

A larger example is the FSCQ file system [14]. The code and the proof are both done in Coq, taking up 31k lines of proof+code. This compares to 3k lines of C for a related unverified file system. Although the initial effort, which included development of a program logic framework in Coq, took several person years, experiments show incremental, lower cost when modifying code+proof.

A commercial example concerns key modules of a preemptive OS kernel, the μC/OS-II [41]. Modules verified include the scheduler, interrupt handlers and message queues. 1.3k lines of C were proven using 216k lines of Coq. It took 4 person years to develop the framework, 1 person year to prove the first module, and then the remaining modules, around 900 lines of C, took 6 person-months.

Automatic Program Analysis. With a verification-oriented program analysis the annotations that a human would supply to a mostly automatic verifier like Smallfoot – invariants and pre/post specs – are inferred. A tool will be able to prove weaker properties when the human is not supplying annotations, but can more easily be deployed broadly to many programmers.

Program analysis with SL has seen a lot of attention. At first, analysis was formulated for simple linked lists [20], and progressively researchers moved on to more involved data structures. A practical high point in this line of work was the verification of pointer safety in Linux and Windows device drivers up to 10k LOC by the *SpaceInvader* program analyzer [43]. *SpaceInvader* was an academic tool; its sibling, *SLAyer* [3], developed in parallel at Microsoft, was used internally to find 10s of memory safety errors in Windows device drivers. *SpaceInvader* and *SLAyer* were able to analyze complex, linear data structures: for example, one Windows driver manipulated five cyclic doubly linked linked lists sharing a common header node, three of which had acyclic sublists.

Like much research in verification-oriented program analysis these techniques worked in a whole-program fashion: you start

from `main()` or other entry points and explore the program graph, perhaps visiting procedure bodies multiple times. This can be expensive. While accurate analysis of 10k LOC can be a leading research achievement, 10k is tiny compared to software found in the wild. A single company can have tens of millions of lines of code. Progress towards big code called for a radical departure.

8. Bi-Abduction and Facebook Infer

In 2008 Calcagno asked: what is the main obstacle blocking application of SpaceInvader and similar tools to programs in the millions of LOC? O'Hearn answered: the need for the human to supply preconditions. He proposed that a "truly modular" analysis based on local reasoning could accept a program component with no human annotations, and generate a pre/post spec where the precondition approximates the footprint. The analysis would then "stitch" these specifications together to obtain results for larger program parts. The analysis would be compositional, in that a spec for a procedure could be obtained without knowing its callers, and the hypothesis was that it would scale because procedures could be visited independently. This implied giving up on whole-program analysis.

Calcagno, O'Hearn, Distefano and Yang set to work on realizing a truly modular analysis. Yang developed a scheme based on gleaning information from failed proofs to discover a footprint. Distefano made a breakthrough on the stitching issue for the modular analysis, that involved a new inference problem:

> *Bi-abduction*: given A and B, find $?\mathit{frame}$ and $?\mathit{anti\text{-}frame}$ such that
>
> $$A * ?\mathit{anti\text{-}frame} \vdash B * ?\mathit{frame}$$

where $\vdash$ is read 'entails' or 'implies'. The inference of $?\mathit{frame}$ (the leftover part in A but not B) was present in Smallfoot, and is used in many tools. The $?\mathit{anti\text{-}frame}$ part (the missing bit needed to establish B), is *abduction*, or inference of hypotheses, an inference problem identified by the philosopher Charles Peirce in his conceptual analysis of the scientific method. As a simple example,

$$(x \mapsto -) * ?\mathit{anti\text{-}frame} \vdash (y \mapsto -) * ?\mathit{frame}.$$

can be solved with $?\mathit{anti\text{-}frame} = y \mapsto -$ and $?\mathit{frame} = x \mapsto -$.

With bi-abduction we can automate the local reasoning idea by abducing assertions that describe preconditions, and using frame inference to keep specifications small. Let us illustrate with the program we started the paper with. We begin symbolic execution with nothing in the precondition, and we ask a bi-abduction question, using the current state `emp` as the A part of the bi-abduction query and the pre of the small axiom for $[x] = y$ as B.

