Program Verification with Separation Logic

Radu Iosif

Program Verification with Separation Logic

Radu Iosif

CNRS/VERIMAG/Université Grenoble Alpes
Radu.Iosif@univ-grenoble-alpes.fr

Abstract. Separation Logic is a framework for the development of modular program analyses for sequential, inter-procedural and concurrent programs. The first part of the paper introduces Separation Logic first from a historical, then from a program verification perspective. Because program verification eventually boils down to deciding logical queries such as the validity of verification conditions, the second part is dedicated to a survey of decision procedures for Separation Logic, that stem from either SMT, proof theory or automata theory. Incidentally we address issues related to decidability and computational complexity of such problems, in order to expose certain sources of intractability.

1 How it all Started

Separation Logic [Rey02] is nowadays a major paradigm in designing scalable modular verification methods for programs with dynamic memory and destructive pointer updates, which is something that most programs written using imperative languages tend to use. The basic idea that enabled the success, both in academia and in industry, is the embedding of a notion of *resource* within the syntax and proof system of the logic, before it's now widely accepted semantics was even defined.

Resources are understood as items, having finite volume, that can be split (separated) among individuals. Since volumes are finite, splitting reduces the resources in a measurable way and cannot be done *ad infinitum*. The story of how resources and separation ended up in logic can be traced back to Girard's *Linear Logic* [Gir87], the first one to restrict the proof rules of *weakening* and *contraction* in natural deduction:

$$\frac{\Gamma \vdash \psi}{\Gamma, \varphi \vdash \psi}\ (W) \qquad\qquad \frac{\Gamma, \varphi, \varphi \vdash \psi}{\Gamma, \varphi \vdash \psi}\ (C)$$

With them, the sequents $\Gamma, \varphi \vdash \psi$ and $\Gamma, \varphi, \varphi \vdash \psi$ can be deduced one from another, but without them, they become unrelated. Removing the (W) and (C) rules leads to two distinct conjunction connectives, illustrated below by their introduction rules:

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (\wedge I) \qquad\qquad \frac{\Gamma \vdash \varphi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \varphi * \psi}\ (*I)$$

While $\wedge$ is the classical conjunction, for which (W) and (C) apply, $*$ is a new *separating* conjunction, for which they don't [OP99].

In natural deduction, conjunction and implication are intertwined, the following introduction rule being regarded as the definition of the implication:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} \ (\rightarrow I)$$

The connection is given by the fact that the comma on a sequent's antecedent is interpreted as conjunction. However, if now we have two conjunctions, we must distinguish them in the antecedents, and moreover, we obtain two implications:

$$\frac{\Gamma; \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi} \ (\rightarrow I) \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \mathrel{-\!\!*} \psi} \ (\mathrel{-\!\!*} I)$$

We use semicolumn and comma for the classical and separating conjunctions, while $\rightarrow$ and $\mathrel{-\!\!*}$ denote the classical and separating implications, respectively. Antecedents are no more viewed as sets but as trees (bunches) with leaves labeled by propositions and internal nodes labeled with semicolumns or commas. Furthermore, $(\mathrel{-\!\!*} I)$ leads to consumption of antecedent nodes and cannot be applied indefinitely. This is where resources became part of the logic, before any semantics was attributed to it.

2 Heaps as Resources

Since the most expensive resource of a computer is the memory, it is only natural to define the semantics of the logic (initially called BI, for *Bunched Implications*) with memory as the resource. There are several memory models and the one on the lowest level views memory as an array of bounded values indexed by addresses.

However, the model which became widespread is the one allowing to reason about the shape (topology) of recursive data structures. In this model, the memory (heap) is viewed as a graph, where nodes represent cells and edges represent pointers between cells, and there is no comparison between cells, other than equality.

This new logic, called *Separation Logic* (SL), is equipped with equality and two atomic propositions, emp for the empty heap and $x \mapsto (y_1, \ldots, y_k)$ meaning that x is the only allocated memory address and there are exactly k pointers from x to $y_1, \ldots, y_k$, respectively. From now on, k is a strictly positive parameter of the logic and SL^k denotes the set of formulae generated by the grammar below:

$$\varphi := \bot \mid \top \mid \mathsf{emp} \mid x \approx y \mid x \mapsto (y_1, \ldots, y_k) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!*} \varphi \mid \exists x \,.\, \varphi$$

SL^k formulae are interpreted over SL-*structures* $I = (\mathfrak{U}, \mathfrak{s}, \mathfrak{h})$, where $\mathfrak{U}$ is a countable set, called the *universe*, the elements of which are called *locations*, $\mathfrak{s} : \mathsf{Var} \rightharpoonup \mathfrak{U}$ is a mapping of variables to locations, called a *store* and $\mathfrak{h} : \mathfrak{U} \rightharpoonup_{\mathit{fin}} \mathfrak{U}^k$ is a finite partial mapping of locations to k-tuples of locations, called a *heap*. We denote by $\mathrm{dom}(\mathfrak{h})$ the domain of the heap $\mathfrak{h}$. A cell $\ell \in \mathfrak{U}$ is *allocated* in I if $\ell \in \mathrm{dom}(\mathfrak{h})$ and *dangling* otherwise.

