DOI:10.1145/3211968

Separation logic is a key development in formal reasoning about programs, opening up new lines of attack on longstanding problems.

BY PETER O'HEARN

Separation Logic

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 vassertions 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 2000.[6]

In joint work with John Reynolds and others we developed separation logic (SL) 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 daily by over one billion people.

Separation logic is an extension of Hoare logic, which employs novel logical operators, most importantly the separating conjunction $*$ (pronounced "and

» 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.**

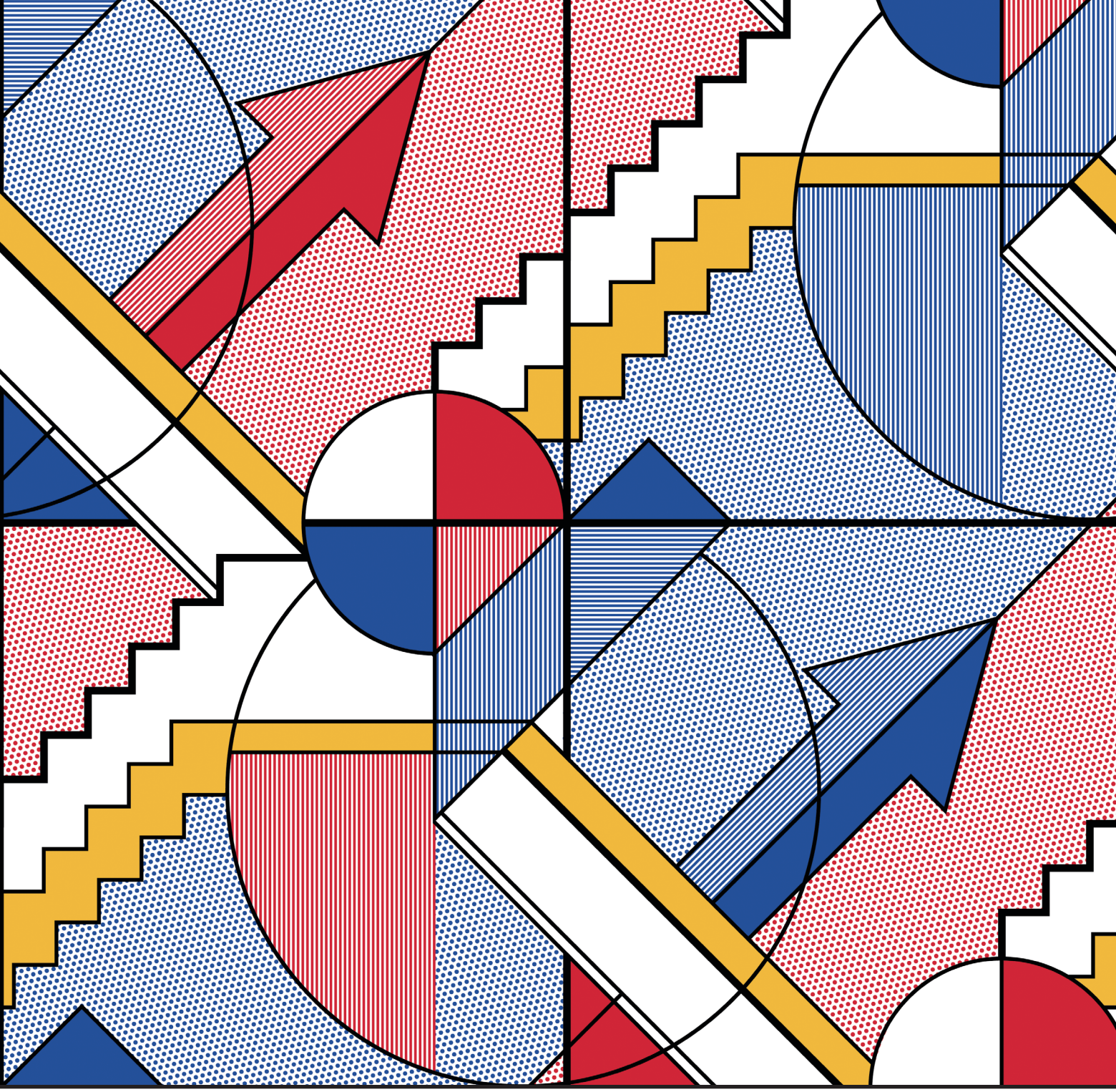


IMAGE BY ANNA GARMATIY

separately") when writing assertions. For example, we might write:

$$\{x \mapsto 0 * y \mapsto 0\}$$
$$[x] = y;$$
$$[y] = x$$
$$\{x \mapsto y * y \mapsto x\}$$

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 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 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 (nonconstructive) variant was possible in which one could reason about explicit memory management (Reynolds had assumed garbage collection). They also introduced the separating implication $-\!\!*$.