- Execution state: $\{\texttt{emp}\}[x] = y;\ [y] = x\ \{???\}$
- Bi-abduction query: $\texttt{emp} * ?\mathit{anti\text{-}frame} \vdash x \mapsto - * ?\mathit{frame}$
- Solution: $?\mathit{anti\text{-}frame} = x \mapsto -$; $?\mathit{frame} = \texttt{emp}$.

Now, we move the abduced anti-frame to the overall precondition, we take one step of symbolic execution using the small axiom for Pointer Write from Figure 2, we install the post of the small axiom as the pre of the next instruction, and we continue.

- Execution state: $\{x \mapsto -\}[x] = y\ \{x \mapsto y\}\ [y] = x\ \{???\}$
- Bi-abduction query: $x \mapsto y * ?\mathit{anti\text{-}frame} \vdash y \mapsto - * ?\mathit{frame}$
- Solution: $?\mathit{anti\text{-}frame} = y \mapsto -$; $?\mathit{frame} = x \mapsto y$.

The formula $y \mapsto -$ in the bi-abduction query is the precondition of the small axiom for the pointer write $[y] = x$: we abduce it as the anti-frame, and add it to the overall precondition. The Frame Rule tells us that the inferred frame $x \mapsto y$ is unaltered by $[y] = x$, when it is separately conjoined with $y \mapsto -$, and this with the small axiom gives us our overall postcondition in

$$\{x \mapsto - * y \mapsto -\}[x] = y;\ [y] = x\ \{x \mapsto y * y \mapsto x\}$$

So, starting from specifications for primitive statements, we can infer *both* a precondition and a postcondition for a compound statement by repeated applications of bi-abduction and the Frame Rule. This facility leads to a high degree of automation. Also, note that the precondition here is more general than the one at the start of the paper, because it does not mention 0. Bi-abductive analysis not infrequently finds more general specifications than a top-down analysis that dives into procedures at call sites; finding general specs is important for both scalability and precision.

The main bi-abduction paper [12] contributed proof techniques and algorithms for abduction, and a novel compositional algorithm for generating pre/post specs of program components. Experimental results scaled to hundreds of thousands of lines, and a part of Linux of 3M lines. This form of analysis finds preconditions supporting safety proofs of clusters of procedures as well as indicating potential bugs where proofs failed.

This work led to the program proof startup Monoidics, founded by Calcagno, Distefano and O'Hearn in 2009. Monoidics developed and marketed the Infer tool, based on the abductive tehnique. Monoidics was acquired by Facebook in 2013 at which point Calcagno, Distefano and O'Hearn moved to Facebook with the Monoidics engineering team (www.fbinfer.com).

The compositional nature of Infer turned out to be a remarkable fit for Facebook's software development process [11]. A codebase with millions of lines is altered thousands of times per day in "code diffs" submitted by the programmers. Instead of doing a whole-program analysis for each diff, Infer analyzes changes (the diffs) compositionally, and reports regressions as a bot participating in the internal code review process. Using bi-abduction, the Frame Rule picks off (an approximation of) just enough state to analyze a diff, instead of considering the entire global program state.. The way that compositional analysis supports incremental diff analysis is even more important than the ability to scale; a linear-time analysis operating on the whole program would usually be too slow for this deployment model. Indeed, Infer has evolved from a standalone SL-based analyzer to a general framework for compositional analyses (http://fbinfer.com/docs/checkers.html and appendix).

9. Conclusion

Some time during 2001, while sitting together in his back garden, Reynolds turned to me and exclaimed: "the logic is nice, but it's the model that's *really* important." My own prejudice for semantics made me agree immediately. We were both beguiled by the fact that this funky species of logic could be described using down-to-earth computer science concepts like RAMs and access bits.

What happened later came as a surprise. The specific heap/RAM model gave way in importance to a more general class of non-standard models based on fictional rather than down-to-earth separation. And the logic itself, particularly its proof theory, turned out to be extremely useful in automatic verification, leading to many novel research tools and eventually to Facebook Infer.

