Philosophy & Technology manuscript No.
(will be inserted by the editor)

Why Separation Logic Works

Blinded

Received: date / Accepted: date

Abstract One might poetically muse that computers have the essence both of logic and machines. Through the case of the history of Separation Logic we explore how this assertion is more than idle poetry. Separation Logic works because it merges the software engineer's conceptual model of a program's manipulation of computer memory with the logical model that interprets what sentences in the logic are true, and because it deploys a proof theory that meets the engineer's design goals of timely and explanatory prediction of errors. Separation Logic is an interesting case because of its widespread success in verification tools, including those used by Facebook and others. For these two senses of model — the engineering/conceptual and the logical — to merge in a genuine sense, each must maintain their norms of use from their home disciplines. When this occurs, the development of both the logic and engineering benefit greatly. Seeking this intersection of two different senses of model provides a strategy for how computer scientists and logicians may be successful. Furthermore, the history of Separation Logic for analysing programs provides a novel case for philosophers of science of how software engineers and computer scientists develop models and the components of such models. We provide three contributions: an exploration of the

Blinded

extent of models merging that is necessary for success in computer science; an introduction to the technical details of Separation Logic, which can be used for reasoning about other exhaustible resources; and an introduction to (a subset of) the problems, process, and results of computer scientists for those outside the field.

1 Introduction

This paper focuses on logic as a technology by reflecting on the achievements in verification of computer programs using logics as tools. Specifically, we follow Separation Logic, a development within theoretical computer science firmly established by O'Hearn and Pym [1999], Ishtiaq and O'Hearn [2001] and Reynolds [2002]. The result is a case study with multiple uses. For logicians, it is an example of success by integrating engineering needs into both model theory and proof theory. Integrating with model or proof theory alone is not enough. For philosophers of science, it is an account of the working and reasoning process of field which is not commonly studied.

Separation logic adds a connective to standard logic called 'and, separately' that solves a problem of reasoning about the resources a computer program will need when it executes. We will lay out what makes reasoning about computer resources hard and explain Separation Logic's special arrangement of properties that enable us to use it effectively in program verification problems.[1] The 2016 Gödel prize, awarded to Brookes and O'Hearn for resolving

[1] A sub-discipline of logic and verification of computer programs has flourished within wider computer science since at latest 1970 with the activity surrounding Floyd–Hoare logic [Apt, 1981]. The first academic conference dedicated to studying programming languages, including the verification of languages using logic as a tool, took place in 1973 (Principles of of Programming Languages, or 'POPL', http://www.sigplan.org/Conferences/POPL/) and a dedicated journal appeared in 1979 (ACM Transactions on Programming Languages and Systems, or 'TOPLAS', http://toplas.acm.org). Independent publi-

these resource management problems with Concurrent Separation Logic, puts the problem in context:

> For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification ... .

Given the modern social reliance on software, program verification holds growing importance. Assigning the appropriate resources to a computer program is important for efficiency, accuracy, reliability, and security. Program verification as a discipline is focused by the practical challenges of making computer software function reliably, such as by assuring the program claims, uses, and releases the correct resources at the correct times. Resource allocation decisions are hard both in principle and in practice. In principle, given computer code (i.e., software), one cannot determine *a priori* when or if the program will halt [Turing, 1936]. In practice, for small programs this impossibility-in-principle is overcome by making an estimate based on similar programs. However, modern software projects at companies like Microsoft, Facebook, and Google have many millions of lines of code, written by thousands of people. With such a fragile, distributed, and complex system in which changing fives lines of code out of a million can drastically change behaviour, estimates based on experience are inadequate, to say the least. Therefore, although software is a human artefact, one does not in general know what any given piece of software will do when executed. To overcome these challenges, companies such as Facebook use Separation Logic to verify their mobile app software [Calcagno et al., 2015b] using a tool called 'Infer'. Separation Logic is not limited to one use; extensions of it are used, for example, to verify operating-system scheduling [Xu et al., 2016], a crash-tolerant file system [Chen et al., 2015], and an

cation venues help mark where an academic community forges it own identity, characteristic problems, and norms. Program verification may be some mix of computer science and logic, but it is also independent.

open-source cryptographic operation [Appel, 2015]. Our case study will be the development of Separation Logic through to these real-world applications.

The resource about which Separation Logic can best reason is computer memory, specifically Random Access Memory (RAM; hereafter, simply 'memory').[2] Appropriately resourcing memory for a computer program is an important task within computer science. Memory errors are not easily handled during program execution, and adversaries can use errors to remotely take control of computers using free, public attacks. Thus, although reasoning about computer memory errors may appear quite specific, it is a salient problem in practical computer science so far best addressed by Separation Logic.