The notion of separable resources is now embedded in the semantics of SL. Two heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are *disjoint* if and only if $\mathrm{dom}(\mathfrak{h}_1) \cap \mathrm{dom}(\mathfrak{h}_2) = \emptyset$, in which case $\mathfrak{h}_1 \uplus \mathfrak{h}_2$

denotes their union ($\uplus$ is undefined for non-disjoint heaps). A heap $\mathfrak{h}$ is a *subheap* of $\mathfrak{h}'$ if and only if $\mathfrak{h}' = \mathfrak{h} \uplus \mathfrak{h}''$, for some heap $\mathfrak{h}''$. The relation $(\mathfrak{U}, \mathfrak{s}, \mathfrak{h}) \models \varphi$ is defined inductively below:

$$\begin{array}{lll}
(\mathfrak{U},\mathfrak{s},\mathfrak{h}) \models \mathsf{emp} & \Leftrightarrow & \mathfrak{h} = \emptyset \\
(\mathfrak{U},\mathfrak{s},\mathfrak{h}) \models x \mapsto (y_1,\ldots,y_k) & \Leftrightarrow & \mathfrak{h} = \{\langle \mathfrak{s}(x), (\mathfrak{s}(y_1),\ldots,\mathfrak{s}(y_k))\rangle\} \\
(\mathfrak{U},\mathfrak{s},\mathfrak{h}) \models \varphi_1 * \varphi_2 & \Leftrightarrow & \text{there exist disjoint heaps} h_1, h_2 \text{ such that } h = h_1 \uplus h_2 \\
 & & \text{and } (\mathfrak{U},\mathfrak{s},\mathfrak{h}_i) \models \varphi_i, \text{ for } i = 1,2 \\
(\mathfrak{U},\mathfrak{s},\mathfrak{h}) \models \varphi_1 \mathrel{-\!\!*} \varphi_2 & \Leftrightarrow & \text{for all heaps } \mathfrak{h}' \text{ disjoint from } \mathfrak{h} \text{ such that } (\mathfrak{U},\mathfrak{s},\mathfrak{h}') \models \varphi_1, \\
 & & \text{we have } (\mathfrak{U},\mathfrak{s},\mathfrak{h}' \uplus \mathfrak{h}) \models \varphi_2
\end{array}$$

The semantics of equality, boolean and first-order connectives is the usual one and thus omitted. The question is now what can be expressed in SL and what kind of reasoning[1] could be carried out?

First, one defines a single (finite) heap structure up to equality on dangling cells. This is done considering the following fragment of *symbolic heaps*:

$$\begin{array}{l}
\Pi := x \approx y \mid x \not\approx y \mid \Pi_1 \wedge \Pi_2 \\
\Sigma := \mathsf{emp} \mid x \mapsto (y_1,\ldots,y_k) \mid \Sigma_1 * \Sigma_2
\end{array}$$

The Π formulae are called *pure* as they do not depend on the heap. The Σ formulae are called *spatial*. A symbolic heap is a conjunction $\Sigma \wedge \Pi$, defining a finite set of heaps.

For instance, $x \mapsto (y_1,y_2) * y_1 \mapsto (x,y_2) \wedge y_1 \not\approx y_2$ defines cyclic heaps of length two, in which x and y_1 are distinct, x points to y_1, y_1 points to x and both point to y_2. Further, y_2 is distinct from y_1, but could be aliased with x. Observe that $x \mapsto (y_1,y_2) * y_1 \mapsto (x,y_2) \wedge x \approx y_1$ is unsatisfiable, because $x \mapsto (y_1,y_2)$ and $y_1 \mapsto (x,y_2)$ define separated singleton heaps, in which $x \approx y_1$ is not possible.

However, being able to define just bounded heaps is not satisfactory, because one needs to represent potentially infinite sets of structures of unbounded size, such as the ones generated during the execution of a program. Since most imperative programmers are used to working with recursive data structures, a natural requirement is using SL to define the usual recursive datatypes, such as singly- and doubly-linked lists, trees, etc. It turns out that this requires *inductive definitions*. For instance, the following inductive definitions describe an acyclic and a possibly cyclic list segment, respectively:

$$\begin{array}{ll}
\widehat{\mathsf{ls}}(x,y) \leftarrow \mathsf{emp} \wedge x \approx y \ \vee \ \neg(x \approx y) \wedge \exists z \,.\, x \mapsto z * \widehat{\mathsf{ls}}(z,y) & \text{acyclic list segment from } x \text{ to } y \\
\mathsf{ls}(x,y) \leftarrow \mathsf{emp} \wedge x \approx y \ \vee \ \exists u \,.\, x \mapsto u * \mathsf{ls}(u,y) & \text{list segment from } x \text{ to } y
\end{array}$$

Intuitively, an acyclic list segment is either empty, in which case the head and the tail coincide [$\mathsf{emp} \wedge x \approx y$], or contains at least one element which is disjoint from the rest of the list segment. Observe that $x \mapsto z$ and $\widehat{\mathsf{ls}}(z,y)$ hold over disjoint parts of the heap, which ensures that the definition unfolds producing distinct cells. The constraint $\neg(x \approx y)$, in the inductive definition of $\widehat{\mathsf{ls}}$, captures the fact that the tail of the list segment is distinct from every allocated cell in the list segment, which ensures the acyclicity condition. Since this constraint is omitted from the definition of the second (possibly cyclic) list segment $\mathsf{ls}(x,y)$, its tail y is allowed to point inside the set of allocated cells.

[1] We use the term “reasoning” in general, not necessarily push-button automated decision.

As usual, the semantics of inductive definitions is given by the least fixed point of a monotone function between sets of finite SL-structures. To avoid clutter, we omit this definition, but point out that the reasons why symbolic heaps define monotone functions are that (i) $*$'s do not occur negated, and (ii) $-\!\!*$'s are not used.

In fact, one may wonder, at this point, why only $*$'s are used for specification of data structures and what is the rôle of $-\!\!*$ in reasoning about programs? The answer is given in the next section.

3 Program Verification

Program verification means providing a proof for the following problem: given a program P and a set of states Ψ, is there an execution of P ending in a state from Ψ? More concretely, Ψ can be a set of "bad" states in which the program ends after attempting to dereference an unallocated pointer variable, or after leaking memory. A program proof consist in annotating the program with assertions that (i) must hold each time the control reaches an assertion site, such that (ii) the set Ψ is excluded from the reachable states of P.