Still, I expect that in the long run it will be the spirit rather than the letter of SL that is more significant. Concepts of frames, footprints and separation as a basis for modular reasoning seem to be of fundamental importance, independently of the syntax used to describe them. Indeed, one of the more important directions that I see for further work is in theoretical foundations that get at the essence of scalable, modular reasoning in as formalism-independent a way as possible. Theoretical synthesis would be extremely useful for three reasons: (i) to make it easier for people to understand what has been achieved by each new idea; (ii) to provide a simpler jumping-off point for future work than the union of the many spe-

cific advances; and (iii) to suggest new, unexplored avenues. Hoare has been advancing an abstract, algebraic theory related to CSL, which has components covering semantics, proof theory and testing [24], and work along these lines is well worth exploring further. Other relevant reference points are works on general versions of SL [13, 17], abstract interpretation [15], and work on "Separation without SL" discussed in the appendix. Semantic fundamentals would be crucial to an adequate general foundation, but I stress that proof theoretic and especially algorithmic aspects addressing the central problem of scale should be covered as well.

In conclusion, scalable reasoning about code has come a long way since the birth of SL around the turn of the millennium, but it seems to me that much more is possible both in fundamental understanding and in mechanized techniques that help programmers in their daily work. I hope that scientists and engineers will continue to innovate on the fascinating problems in this area.

ACKNOWLEDGEMENTS. This paper is dedicated to the memory of John C Reynolds (1935-2013). Our work together at the formative stage of Separation Logic was incredibly intense, exciting, and huge fun. I am fortunate to have worked so closely with such a brilliantly insightful scientist, who was also a valued friend.

I thank my many other collaborators in the development of this research, particularly David Pym, Hongseok Yang, Richard Bornat, Cristiano Calcagno, Josh Berdine, Dino Distefano, Steve Brookes, Matthew Parkinson, Philippa Gardner and Tony Hoare. Finally, thanks to my colleagues at Facebook for our work together and for teaching me about applying logic in the real world.

References

[1] A. W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO, volume 4111 of LNCS*, pages 115–137, 2005.

[3] J. Berdine, B. Cook, and S.Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In *CAV*, pages 178–183, 2011.

[4] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221, 2015.

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

[6] R. Bornat. Proving pointer programs in Hoare logic. In *MPC, volume 1837 of LNCS*, pages 102–126, 2000.

[7] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.

[8] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[9] S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.

[10] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7(1):23–50, 1972.

[11] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11, 2015.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011. Preliminary version in POPL'09.

[13] C. Calcagno, P. W. O'Hearn, and H.Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.

[14] H. Chen, F. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37, 2015.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[16] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[17] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.

[18] T. Dinsdale-Young, M. Dodds, M. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[19] T. Dinsdale-Young, P. Gardner, and M. J. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, pages 199–215, 2010.

[20] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[21] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

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

[23] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*. Academic Press, 1972.

[24] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[25] A. Hobor and J. Villard. The ramifications of sharing in data structures. In *40th POPL*, pages 523–536, 2013.

[26] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[27] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[28] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. *PACMPL 1(POPL), 2018*, 2018.

[29] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, pages 696–723, 2017.

[30] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

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

[32] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[33] M. J. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[34] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *21th LICS*, pages 137–146, 2006.

[35] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with verifast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97, 2014.

[36] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoret. Comp. Sci.*, 315(1):257–305, 2004.

[37] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, Cornerstones of Computing. Palgrave Macmillan, 2000.

[38] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[39] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *36th PLDI*, pages 77–87, 2015.

[40] A. M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge*, pages 67–69, 1949.

[41] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. CAV, 2016.

[42] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, 2001.

[43] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

[44] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS*, pages 402–416, 2002.

A. Appendix on Mechanized Reasoning

There has been a significant amount of research on tools that use Separation Logic, only a small selection of which we were able to reference in the main body of the paper. Tools are the main vehicle for practical impact of the theory, and yet there is no comprehensive survey to point to on tools work. This appendix fills in some of the context by providing additional references as well as a brief summary of developments in the main lines of work.

A.1 Mostly Automatic Verification

The Smallfoot tool referenced in the paper follows a style of verification where a program procedure is annotated with loop invariants and pre/post specs: then the rules of program logic can be used to reduce its correctness to a number of theorem prover calls which ask whether entailments of the form $A \vdash B$ are valid. This type of verification tool was pioneered by James King in the early 1970s [45]; prominent examples of such tools include *ESC Java* (https://en.wikipedia.org/wiki/ESC/Java), *VCC* (https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/) and *SPARK* (https://www.adacore.com/sparkpro). Smallfoot attempted to do similar work with a fragment of SL, to explore whether the logic's simple by-hand proofs could be of benefit in an automatic setting.