Separation Logic was developed at a crossroads of two problems, a logical–theoretical issue of reasoning about resources generally, and a technological–computer-science problem of preventing errors in software memory usage. In this paper, we relate the special constellation of properties of Separation Logic due to this situation. Under the right conditions, interpreted the right way, the logical model can be at once a logic model and an engineering model. To be a genuine joint logic-engineering model is to take on the features and norms of use of models both in a logic and in an engineering discipline. In practice, not just the model but also the proof theory adapts to satisfice the needs of the engineering problem at hand ('satisfice' as per Simon [1996]). We do not propose anything surprising about the features of models in logic nor models in engineering. The surprise has come forward in

[2] RAM is the computer's scratch board of what it is working on presently, where the present is measured on the order of seconds or minutes. RAM is fast, but it is volatile, meaning roughly that if the computer is powered off RAM is lost. Verifying how a program uses RAM is important because it is volatile, and so information stored there is likely to be lost if certain errors occur. Computers have other types of memory; hard drives are persistent memory, and are stable when the computer is powered off.

powerful results for reasoning about computer programs once we gave Separation Logic the constellation of properties that genuinely make for features of both types of model.

We take Separation Logic as an example case out of several projects in computer science that exhibit this simultaneity of logic and engineering models. In the 1970s, Floyd–Hoare logic merged the engineer's notion of program execution into first-order logic. However, Hoare logic by itself does not adequately adapt to the task of efficiently reasoning about pointers and mutable data; proofs are not practically tractable [Bornat, 2000]. A primary feature of an engineering model is to be satisfactorily useful, not merely to include considerations from the engineered world. This further step is more rare. It requires adapting the logic model, model structure, and proof theory to suit the engineering model or task and to test that suitability empirically. For example, temporal logic, as used in tools built by Amazon to manage its infrastructure, also seems to have achieved this special merging of logic and engineering models [Newcombe et al., 2015]. We will survey how Separation Logic has adapted its features to the task of verifying a program's use of memory. It is adapted through its syntax and proof theory as well as its model structure — the engineering model does not merely supply semantics to a logical syntax.

There is a historical connection between scientific laws and logic models. The Logical Positivists in the mid-20th century held that a scientific theory was a set of sentences in first order logic. The physical world and its laws of nature are interpreted as a model of true scientific theories [Frigg and Hartmann, 2012, §1.3]. Logical Positivism and this usage of model have fallen out of favour. Practitioners use scientific or engineering models to represent phenomena or data [Frigg and Hartmann, 2012, §1]. When we say that Separation Logic merges logic and engineering models, we do not mean a by-definition (*de dicto*) merging reminiscent of Logical Positivism. We mean a logic built and empirically tested to usefully reason about a phenomenon.

Our focus is on practical questions of the efficient use of models for reliable reasoning, not on ontological questions of what processes are computation. Separation Logic reasons about machines colloquially called 'computers'. However, the important part of Infer is that it predicts what a complex object we care about will do, not questions of whether physical objects compute [Piccinini, 2007] or whether a computation is a miscomputation or dysfunctional [Floridi et al., 2015]. This distinction gives a sense of the extent to which tools like Infer are pragmatic, engineering projects. Yet, testing for mundane properties like stability still requires a novel development of a logical technology. As Swoyer might say, we represent computer programs 'in a medium that facilitates inference' [Swoyer, 1991] about them — Separation Logic.

The history of Separation Logic is presented as a case study for understanding experimentation and model development in computing. Discussions of such questions are available centering on general methodological views [Schiaffonati and Verdicchio, 2014] or a mechanistic view [Hatleback and Spring, 2014]. The case of Separation Logic also might inform discussions of how to overcome challenges in 'science of security', especially in the area of the relationship between science, engineering and formalism [Spring et al., 2017]. However, we focus on clearly presenting the case study and highlighting salient developments, and leave interpretation out of scope.

Another lens of interpretation for the case of Separation Logic is that of (scientific) representation. Suárez [2010] distinguishes between analytical and practical inquiries into the nature of representation. We consider Separation Logic tools to be a case study in pragmatic representation. The case has value within the analytic–practical distinction because within these verification tools one logical model (Separation Logic) is used as a pragmatic representation of another logical system (computer code). Tools such as Infer have both representational force and inferential capacities, as defined by Suárez [2010, p. 97]. Our case

study describes the details of Separation Logic that give it these two properties, thus making it both a model in the scientific sense and in the logical sense. We claim this merging, of both logical models and engineering- or scientific-type models, is a vital feature of what makes Separation Logic successful, and is worth emulating. A logic model that is also a scientific model also poses an interesting case for analytical inquiry. However, here we focus on the practical description of how programmers use Separation Logic to solve problems using this form of model building.