Figure 1. Picture semantics.

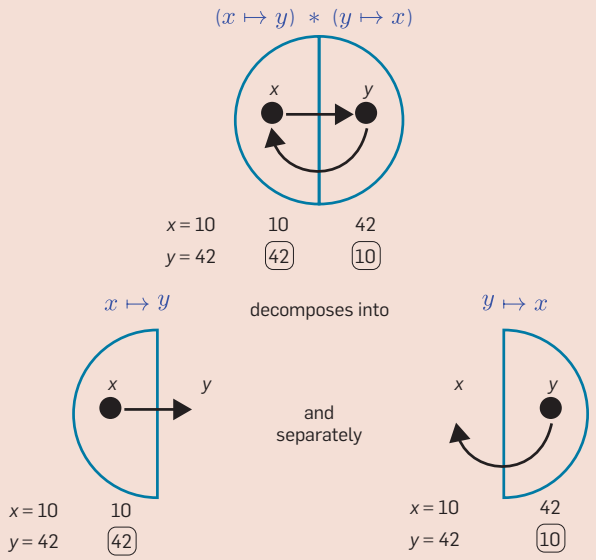


Figure 2. Mathematical semantics.

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

SL 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 do not change. The frame rule is the key to scalable reasoning in SL.

Reynolds' influential survey article summarized the early developments up to 2002.[38] At the end of this early period, O'Hearn circulated a note that proposed a concurrent separation logic (CSL). CSL showed efficient reasoning about threads that share access to storage, proofs that 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,[8,30] the significance of which was summed up as follows:

"For the last 30 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[a]

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

a https://bit.ly/2ywwlpp

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 used at Facebook, Mozilla, Spotify, Amazon Web Services, and other companies (www.fbinfer.com).

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

Separating Conjunction and Implication

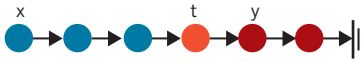
Mathematical semantics has been critical to the discovery and further SL development, but many of the main points can be gleaned from "picture semantics." Consider the first picture in Figure 1. We read the formula at the top of this figure as "x points to y and separately y points to x." Going down the middle of the diagram is a line that represents a heap partitioning: a 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 here 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, $(x \mapsto -) * ((x \mapsto 3) \mathrel{-\!\!*} Q)$ 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, $\mathrel{-\!\!*}$ and $*$ fit together is a way similarly to how implication $\Rightarrow$ and conjunction $\wedge$ do in standard logic. For example, the entailment

$$A * (A \mathrel{-\!\!*} B) \vdash B$$

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

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

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 afterward, and it says this for a small portion of the state in which x is the only active cell.

Figure 3. Separation logic proof system (a selection).

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

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

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 O'Hearn et al.[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 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 Ph.D. thesis.[42] He showed 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 behavior, and their connection to logic,[13,44] 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 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 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, and so on. The SL proofs resemble the box-and-pointer arguments that 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 there is no data race in the parallel program.

Figure 4. Frame and concurrency examples.

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

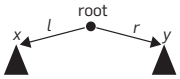
$$\frac{\{x \mapsto 0\}\,[x] = y\,\{x \mapsto y\} \qquad \{y \mapsto 0\}\,[y] = x\,\{y \mapsto x\}}{\{(y \mapsto 0) * (x \mapsto 0)\}\,[x] = y \parallel [y] = x\,\{(y \mapsto x) * (x \mapsto y)\}}$$

Figure 5. `deletetree` example.

(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`}

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.[b] $*$ 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]

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. Here, the rules' more fundamental role as a basis for scalable reasoning is explained.

I 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 diagram 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.

The use of `emp` in the `if` branch of the definition means that *tree*(*r*) is true of a heaplet that 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* ! = 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 inplace 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.

This reasoning uses a tree predicate 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* (τ, *root*) that says 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

$$tree\ (\tau,\ oldroot)\ *\ tree\ (\tau,\ newroot)$$

that says 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 Ph.D. 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

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

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 (for example, a list overlaying a tree), and culminating in the copying algorithm in Cheney's garbage collector.