Smallfoot did automatic verification for a fragment of SL by defining a symbolic execution mechanism [46] to generate prover calls, which fell within a decidable fragment of the logic. For example, the following specifies that if you start with two acyclic linked lists occupying separate memory then a pointer to an acyclic list is returned.

```
1  list_append(x,y) [Pre: list(x) * list(y)] {
2   local t;
3   if (x == NULL) { x = y
4   } else {
5     t = x; n = t->tl;
6     while (n != NULL)
7        [Inv: lseg(x,t) * t |-> n * list(n)]
8      {  t = n;  n = t->tl;
9      }
10     t->tl = y;
11  } /* lseg(x,t) * t |-> y * list(y) */
12 } [Post: list(x)]
```

Here, the loop invariant does not include $list(y)$, where with prior tools it would be needed: Smallfoot guesses that this is a frame for the loop. Smallfoot's theorem prover also needs to establish the entailment $lseg(x,t) * t \mapsto y * list(y) \vdash list(x)$ at the end of the loop, automatically, without enumeratively searching possible induction hypotheses when reasoning about linked list predicates. ($lseg(x,t)$ here is a predicate for linked list segments, and is Smallfoot's notation for what we wrote $list(x,t)$ in the main body of the paper.)

Research on mostly automatic verification with SL has proceeded in several directions.

Richer programming features. Smallfoot considered a toy programming language, designed to be close to the theory of SL. Subsequent tools have targeted real languages and varied language features, for instance: C [47], Java [48], JavaScript [49], message-passing concurrency [50], GPGPU programs [51] and fine-grained concurrency [52, 53].

Richer assertions. Smallfoot only included fixed hardwired inductive predicates for simple linked lists and trees. Furthermore, it did not reason about the contents of the data structures, concentrating instead on their shapes and on showing memory safety properties of programs. How far one can go in relaxing these restrictions is not obvious as retaining decidability is challenging. Indeed, even propositional SL is undecidable when it includes $-\!\!*$ [54]. But there have been impressive advances incorporating general classes of inductive definitions [55, 56].

Integration with classical theorem proving. Many of the works on automating SL use specialized SL proof rules for reasoning about heap data, such as

$$\text{Subtraction}\quad \frac{A \vdash B}{A * H \vdash B * H} \qquad \text{Abstraction}\quad \frac{}{x \mapsto y * y \mapsto \texttt{nil} \vdash \texttt{lseg}(x, \texttt{nil})}$$

Subtraction rules make assertions spatially smaller, where Abstraction rules forget information (e.g., we forget we have a list of length 2, and remember that we have a list of some arbitrary length). Berdine and Calcagno initially devised a proof theory for whereby iterated application of Abstraction and Subtraction rules like these would eventually reduce theorem prover questions about heap shape to "pure" facts which don't describe the shape of the heap [46]. They implemented their own prover for a certain kind of pure fact (simple equalities and disequalities), but this sort of proof theory lends itself as well to using a classical automatic theorem prover, such as those based on SMT (Satisfiability Modulo Theories), for richer types of pure facts such as about integers [48, 57, 55]. The VeriFast tool mentioned in the body of the paper is also of this variety.

Instead of hybrid reasoning mixing application of SL proof rules with calls to a classical prover, there is also the option to embed a fragment of SL entirely into a decidable fragment of first-order logic. The most obvious encodings make heavy use of existential quantifiers, by mapping the SL semantics of $*$ to formulae existentially quantifying over heaps, but existentials cause difficulty for automatic theorem provers. This originally led some researchers to the conclusion that SL was difficult to automate, while it led others to pursue approaches based on proof theory. Completeness properties of proof-theoretic procedures are linked to semantic control over existentials by means of logic restrictions [58, 46]. And controlling what the existentials can do in turn opens up the possibility of more effective direct encodings into SMT [59], leading to the automatic verification of many small C programs [47].