Ensuring the point (i) requires proving the validity of a certain number of Hoare triples of the form $\{\phi\}\ \mathsf{C}\ \{\psi\}$, each of which amounts to proving the validity of an entailment $\phi \Rightarrow \widetilde{\mathrm{pre}}(C,\psi)$, or equivalently, $\widetilde{\mathrm{post}}(C,\phi) \Rightarrow \psi$, where $\widetilde{\mathrm{pre}}$ and $\widetilde{\mathrm{post}}$ are the weakest precondition and the strongest postcondition predicate transformers. Such entailments are called *verification conditions*.

In the seminal papers of Ishtiaq and O'Hearn [IO01] and Reynolds [Rey02], SL was assigned the purpose of an assertion logic in a Hoare-like logic used for writing correctness proofs of pointer-manipulating programs. This turn from proof theory [OP99] to program proofs removes a long-standing thorn from the side of Hoare logicians, namely that the substitution-based assignment rule $\{\phi[\mathsf{E}/\mathsf{x}]\}\ \mathsf{x} = \mathsf{E}\ \{\phi\}$ [Hoa69] is no longer valid in the presence of pointers and aliasing.

The other problems, originally identified in the seminal works of Floyd [Flo67] and Hoare [Hoa69] are how to accomodate program proofs with procedure calls and concurrency. Surprisingly, the $*$ connective provides very elegant solutions to these problems as well, enabling the effective transfer of this theory from academia to industry.

In general, writing Hoare-style proofs requires lots of human insight, essentially for (i) infering appropriate loop invariants, and (ii) solving the verification conditions obtained from the pre- or postcondition calculus that captures the semantics of a straight-line program. With SL as an assertion language, automation is possible but currently at (what we believe are) the early stages.

For instance, the problem (i) can be tackled using abstract interpretation [CC79], but the (logical) abstract domains currently used are based on the hardcoded $\mathsf{ls}(x,y)$ predicate, making use of specific properties of heaps with a single pointer field, that are composed of lists, and which can be captured by finitary abstractions [BBH$^+$06]. An extension to nested lists (doubly-linked lists of ... of doubly-linked lists) has been since developped in the SPACEINVADER tool [BCC$^+$07] and later shipped to industry in the INFER tool [CD11].

Alternatives to invariant inference using fixed point computations are also possible. These methods use the expressive power of the higher-order inductive definitions of SL and attempt to define inductive predicates that precisely define loop invariants, using only a single symbolic execution pass through the loop [LGQC14]. The difficulty of the verification problem is then shipped to the decision procedure that solves the verification condition thus obtained (ii). This is probably the real problem standing in front of the researchers that aim at developping a fully push-button program verification method for real-life programs, based on SL. We shall survey this issue at large in §4

3.1 While Programs

Perhaps the longest lasting impression after reading Ishtiaq and O'Hearn's paper [IO01] is that a sound and complete weakest precondition calculus for programs with pointers and destructive updates has finally been found. But let us first recall the problem with Hoare's substitution-based assignment rule. Consider the triple

$$\{(\mathsf{y.data} = 2 \wedge \mathsf{x} = \mathsf{y})[1/\mathsf{x.data}]\}\ \mathsf{x.data} := 1\ \{\mathsf{y.data} = 2 \wedge \mathsf{x} = \mathsf{y}\}$$

which is the same as $\{\mathsf{y.data} = 2 \wedge \mathsf{x} = \mathsf{y}\}\ \mathsf{x.data} := 1\ \{\mathsf{y.data} = 2 \wedge \mathsf{x} = \mathsf{y}\}$ because the substitution has no effect on the precondition. The triple is clearly invalid, because the assignment $\mathsf{x.data} := 1$ makes the assertion $\mathsf{x.data} = 2$ false.

The solution provided by SL is of great simplicity and elegance. Since pointer updates alter a small part of the heap, we can "remove" this part using $*$ and "replace" it with an updated heap, using $-\!\!*$, while requiring that the postcondition hold afterwards:

$$\{\exists \mathsf{x} \exists \mathsf{y}\ .\ \mathsf{u} \mapsto (\mathsf{x},\mathsf{y}) * ((\mathsf{u} \mapsto (\mathsf{v},\mathsf{y})) -\!\!* \phi)\}\ \mathsf{u.1} := \mathsf{v}\ \{\phi\}$$

where $\mathsf{u.1} := \mathsf{v}$ denotes the assignment of the first selector of the cell referred to by u to v. We have a similar weakest precondition for memory allocation, where we adopt the more structured object-oriented constructor $\mathsf{cons}(\mathsf{v},\mathsf{w})$ instead of C's $\mathsf{malloc}(\mathsf{n})$:

$$\{\forall \mathsf{x}.\ (\mathsf{x} \mapsto (\mathsf{v},\mathsf{w})) -\!\!* \phi[\mathsf{x}/\mathsf{u}]\}\ \mathsf{u} := \mathsf{cons}(\mathsf{v},\mathsf{w})\ \{\phi\}$$

Observe that this calculus produces preconditions that mix $*$ and $-\!\!*$ with quantifiers. Very early, this hindered the automation of program proofs, because at the time, there was no "*automatic theorem prover which can deal with the form of these assertions (which use quantification and the separating implication)*" [BCO05]. This issue, together with a rather coarse undecidability result for quantified SL formulae [CYO01] made researchers almost entirely forget about the existence of $-\!\!*$ and of the weakest precondition calculus, for more than a decade. During this time, program verifiers used (and still do) incomplete, overapproximating, postcondition calculi on (inductive definitions on top of) $*$-based symbolic heaps. In the light of recent results concerning decision procedures for the base assertion SL language, we believe this difficulties can be overcome. A detailed presentation of these results is given in §4.1.