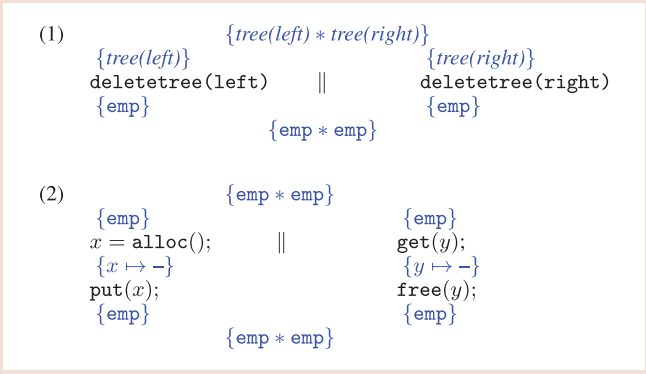
Many papers on SL have avoided $\mathrel{-\!\!*}$, often on the grounds that it complicates automation and is only needed for programs with significant sharing. However, $\mathrel{-\!\!*}$ is recently making something of a comeback. For example, it is used routinely as a basic tool in the Iris higher-order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

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 postcon-

b There are variable conditions in some presentations of SL, that can technically be done away with eliminated by using a version of $*$ that separates variables as well as heap.[34] This article glosses over this issue.

Figure 6. Concurrency proofs.



ditions 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 earlier, 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 *pre*1 ∗ *pre*2) 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 interprocess 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 subproofs of buffer operations here—for that, consult O'Hearn[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 `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 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*:

> "[A]part 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.

Abstraction and the Fiction of Separation

There was considerable work on extending SL after those early papers. 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 major new directions.

▸ In his Ph.D. 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.

Today, 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. 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 Pym et al.[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 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 worked to reimagine 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]

Directions in Mechanized Reasoning

SL spawned new approaches to verification tools. In order to provide a taste of where the field has gone, 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 (https://bit.ly/2CQD9CU).

Mostly automatic verification. Smallfoot,[2] from 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 heap," 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 (https://bit.ly/2CQD9CU).

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 2,832 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 four person years to develop the framework, one-person year to prove the first module, and then the remaining modules, around 900 lines of C, took six 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 received a great deal 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, oneWindows driver manipulated five-cyclic doubly 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 toward big code called for a radical departure.

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 *?frame* and *?anti-frame* such that

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

where $\vdash$ is read 'entails' or 'implies.' The inference of *?frame* (the leftover part in A but not B) was present in Smallfoot, and is used in many tools. The *?anti-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,

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

can be solved with

$?anti\text{-}frame = y \mapsto -$ and $?frame = x \mapsto -$.

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:
  $\texttt{emp} * ?anti\text{-}frame \vdash x \mapsto - * ?frame$
- Solution:
  $?anti\text{-}frame = x \mapsto -\ ;\quad ?frame = \texttt{emp}.$

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:
  $x \mapsto y * ?anti\text{-}frame \vdash y \mapsto - * ?frame$
- Solution:
  $?anti\text{-}frame = y \mapsto -\ ; ?frame = x \mapsto y.$

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

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

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 technique. 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; https://bit.ly/2CQD9CU).

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 nonstandard 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 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: To make it easier for people to understand what has been achieved by each new idea; to provide a simpler jumping-off point for future work than the union of the many specific advances; and, 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 (https://bit.ly/2CQD9CU). 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.