Separation without SL. A related strand of work has developed logics aimed at bringing the same benefits as regards modular reasoning as SL, but in a way that meshes more directly with classical first-order theorem proving. Some logics axiomatize the concept of footprint directly, in a sense formalizing a relative of SL's semantics [60, 61]. Other logics aim to preserve SL's implicit treatment of footprints (the precondition mentions all the resources used by a program fragment) [62], and they employ a relative of the Frame Rule, but are not identical to existing Separation Logics [63]. Dafny is a verification framework based on the former "explicit footprints" approach, and Viper [64] is a verification framework based on "implicit footprints" which has been used to host several verifiers based both on SL and on other related logics.

A.2 Interactive Verification

In interactive verification the human participates more directly in finding proofs. Interactive verification can be hard work, but strong results can be proven, as illustrated by the referenced applications [39, 4, 14, 41] in the main body of the paper.

Proof assistants such as Coq and HOL4 come with native type theories that can represent powerful higher-order set theories. One way to embed a logic like SL is by encoding its semantics in the type theory of the logic (what is called a "shallow" embedding), with the rules of SL then appearing as lemmas proven in the

proof assistant's base logic. For instance, we could define a binary operator $[\![*]\!]$ on a powerset of states. A basic lemma such as that $*$ is commutative can then be proven by set-theoretic reasoning in the semantics, and adopted as a lemma which can be reused by proof tactics in the proof assistant. Likewise, the Frame Rule and other rules of SL are proven as lemmas.

The earliest embeddings of SL in proof assistants followed the approach of the original heap model of SL closely [65, 66]: they would define $State$ as consisting of partial functions from locations to values. Then, researchers began to adopt a more abstract point of view [67], employing variations on Pym's resource semantics, where $State$ is assumed to be any set with certain structure such as that of a partial commutative monoid. Nowadays it is common to develop a very general form of SL inside a proof assistant, and then to specialize to applications by either picking a particular monoid or considering axioms on such monoids as the model of separation [29].

This ability to "pick a monoid" conveys an important additional level of flexibility compared to the automatic reasoning systems which, thus far, have essentially had to bake a particular monoid in. Bringing more automation to the "pick a monoid" step is a direction for future work.

An interesting point is that, given a (shallow) embedding of a logic in a proof assistant, the higher-order features of the host logic transfer to the embedded logic, and proceeding as above so one automatically obtained a "higher-order separation logic" to work with. Higher-order logic and type theory have been repeatedly shown to be useful for reasoning about data abstraction, and this carries over to SL. A consequence is summarized as follows in a paper of Chlipala on his system Bedrock:

> Verifications based on automated theorem proving have omitted reasoning about first-class code pointers, which is critical for tasks like certifying implementations of threads and processes [68]

This aspect of reasoning in higher-order logics is prominent in a line of work starting with the CAP project at Yale, an early example being a verification framework for interrupts [69] and a more recent development the verification of the industrial microkernel referred to in the main body of the paper [41].

Program proofs that require involved non-heap reasoning benefit from the existing theories and proofs often hosted in a proof assistant. For example, the proof of HMAC in [4] used a Coq library, the Foundational Cryptography Framework, to specify cryptographic primitives, and SL reasoning to connect crypto properties to the implemented C code.

One of the benefits often quoted for verification with proof assistants is that they are "foundational", meaning the soundness of the proof systems represented as lemmas reduces in a mechanized way to an assumed, foundational set/type theory. This foundational point of view supports meta-theoretic work. New logics are often developed in a way that mechanically verifies their soundness [1, 29]. Verified verifiers can be produced, such as verified versions of Smallfoot [70, 71]. The metatheory of a representative fragment of Rust [28], mentioned in the main paper, is a good example of an output of this line of work.

When doing interactive proofs there are a number of issues to balance which affect both generality and the degree of automation. The effectiveness of automated proof tactics, the style of semantic encoding, and the level of abstraction of proof steps all play off against one another. These issues are tackled to varying degrees and in different ways in almost all the referenced works in this section; see the recent work on "Iris Proof Mode" [72] and the earlier paper on structuring proofs [67] for an overview of the problems as well a comparison of different approaches.

A.3 Automatic Program Analysis

Where an interactive verifier asks more of the tool user in the way of direction than a mostly automatic verifier, an automatic program analyzer asks for less. In the extreme case an analysis tool can act on bare code, without any user annotations whatsoever. But most research papers assume at least that a precondition is given, like at the beginning of the `list_append` program; then, the job of the analyzer is to infer any loop invariants and a postcondition. (Instead of a precondition a tool might instead assume a main program, or a verification analogue of the concept of a test harness.)