Before going further, an interesting observation can be made. The weakest precondition of a straight-line program, in which more than one such statement occurs in a sequence, would be a formula in which first-order quantifiers occur within the scope of a separating connective. This could potentially be problematic for automated reasoning because, unlike first order logic, SL formulae do not have a prenex form: $\phi * \forall x \,.\, \psi(x) \not\equiv \forall x \,.\, \phi * \psi(x)$ and $\phi \mathrel{-\!\!*} \exists x \,.\, \psi(x) \not\equiv \exists x \,.\, \phi \mathrel{-\!\!*} \psi(x)$. However, the following notion of *precise formulae* comes to rescue in this case [OYR04]:

Definition 1. *A* SL *formula* ϕ *is* precise *if and only if, for all* SL*-structures* $I = (\mathfrak{U}, \mathfrak{s}, \mathfrak{h})$ *there exists at most one subheap* $\mathfrak{h}'$ *of* $\mathfrak{h}$ *such that* $(\mathfrak{U}, \mathfrak{s}, \mathfrak{h}') \models \phi$.

If ϕ is precise, we recover the equivalences $\phi * \forall x \,.\, \psi(x) \equiv \forall x \,.\, \phi * \psi(x)$ and $\phi \mathrel{-\!\!*} \exists x \,.\, \psi(x) \equiv \exists x \,.\, \phi \mathrel{-\!\!*} \psi(x)$. Moreover, since formulae such as $x \mapsto (y,z)$ are precise, one can hoist the first-order quantifiers and write any precondition in prenex form. As we shall see (§4.1) prenexisation of formulae is an important step towards the decidability of the base assertion language.

3.2 Local Reasoning and Modularity

Being able to provide easy-to-write specifications of recursive data structures (lists, trees, etc.) as well as concise weakest preconditions of pointer updates, were the first salient features of SL. With separating conjunction $*$ as the main connective, a principle of *local reasoning* has emerged:

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 [ORY01].

The rôle of the separating conjunction in local reasoning requires little explanation. If a set of program states is specified as $\varphi * \psi$, and the update occurs only in the φ part, then we are sure that ψ is not affected by the update and can be copied from pre- to postcondition as it is. This is formalized by the following *frame rule*:

$$\frac{\{\varphi\}\ \mathsf{P}\ \{\phi\}}{\{\varphi * \psi\}\ \mathsf{P}\ \{\phi * \psi\}}\ \mathsf{modifies}(\mathsf{P}) \cap \mathsf{var}(\psi) = \emptyset$$

where $\mathsf{modifies}(\mathsf{P})$ is the set of variables whose value is changed by the program P, defined recursively on the syntactic structure of P.

The frame rule allows to break a program proof into small pieces that can be specified locally. However, locality should not be confounded with *modularity*, which is the key to scalability of program verification technique. Indeed, most large, industrial-size programs are built from a large number of small *components*, such as functions (procedures) or threads, in a concurrent setting.

By a *modular program verification* we understand a method capable of inferring specifications of any given program component in isolation, independently on the context in which the other components interact with the component's environment. Then the local specifications are combined into a global one, using the frame rule or a variant thereof. Observe that this is not to be mixed up with program verification algorithms that store and reuse analysis results.

An example of modular program verification with SL is the compositional shape analysis based on the inference of *footprints* [CDOY07]. These are summaries specified as pairs of pre-/postconditions, that guarantee absence of implicit memory faults, such as null pointer dereferences or memory leaks. The important point is that footprints can be inferred directly from the program, without user-supplied pre- or postconditions.

Combining component footprints into a global verification condition is the other ingredient of a modular verification technique. Since footprints are generated without knowledge of their context, sometimes their combination requires some "adjustment". To understand this point, consider an interprocedural analysis in which a function $\mathsf{foo}(\mathsf{x},\mathsf{y})$ is invoked at a call site. The summary of the function, inferred by footprint analysis, is say $\mathsf{ls}(\mathsf{x},\mathsf{z}) * \mathsf{ls}(\mathsf{y},\mathsf{nil})$, using the previously defined inductive predicates (§2). Informally, this sais that there is a list segment from x to z and disjointly a nil-ending list starting with y. Assume further that $\mathsf{x} \mapsto \mathsf{z}$ is the assertion at the call site. Clearly $\mathsf{x} \mapsto \mathsf{z}$ does not entail $\mathsf{ls}(\mathsf{x},\mathsf{z}) * \mathsf{ls}(\mathsf{y},\mathsf{nil})$, in which case a classical interprocedural analysis would give up.

However, an SL-based interprocedural analysis uses the frame rule for function calls and may reason in the following way: find a frame ϕ such that $\mathsf{x} \mapsto \mathsf{z} * \phi \Rightarrow \mathsf{ls}(\mathsf{x},\mathsf{z}) * \mathsf{ls}(\mathsf{y},\mathsf{nil})$. In this case, a possible answer (frame) is $\phi = \mathsf{ls}(\mathsf{y},\mathsf{nil})$. If the current precondition of the caller of $\mathsf{foo}(\mathsf{x},\mathsf{y})$ is φ, we percolate the frame all the way up to the caller and modify its precondition to $\varphi * \phi$, as local reasoning allows us to do. This method is implemented by the INFER analyzer and is used, on an industrial scale, at Facebook [CD11].

This style of modular verification introduces *abductive reasoning* as a way to perform frame inference. In classical logic, the abduction problem is: given an assumption ϕ and a goal ψ, find a missing assumption X such that $\phi \wedge X \Rightarrow \psi$. Typically, we aim at finding the weakest such assumption, which belongs, moreover, to a given language (a disjunction of conjunctive assertions pertaining to a restricted set). If we drop the latter requirement, about the allowed form of X, we obtain the weakest solution $X = \phi \rightarrow \psi$.