Acknowledgments. ***This article 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. Appel, A.W. *Program Logics for Certified Compilers*. Cambridge University Press, U.K., 2014.
2. Berdine, J. Calcagno, C. and O'Hearn, P.W. Smallfoot: Modular automatic assertion checking with separation logic. *LNCS FMCO 4111* (2005) 115–137, 2005.
3. Berdine, J., Cook, B. and Ishtiaq, S. SLAyer: Memory safety for systems-level code. In *Proceedings of CAV*, 2011, 178–183.
4. Beringer, L., Petcher, A., Ye, K.Q. and Appel, A.W. Verified correctness and security of OpenSSL HMAC. In *Proceedings of 24th USENIX Security Symposium*, 2015, 207–221.
5. Biering, B., Birkedal, L. and Torp-Smith, N. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS 29*, 4 (2007).
6. Bornat, R. Proving pointer programs in Hoare logic. *LNCS MPC 1837* (2000) 102–126.
7. Bornat, R., Calcagno, C., O'Hearn, P.W. and Parkinson, M.J. Permission accounting in separation logic. In *Proceedings of POPL*, 2005, 259–270.
8. Brookes, S. A semantics for concurrent separation logic. Theor. Comput. Sci., 375, 1–3 (2007), 227–270.
9. Brookes, S. and O'Hearn, P.W. Concurrent separation logic. *SIGLOG News 3*, 3 (2016), 47–65.
10. Burstall, R.M. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7*, 1 (1972), 23–50.
11. Calcagno, C. et al. Moving fast with software verification. In *Proceedings of NASA Formal Methods Symposium*, 2015, 3–11.
12. Calcagno, C., Distefano, D., O'Hearn, P.W. and Yang, H. Compositional shape analysis by means of bi-abduction. *J. ACM 58*, 6 (2011), 26. Preliminary version in Proceedings of POPL'09.
13. Calcagno, C., O'Hearn, P.W. and Yang, H. Local action and abstract separation logic. *LICS*, 2007, 366–378.
14. Chen, H., Ziegler, F., Chajed, T., Chlipala, A., Kaashoek, M.F. and Zeldovich, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of SOSP*, pages 18–37, 2015.
15. Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*, 1977, 238–252.
16. Dijkstra, E.W. Cooperating sequential processes. *Programming Languages*, Academic Press, 1968, 43–112.
17. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J. and Yang, H. Views: Compositional reasoning for concurrent programs. In *Proceedings of POPL*, 2013, 287–300.
18. Dinsdale-Young, T., Dodds, M., Gardner, M., Parkinson, M.J. and Vafeiadis, V. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010, 504–528.
19. Dinsdale-Young, T., Gardner, P. and Wheelhouse, M.J. Abstraction and refinement for local reasoning. In *Proceedings of VSTTE*, 2010, 199–215.
20. Distefano, D., O'Hearn, P.W. and Yang, H. A local shape analysis based on separation logic. In *Proceedings of TACAS*, 2006, 287–302.
21. Floyd, R.W. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*. J.T. Schwartz, ed. AMS, 1967, 19–32.
22. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580.
23. Hoare, C.A.R. Towards a theory of parallel programming. *Operati ng Systems Techniques*. Academic Press, 1972.
24. Hoare, T., Möller, B., Struth, G. and Wehrman, I. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program 80*, 6 (2011), 266–296.
25. Hobor, A. and Villard, J. The ramifications of sharing in data structures. In *Proceedings of 40th POPL*, 2013, 523–536.
26. Ishtiaq, S.S. and O'Hearn, P.W. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, 2001, 14–26.
27. Jones, C.B. Specification and design of (parallel) programs. In *Proceedings of IFIP Congress*, 1983, 321–332.
28. Jung , R. Jourdan, J.-H., Krebbers, R. and Dreyer. D. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of PACMPL*, 2018.
29. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J-H, Dreyer, D. and Birkedal, L. The essence of higher-order concurrent separation logic. In *Proceedings of ESOP*, 2017, 696–723.
30. O'Hearn, P.W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375*, 1-3 (2007), 271–307.
31. O'Hearn, P.W and Pym, D.J. The logic of bunched implications. *Bulletin of Symbolic Logic 5*, 2 (1999), 215–244.
32. O'Hearn, P.W., Reynolds, J.C. and Yang, H. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, 2001, 1–19.
33. Parkinson. M.J. Local reasoning for Java. Ph.D. thesis. University of Cambridge, U.K., 2005.
34. Parkinson, M.J., Bornat, R. and Calcagno, C. Variables as resource in Hoare logics. In *Proceedings of 21st LIC*, 2006, 137–146.
35. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B. and Piessens, F. Software verification with verifast: Industrial case studies. *Sci. Comput. Program. 82* (2014), 77–97.
36. Pym, D., O'Hearn, P. and Yang, H. Possible worlds and resources: The semantics of BI. *Theoret. Comp. Sci. 315*, 1 (2004), 257–305.
37. Reynolds, J,C. Intuitionistic reasoning about shared mutable data structure. *Millennial Perspectives in Computer Science, Cornerstones of Computing*. Palgrave Macmillan, 2000.
38. Reynolds, J.C. Separation logic: A logic for shared mutable data structures. *LICS*, 2002, 55–74.
39. Sergey, I., Nanevski, A. and Banerjee, A. Mechanized verification of fine-grained concurrent programs. In *Proceedings of 36th PLDI*, 2015, 77–87.
40. Turing, A.M. Checking a large routine. *Report of a Conference on High-Speed Automatic Calculating Machines*. Univ. Math. Lab., Cambridge, U.K., 1949, 67–69.
41. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H. and Li, Z. A practical verification framework for preemptive OS kernels. In *Proceedings of CAV*, 2016.
42. Yang, H. Local Reasoning for Stateful Programs. Ph.D. thesis. University of Illinois, 2001.
43. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D. and O'Hearn, P.W. Scalable shape analysis for systems code. In *Proceedings of CAV*, 2008, 385–398.
44. Yang, H. and O'Hearn, P.W. A semantic basis for local reasoning. In *Proceedings of FoSSaCS*, 2002, 402–416.

Peter O'Hearn (p.ohearn@ucl.ac.uk) is a research scientist at Facebook and professor of computer science at University College London, U.K.