The early 2000s saw a surge in interest in verification-by-static-analysis, with prominent success stories such as SLAM's verification of temporal safety properties in Windows device drivers [73] and ASTRÉE's proof of the absence of run-time errors in Airbus code [74]. But, most practical tools for verification-oriented static analysis ignored pointer-based data structures, or used coarse models that are insufficient to prove basic properties of them; for example, ASTRÉE works only on input programs that do not use dynamic allocation, and SLAM *assumes* memory safety (it thus misses memory safety bugs). Accurate treatment of the heap was considered a major open problem in practical verification. The first papers on SL appeared around the same time and there was immediate interest in using it to address this problem.

The first SL-based program analyses discovered loop invariants for separating conjunctions of list predicates and points-to facts, starting from a given precondition, they worked for a toy programming language similar to Smallfoot's, and they were used to certify memory-safety properties of small programs only [20, 75]. They can be seen as abstract interpreters [15], where the states of the interpreter are SL formulae and loop invariants (fixed points) are calculated using Abstraction proof rules like the one in the previous subsection. This work represented a new foray into shape analysis [76], a known hard problem in static analysis, which caused great challenges for both precision and scaleability: Most research papers at the time reported results for illustrative programs only in the 10s or, rarely, hundreds of LOC.

After these initial results, SL-based program analyses were gradually extended in a way that roughly mirrored advances in SL-based theorem proving. Larger ranges of inductive predicates [77], or data such as arithmetic [78], or both [79] were considered. Remarkably, recent works discover invariants automatically for graph algorithms exhibiting general sharing [80], a non-trivial distance from the simple list programs analyzed in the beginning works. Some of the the more significant examples of reasoning about programs with sharing were given in proofs of memory safety of the Linux deadline IO scheduler and AFS server, programs that use red-black trees overlaid with doubly-linked lists [81].

In most cases, the human needs to at least provide the predicate definitions an analysis will use before getting started, though there have been some attempts to infer predicate definitions [82, 83]. The practical verification results on device drivers discussed in the main paper with SLAyer and SpaceInvader relied on a restricted form of predicate synthesis concerning the layout of data within a (possibly doubly) linked list [84]. That is, enough automation was built in to cover the data structures found in a collection of device drivers rather than only one, but not so much automation as to be able to approach general data structures in general programs (this remains out of reach).

Although we have described forwards-running program analyses above, it is possible in principle to consider algorithms which run backwards, from post to pre. Both backwards and forwards algorithms were emphasised from the very beginning of the program analysis field, such as in the founding work of Kildall on data flow analysis [85]. However, while possible in principle backwards analysis for heap manipulation has proven to be difficult to do with non-

trivial precision, because the number of aliasing possibilities leads to an explosion. One might view the abductive program analysis algorithm of [12] as a way to infer preconditions while still going forwards, thus avoiding the explosion that had been encountered with backwards; indeed, a backwards analysis had been attempted by the authors of [12], but it was abandoned for efficiency reasons.

While pure backwards heap analysis has proven challenging, backwards reasoning can be used effectively to refine an abstraction *after* a forwards run finds initial, coarse information (and avoids the aliasing explosion). Forwards-backwards alternation is a standard technique in abstract interpretation [15], and it has been used in a hybrid SL/SMT analysis that reasons about shape using SL and properties of data using the logical technique of interpolation [86]; it can be used, for instance, to discover that a program preserves sortedness of linked lists.

The compositional nature of the bi-abduction analysis is fundamental for the deployment of Facebook Infer against a large, rapidly changing codebase. The essential ideas behind this analysis method are, however, more general and, as mentioned in the main body of the paper, Infer now has a framework for compositional analyses, named Infer.AI because of its use of abstract interpretation (http://fbinfer.com/docs/checkers.html). Alongside the bi-abduction analysis are several others that share its compositionality and incremental deployment, and they use the footprint idea to obtain succinct and general specs, but they do not literally use SL. For example, the data race detector RacerD [87] calculates a collection of memory accesses for each procedure (analogous to the footprint), and it performs sequential reasoning related to that of CSL, but it does not literally use CSL. This illustrates our point in the paper about intuitions concerning modular reasoning being formalism-independent, as well as the need for a general foundation explaining these and other instances.