Abductive reasoning in SL follows a very similar pattern. An abduction problem is: given assertions ϕ and ψ, find an assertion X such that $\phi * X \Rightarrow \psi$. Similar to the classical case, the weakest solution, in this case is ... $X = \phi \multimap \psi$! So the long-forgotten magic wand comes back to program analysis, this time by way of abduction. However, since decision procedures for SL still have a hard time in dealing with $\multimap$, they use underapproximations of the weakest solutions, that are mostly good enough for the purposes of the proof [CDOY11].

4 Decision Procedures

As mentioned in the introduction of §3, the job of a verifier is turning a program verification problem into a (finite) number of logical entailments of the form $\phi \Rightarrow \psi$, called verification conditions. In this section, we survey how one can establish the validity of such entailments, provided that ϕ and ψ are SL formulae.

If the fragment of SL to which ϕ and ψ belong is closed under negation, the entailment $\phi \Rightarrow \psi$ is valid if and only if the formula $\phi \wedge \neg\psi$ is unsatisfiable. Usually negation is part of basic SL assertions, that do not use inductive definitions. These are mostly discussed in §4.1. In this case, we reduce the entailment to a satisfiability problem, that ultimately, can be solved using SMT technology.

If the logic in which ϕ and ψ are written does not have negation, which is typically the case of inductive definitions built on top of symbolic heaps, we deal with entailments directly, either by proof-theoretic (we search for a sequent calculus proof of the entailment) or automata-theoretic (we reduce to the inclusion between the languages of two automata) arguments. The pros and cons of each approach are discussed in §4.2.

4.1 Basic Logic

Let us consider the language SL^k given in §2. For $k \geq 2$, undecidability of this logic occurs even if separating connectives are almost not used at all. If one encodes an uninterpreted binary relation $R(x,y)$ as $\exists z \,.\, z \mapsto (x,y) * \top$, undecidability occurs as a simple consequence of Trakhtenbrot's result for finite satisfiability of first-order logic [BGG97]. If $k = 1$, the logic is still undecidable, but the fragment of SL^1 without $-\!\!*$ becomes decidable, with a nonelementary recursive complexity lower bound [BDL12].

On the other hand, the quantifier-free fragment of SL^k is PSPACE-complete, for any $k \geq 1$ [CYO01]. The crux of this proof is a small model property of quantifier-free SL^k. If a formula ϕ in this language has a model $(\mathfrak{U}, \mathfrak{s}, \mathfrak{h})$ then it has a model where $||\mathfrak{h}|| = O(\mathsf{size}(\phi))$. This also provides effective algorithms for the satisfiability problem. It is possible, for instance, to encode the quantifier-free SL^k formula in first-order logic with bitvectors and use existing SMT technology for the latter [CGH05], or directly using a DPLL(T)-style algorithm that attempts to build a model of bounded size and learns from backtracking [RISK16].

The quantifier-free fragment of SL^k is also important in understanding the expressiveness of SL, relative to that of classical first- and second-order logics. This point addresses a more fundamental question, relative to the presence of the separating connectives $*$ and $-\!\!*$: is it possible to reason about resources and separation in first-order logic, or does one need quantified relations, as the heap semantics of SL^k suggests?

It turns out that, surprisingly, the entire *prenex fragment* of SL^k can be embedded into uninterpreted first-order logic. This is the set of formulae $Q_1 x_1 \ldots Q_n x_n \,.\, \phi$, where ϕ is quantifier-free. First, we consider a small set of patterns, called *test formulae*, that use $*$ and $-\!\!*$ in very restricted ways:

Definition 2. *The following patterns are called* test formulae*:*

$$x \hookrightarrow (y_1,\ldots,y_k) \stackrel{\mathsf{def}}{=} x \mapsto (y_1,\ldots,y_k) * \top \qquad |U| \geq n \stackrel{\mathsf{def}}{=} \neg(\top -\!\!* \neg(|h| \geq n)),\ n \in \mathbb{N}$$

$$\mathsf{alloc}(x) \stackrel{\mathsf{def}}{=} x \mapsto \underbrace{(x,\ldots,x)}_{k\ times} -\!\!* \bot \qquad |h| \geq |U| - n \stackrel{\mathsf{def}}{=} |h| \geq n+1 -\!\!* \bot, n \in \mathbb{N}$$

$$|h| \geq n \stackrel{\mathsf{def}}{=} \begin{cases} |h| \geq n-1 * \neg\mathsf{emp}, & if\ n > 0 \\ \top, & if\ n = 0 \\ \bot, & if\ n = \infty \end{cases}$$

and $x \approx y$, where $x, y \in \mathsf{Var}$, $\mathbf{y} \in \mathsf{Var}^k$ and $n \in \mathbb{N}_\infty$ is a positive integer or ∞.

Observe first that $-\!\!*$ is instrumental in defining allocation without the use of existential quantifiers, as in $\mathsf{alloc}(x) \stackrel{\mathsf{def}}{=} \exists y_1 \ldots \exists y_n \,.\, x \mapsto (y_1,\ldots,y_k) * \top$. Second, it can express cardinality constraints relative to the size of the universe $|U| \geq n$ and $|h| \geq |U| - n$, assuming that it is finite.

In contrast with the majority of the literature on Separation Logic, here the universe of *available* memory locations (besides the ones occurring in the heap, which is finite) is not automatically assumed to be infinite. In particular, the finite universe hypothesis is useful when dealing with bounded memory issues, for instance checking that the execution of the program satisfies its postcondition, provided that there are sufficiently many available memory cells. Having different interpretations of the universe is also motivated by a recent integration of SL^k within the DPLL(T)-based SMT solver CVC4 [RISK16,RIS17], in which the SL theory is parameterized by the theory of locations, just like the theories of arrays and sets are parameterized by theories of values.