Most of the SL-based analysis and automatic verification work has targeted proving safety properties of programs (such as memory safety), or finding safety bugs. But SL has been used as well for liveness properties such as in program termination analysis [88] and in automatic parallelization [89]. These advances then fed into the synthesis of hardware circuits from code [90, 91]. Note that this is distinct from program synthesis, which seeks to generate programs from specifications; the latter has, however, also seen SL developments, building on the previously referenced work on the Dryad verification framework for C programs [92]. Of course, there is then the eventual prospect of composing hardware and software synthesis, by first synthesizing a program from a spec, then further analyzing the synthesized program to obtain hardware.

I am grateful to Josh Berdine for advice on the material in this appendix.

Additional References for Appendix

[45] J. C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.

[46] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[47] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*, pages 440–451, 2014.

[48] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[49] J. F. Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner. JaVerT: JavaScript Verification Toolchain. *PACMPL 1(POPL), 2018*, 2018.

[50] J. Villard, E.Lozes, and C. Calcagno. Tracking heaps that hop with Heap-Hop. In *TACAS*, pages 275–279, 2010.

[51] S. Darabi, S. C. C. Blom, and M. Huisman. A verification technique for deterministic parallel programs. In *NASA Formal Methods Symposium*, pages 247–264, 2017.

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

[53] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal. Caper - automatic verification for fine-grained concurrency. In *ESOP*, pages 420–447, 2017.

[54] J. Brotherston and M. I. Kanovich. Undecidability of propositional separation logic and its neighbours. *J. ACM*, 61(2):14:1–14:43, 2014.

[55] Q. L. Le, M. Tatsuta, J. Sun, and W.-N. Chin. A decidable fragment in separation logic with inductive predicates and arithmetic. In *CAV*, pages 495–517, 2017.

[56] R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, pages 21–38, 2013.

[57] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566, 2011.

[58] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of Separation Logic. In *FSTTCS*, pages 97–109, 2004.

[59] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV*, pages 773–789, 2013.

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

[61] A. Banerjee, D. A. Naumann, and S. Rosenberg. Local reasoning for global invariants, part I: region logic. *J. ACM*, 60(3):18:1–18:56, 2013.

[62] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM TOPLAS.*, 34(1):2:1–2:58, 2012.

[63] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.

[64] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, pages 41–62, 2016.

[65] N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *ICFEM*, pages 400–419, 2006.

[66] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, pages 62–73, 2006.

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

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

[69] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, 2008.

[70] T. Tuerk. A formalisation of Smallfoot in HOL. In *TPHOLs*, pages 469–484, 2009.

[71] A. W. Appel. VeriSmall: verified Smallfoot shape analysis. In *CPP*, pages 231–246, 2011.

[72] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.

[73] 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 *EuroSys*, pages 73–85, 2006.

[74] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, pages 21–30, 2005.

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

[76] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.

[77] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.

[78] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, pages 428–432, 2008.

[79] B.-Y. E. Chang and X. Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.*, pages 161–185.

[80] H. Li, X. Rival, and B.-Y. E. Chang. Shape analysis for unstructured sharing. In *SAS*, pages 90–108, 2015.

[81] O. Lee, H. Yang, and R. Petersen. A divide-and-conquer approach for analysing overlaid data structures. *Formal Methods in System Design*, 41(1):4–24, 2012.

[82] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265, 2007.

[83] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *SAS*, pages 68–84, 2014.

[84] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[85] G. A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[86] A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In *ESOP*, pages 634–660, 2015.

[87] S. Blackshear, N. Gorogiannis, I. Sergey, and P. O'Hearn. RacerD: Compositional static race detection. In *OOPSLA*, 2018.

[88] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV: Computer Aided Verification, 18th International Conference*, 2006.

[89] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, pages 348–362, 2009.

[90] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, pages 205–212, 2009.

[91] F. J. Winterstein, S. R. Bayliss, and G. A. Constantinides. Separation logic for high-level synthesis. *TRETS*, 9(2):10:1–10:23, 2016.

[92] X. Qiu and A. Solar-Lezama. Natural synthesis of provably-correct data-structure manipulations. *PACMPL*, 1(OOPSLA):65:1–65:28, 2017.