A first nice result is that any quantifier-free SL^k formula is equivalent to a boolean combination of test formulae [EIP18b]. Then we can define an equivalence-preserving translation of the quantifier-free fragment of SL^k into FO. Let $\mathfrak{d}$ be a unary predicate symbol and let $\mathfrak{f}_i$ (for $i = 1, \ldots, k$) be unary function symbols. We define the following transformation from quantified boolean combinations of test formulae into first order formulae:

$$\begin{aligned}
\Theta(x \approx y) &\stackrel{\mathsf{def}}{=} x \approx y \\
\Theta(x \hookrightarrow (y_1, \ldots, y_k)) &\stackrel{\mathsf{def}}{=} \mathfrak{d}(x) \wedge \textstyle\bigwedge_{i=1}^{k} y_i \approx \mathfrak{f}_i(x) \\
\Theta(\mathsf{alloc}(x)) &\stackrel{\mathsf{def}}{=} \mathfrak{d}(x) \\
\Theta(\neg\phi) &\stackrel{\mathsf{def}}{=} \neg\Theta(\phi) \\
\Theta(\phi_1 \bullet \phi_2) &\stackrel{\mathsf{def}}{=} \Theta(\phi_1) \bullet \Theta(\phi_2) \qquad \text{if } \bullet \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
\Theta(\mathrm{Q}x \,.\, \phi) &\stackrel{\mathsf{def}}{=} \mathrm{Q}x \,.\, \Theta(\phi) \qquad \text{if } \mathrm{Q} \in \{\exists, \forall\} \\
\Theta(|U| \geq n) &\stackrel{\mathsf{def}}{=} \exists x_1, \ldots, x_n \,.\, \mathsf{distinct}(x_1, \ldots, x_n) \\
\Theta(|h| \geq n) &\stackrel{\mathsf{def}}{=} \exists x_1, \ldots, x_n \,.\, \mathsf{distinct}(x_1, \ldots, x_n) \wedge \textstyle\bigwedge_{i=1}^{n} \mathfrak{d}(x_i) \\
\Theta(|h| \geq |U| - n) &\stackrel{\mathsf{def}}{=} \exists x_1, \ldots, x_n \forall y \,.\, \textstyle\bigwedge_{i=1}^{n} y \not\approx x_i \rightarrow \mathfrak{d}(y)
\end{aligned}$$

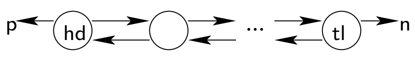
As a result of this translation, any formula of the prenex fragment of SL^k is equivalent to a first-order formula that uses one monadic predicate and k monadic function symbols. Thus, we obtain the decidability of the prenex fragment of SL^1 as a consequence of the decidability of first-order logic with one monadic function symbol and any number of monadic predicate symbols [BGG97]. Moreover, for $k \geq 2$, undecidability occurs even for the quantifier prefix of the form $\exists^*\forall^*$, if universally quantified variables occur under the scope of $\mathrel{-\!\!*}$. However, if this is not the case, $\exists^*\forall^*$ fragment of SL^k becomes PSPACE-complete [EIP18b].

Interestingly, if $k = 1$ again, the $\exists^*\forall^*$ fragment is PSPACE-complete independently of how $\mathrel{-\!\!*}$ is used [EIP18a]. This result points out the difference between the prenex fragment of SL^1 and SL^1 with unrestricted use of quantifiers, which is undecidable: in fact, the full second-order logic can be embedded within it [BDL12]. For program verification, the good news is that, as discussed in §3, the prenex fragment is closed under weakest preconditions, making it possible to verify straight-line programs obtained by loop unfolding, à la Bounded Model Checking.

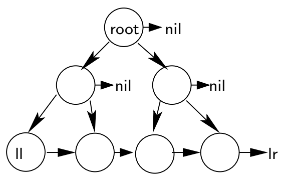
4.2 Inductive Definitions

Let us now turn to the definition of recursive data structures using inductive definitions in SL^k. As a first remark, the base assertion language is usually that of symbolic heaps $\Sigma \wedge \Pi$, where Σ is either emp or a finite $*$-conjunction and Π is either $\top$ or a nonempty conjunction of equalities and disequalities between variables. A system of inductive definitions is a set of rules of the form $p(\mathbf{x}_0) \leftarrow \Sigma \wedge \Pi * p_1(\mathbf{x}_1) * \ldots p_n(\mathbf{x}_n)$, where w.l.o.g. $\mathbf{x}_0, \ldots, \mathbf{x}_n$ are pairwise disjoint sets of variables and $\mathsf{var}(\Sigma \wedge \Pi) \subseteq \bigcup_{i=0}^{n} \mathbf{x}_i$. The examples below show the inductive definitions of a doubly-linked list $\mathsf{dll}(\mathsf{hd}, \mathsf{p}, \mathsf{tl}, \mathsf{n})$ and of a tree with linked leaves $\mathsf{tll}(\mathsf{root}, \mathsf{ll}, \mathsf{lr})$, respectively:

$$\begin{aligned}
\mathsf{dll}(\mathsf{hd},\mathsf{p},\mathsf{tl},\mathsf{n}) &\leftarrow \mathsf{hd} \mapsto (\mathsf{n},\mathsf{p}) \wedge \mathsf{hd} = \mathsf{tl} \\
\mathsf{dll}(\mathsf{hd},\mathsf{p},\mathsf{tl},\mathsf{n}) &\leftarrow \exists \mathsf{x} \,.\, \mathsf{hd} \mapsto (\mathsf{x},\mathsf{p}) * \mathsf{dll}(\mathsf{x},\mathsf{hd},\mathsf{tl},\mathsf{n})
\end{aligned}$$



$$\begin{aligned}
\mathsf{tll}(\mathsf{root},\mathsf{ll},\mathsf{lr}) &\leftarrow \mathsf{root} \mapsto (\mathsf{nil},\mathsf{nil},\mathsf{lr}) \wedge \mathsf{root} = \mathsf{ll} \\
\mathsf{tll}(\mathsf{root},\mathsf{ll},\mathsf{lr}) &\leftarrow \exists \mathsf{l} \exists \mathsf{r} \exists \mathsf{z} \,.\, \mathsf{root} \mapsto (\mathsf{l},\mathsf{r},\mathsf{nil}) * \mathsf{tll}(\mathsf{l},\mathsf{ll},\mathsf{z}) * \mathsf{tll}(\mathsf{r},\mathsf{z},\mathsf{lr})
\end{aligned}$$



A solution of an inductive predicate system is a mapping of predicates to sets of SL-structures and the semantics of a predicate p, denoted $[\![p]\!]$ corresponds to the set of structures assigned to it by the least solution of the system. The *entailment problem* is given a system and two predicates p and q, does $[\![p]\!] \subseteq [\![q]\!]$ hold? In general, this problem is undecidable [AGH$^+$14,IRV14].

Automata-based Techniques As usual, we bypass undecidability by defining a number of easy-to-check restrictions on the system of predicates:

- *Progress*: each rule allocates exactly one node, called the root of the rule. This condition makes our technical life easier, and can be lifted in many cases — rules with more than one allocation can be split by introducing new predicates.
- *Connectivity*: for each inductive rule of the form $\Sigma * p_1(\mathbf{x}_1) * \ldots * p_n(\mathbf{x}_n) \wedge \Pi$, there exists at least one edge between the root of the rule and the root of each rule of p_i, for all $i \in [1,n]$. This restriction prevents the encoding of context-free languages in SL, which requires disconnected rules, leading to undecidability.
- *Establishment*: all existentially quantified variables in a recursive rule are eventually allocated. This restriction is not required for the satisfiability problem, but it is essential for entailment.

The fragment of SL obtained by applying the above, rather natural, restrictions, is denoted $\mathsf{SL}_{\mathsf{btw}}$ in the following. The proof of decidability for entailments in $\mathsf{SL}_{\mathsf{btw}}$ relies on three main ingredients:

1. for each predicate p in the system, all heaps from the least solution $[\![p]\!]$ are represented by graphs, whose *treewidth is bounded* by a linear function in the size of the system.
2. we define, for each predicate p, a formula Φ_p in *monadic second-order logic* (MSO) of graphs whose models are exactly the graphs encoding the heaps from the least solution $[\![p]\!]$.
3. the entailment problem $[\![p]\!] \subseteq [\![q]\!]$ is reduced to the satisfiability of an MSO formula $\Phi_p \wedge \neg\Phi_q$. Since all models of p (thus of Φ_p) have bounded treewidth, this problem is decidable, by Courcelle's Theorem [Cou90].

This approach suggests that a direct encoding of the least solution of a system of inductive definitions is possible, using tree automata. That is, for each predicate p, we define a tree automaton A_p that recognizes the set of structures from $[\![p]\!]$, and the entailment problem $[\![p]\!] \subseteq [\![q]\!]$ reduces to a language inclusion problem between tree automata $\mathcal{L}(A_p) \subseteq \mathcal{L}(A_q)$.

However, there are instances of the entailment problem that cannot be directly solved by language inclusion between tree automata, due to the following *polymorphic representation problem*: the same set of states can be defined by two different inductive predicates, and the tree automata mirroring their definitions will report that the entailment does not hold. For example, doubly-linked lists can also be defined in reverse:

$$\mathsf{dll}_{\mathrm{rev}}(\mathsf{hd},\mathsf{n},\mathsf{tl},\mathsf{p}) \equiv \mathsf{hd} \mapsto (\mathsf{p},\mathsf{n}) \wedge \mathsf{hd} = \mathsf{tl} \ \vee\ \exists \mathsf{x} \,.\, \mathsf{tl} \mapsto (\mathsf{x},\mathsf{n}) * \mathsf{dll}_{\mathrm{rev}}(\mathsf{hd},\mathsf{tl},\mathsf{x},\mathsf{p})$$

A partial solution is to build tree automata only for a restriction of $\mathsf{SL}_{\mathsf{btw}}$, called the *local fragment* ($\mathsf{SL}_{\mathsf{loc}}$). The structures that are models of predicates defined in this fragment have the following nice property: whenever the same structure is encoded by two different spanning trees, the two trees are related by a *rotation* relation, which changes the root to an arbitrary internal node, and the orientation of the edges from the path to the root to that node. We can thus check $[\![p]\!] \subseteq [\![q]\!]$ in this fragment by checking the inclusion between A_p and A_q^{rot}, where A_q^{rot} is automaton recognizing the closure of the languge of A_q under rotation. Moreover, since the rotation closure of a tree automaton is possible in polynomial (quadratic) time, the entailment problem for $\mathsf{SL}_{\mathsf{loc}}$ can be shown to be $\mathsf{EXPTIME}$-complete [IRV14]. It is still an open question, whether this tight complexity bound applies to the entire $\mathsf{SL}_{\mathsf{btw}}$ logic.

Proof-based Techniques An experimental evaluation of SL inductive entailment solvers, carried out during the SL-COMP 2014 solver competition [SC14], shows the strengths and weaknesses of automata-based versus proof-based solvers. On one hand, automata-based solvers can cope with polymorphic representations in a much better way than proof-based solvers, which require complex cut rules, whose completness is far from being understood [CJT15].

On the other hand, proof-based solvers are more flexible in deadling with extensions of the base theory (symbolic heaps), such as arbitrary equalities and disequalities or even (limited forms of) data constraints. Moreover, they can easily interact with SMT solvers and discharge proof obligations belonging to the base logic. However, as the example below shows, this requires quantifiers.

Consider a fragment of the inductive proof showing that any acyclic list segment is also a list segment, given below:

$$\dfrac{\dfrac{\widehat{\mathsf{ls}}(z,y) \vdash \mathsf{ls}(z,y)}{\neg(x \approx y) \wedge x \mapsto z * \widehat{\mathsf{ls}}(z,y) \vdash \exists u \,.\, x \mapsto u * \mathsf{ls}(u,y) \vee \mathsf{emp} \wedge x \approx y} \quad \begin{array}{l} \neg(x \approx y) \wedge x \mapsto z \models \exists u \,.\, x \mapsto u \\ \text{by instantiation } u \leftarrow z \end{array}}{\widehat{\mathsf{ls}}(x,y) \vdash \mathsf{ls}(x,y)}$$

The bottom inference rule introduces one of the two cases produced by unfolding the inductive definitions on both sides of the sequent[2]. The second inference rule is a reduction of the sequent obtained by unfolding, to a sequent matching the initial one (by renaming z to x), and allows to close this branch of the proof by an inductive argument, based on the principle of infinite descent [BDP11].

The simplification applied by the second inference above relies on the validity of the entailment $\neg(x \approx y) \wedge x \mapsto z \models \exists u \,.\, x \mapsto u$, which reduces to the (un)satisfiability of the formula $\neg(x \approx y) \wedge x \mapsto z \wedge \forall u \,.\, \neg x \mapsto u$. The latter falls into the prenex fragment, defined by the $\exists^*\forall^*$ quantifier prefix, and can be proved unsatisfiable using the instantiation of the universally quantified variable u with the existentially quantified variable z (or a corresponding Skolem constant). In other words, this formula is unsatisfiable because the universal quantified subformula asks that no memory cell is pointed to by x, which is contradicted by $x \mapsto z$. The instantiation of u that violates the universal condition is $u \leftarrow z$, which is carried over in the rest of the proof.

This example shows the need for a tight interaction between the decision procedures for the (quantified) base logic SL^k and the entailment provers for systems of inductive definitions built on top of it. An implementation of an inductive prover that uses SMT technology to simplify sequents is INDUCTOR [Ser17], which uses CVC4 [BCD$^+$11] for the reduction step. Just like the CYCLIST prover [BGP12] before, INDUCTOR is based on the principle of infinite descent [BDP11].

Future plans for INDUCTOR involve adding cut rules that would allow dealing with polymorphic representations of recursive data structures in a proof-theoretic fashion, as well as dealing with theories of the data within memory cells.

5 Conclusions

This paper surveys Separation Logic, a logical framework used to design modular and local analyses for programs that manipulate pointers and dynamically allocated memory cells. The essence of the logic is a notion of separable resource, which is embedded in the syntax and the proof system of the logic, before its nowadays widely accepted semantics was adopted. Program verifiers based on Separation Logic use local reasoning to model the updates in a compact way, by distinguishing the parts of the heap

[2] The second case $\mathsf{emp} \wedge x \approx y \vdash \exists u \,.\, x \mapsto u * \mathsf{ls}(u,y) \vee \mathsf{emp} \wedge x \approx y$ is trivial and omitted for clarity.

modified from the ones that are unchanged. The price to be paid for this expressivity is the difficulty of providing push-button decision procedures for it. However, recent advances show precisely what are the theoretical limits of decidability and how one can accomodate interesting program verification problems within them.

References

[AGH$^+$14] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *Foundations of Software Science and Computation Structures: 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proc.*, pages 411–425, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BBH$^+$06] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, volume 4144 of *LNCS*, pages 517–531, 2006.

[BCC$^+$07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Proc. CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

[BCD$^+$11] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification (CAV)*. Springer, 2011.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 52–68. Springer-Verlag, 2005.

[BDL12] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Information and Computation*, 211:106 – 137, 2012.

[BDP11] James Brotherston, Dino Distefano, and Rasmus L. Petersen. Automated cyclic entailment proofs in separation logic. In *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011, Proceedings*, pages 131–146, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *APLAS*, pages 350–367, 2012.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[CD11] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of c programs. In *Proc. of NASA Formal Methods'11*, volume 6617 of *LNCS*. Springer, 2011.

[CDOY07] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis*, pages 402–418. Springer Berlin Heidelberg, 2007.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, December 2011.

[CGH05] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *Foundations of Software Science and Computational Structures*, pages 395–409, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CJT15] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 457–466, New York, NY, USA, 2015. ACM.

[Cou90] B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12 – 75, 1990.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, pages 108–119, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[EIP18a] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. The complexity of prenex separation logic with one selector. *CoRR*, abs/1804.03556, 2018.

[EIP18b] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. On the expressive completeness of bernays-schönfinkel-ramsey separation logic. *CoRR*, abs/1802.00195, 2018.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[IO01] Samin S Ishtiaq and Peter W O'Hearn. Bi as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26, 2001.

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Deciding entailments in inductive separation logic with tree automata. In *Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proc.*, pages 201–218, Cham, Switzerland, 2014. Springer International Publishing.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 52–68, 2014.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, 2001.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *SIGPLAN Not.*, 39(1):268–280, 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74. IEEE Computer Society, 2002.

[RIS17] Andrew Reynolds, Radu Iosif, and Cristina Serban. Reasoning in the bernays-schönfinkel-ramsey fragment of separation logic. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 462–482, Cham, 2017. Springer International Publishing.

[RISK16] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in smt. In *Automated Technology for Verification and Analysis*, pages 244–261, Cham, 2016. Springer International Publishing.

[SC14] Mihaela Sighireanu and David Cok. Report on sl-comp 2014. *Journal on Satisfiability, Boolean Modeling and Computation*, 1, 2014.

[Ser17] Cristina Serban. Inductor: an entailment checker for inductive systems. URL: `https://github.com/cristina-serban/inductor`, 2017.