The two categories in which we elaborate Separation Logic's properties are its semantics, which has a clear interpretation in the mathematical model of computer memory, and its deployable proof theory, which is both automatable and modular so we can scale it to real-world problems. Both of these features are related to the properties of the connective 'and, separately', represented in symbols as $*$. We will survey the properties of this connective and its interpretation; however, the formalism is not strictly necessary to understand the impact of Separation Logic as a technology. The primary insight is to learn to recognize situations in which a logic model, by coincidence or by design, usefully overlaps with a model of a practical problem. When this coincidence is recognized and pursued the development of both the logic and the practical solution benefit.

Separation Logic mirrors the computers in the physical world in a deep and important way, in a way that first-order logic does not. Both atoms of Separation Logic and computer parts are composable in a natural way. In some sense, the other beneficial properties of Separation Logic derive from pursuing and refining the benefits of a logical primitive ($*$) that directly and cleanly captures the compositionality of resources in the physical world.

Section 2 describes the context of the application, including details about what challenges make program verification of allocation of computer memory resources a hard and important problem to solve. Section 3 accessibly introduces the properties of Separation

Logic that meet the relevant challenges. Section 4 surveys the logical properties (semantics, syntax, &c.) of Separation Logic, focusing on its novel connective ‘and, separately’ ($*$). For those not disposed to logical formalism, Section 4 can be safely glossed over without losing the broader narrative. Section 5 describes how the Frame Rule and automated abduction make Separation Logic a solution for reasoning about computer memory resources that is practical for large software development firms to deploy. Section 6 concludes by extracting advice for logicians from this case study: that merging aspects of a simultaneous logic–engineering model is a good start, but to succeed the logic model’s structure must be exploited to give some measurable benefit.

2 Solving a Hard Problem

Memory management is challenging, and errors potentially lead to unstable behaviour, resource exhaustion, or security threats. Management of the computer’s memory falls directly to the programmer in languages like C. This section will introduce the importance of C-like languages and the task of memory management. This description amounts to the programmer’s model of what the computer does, a model very much like any other engineering model. The main features of the model are pointers and memory locations, to which we give a minimal introduction. Next, we describe how verification using logic provides a satisfactory solution to the programmer’s challenge of memory management in C-like languages. The section concludes with some remarks on the additional practical challenges of solving these problems at scale within software-development companies.

The C programming language was first developed in 1972, to implement the UNIX operating system. Every major computer operating system is now written using C. C and languages derived from it — such as Java, C++, C#, and Python — may account for as

much as half the computer code written every year.[3] The impact of C is perhaps even more than this ratio may seem. C code is in the critical path for almost any task done using a computer, since the operating system on a computer controls what the rest of the code on the system is permitted to do.

As introduced above, the programmer cannot in general know what her program will do once written. The popularity of C has to do with its expressivity, speed, and effectiveness. Unfortunately, its benefits do not include easy identification or tolerance of errors. The computer can check that it understands the syntax the programmer wrote, which catches some errors. Beyond this, the programmer is ignorant to any errors that will occur during execution of the program, known as run-time errors. There are various technical types of run-time errors, but for our purposes we partition them into 'annoying' and 'catastrophic'. Annoying errors lead to incorrect results, for example dividing by zero. The program can catch annoying errors and recover. Catastrophic errors lead to the program fatally failing, such as by crashing or exhausting available resources. Recovering intermediate progress is not generally possible after a fatal failure. If the program is able to exhaust the whole system's resources, such an error may bring down the rest of the computer system as well, not just the program. Memory management errors are one common type of catastrophic run-time error. Memory management is not the only task for which Separation Logic provides a suitable logical substrate. Task scheduling within an operating system is another source of catastrophic run-time errors [Xu et al., 2016]. We touch on the use of Separation Logic to address this second example in Section 5.2; however, the memory management context is our primary example. To see what makes memory management errors catastrophic, we first consider the basics of computer memory.

[3] There is no precise way to count code written; however, analysis of publicly available web sites indicate C and descendant languages account for about half. See http://www.tiobe.com/tiobe_index.

The programmer's model of memory management abstracts away from the hardware. Computer memory is a slab of silicon electronics. Conventionally, the smallest elements are binary digits, or bits, interpreted as 1 or 0 based on whether the local voltage is high or low. The hardware is designed such that any location can be read or written equally quickly (thus 'random access' memory). Eight bits are usually grouped into a byte for the basic unit of memory with which humans interact. The programmer, and the human-readable programming languages we work with, thus model the memory as a list of individual bytes, like houses on a very long street. These bytes in memory are given a address from one to $2^{64} - 1$, in modern 64-bit operating systems, based on their position in this one-dimensional vector. This is a programmer's model of computer memory; memory location 12 need not actually be physically next to location 13.

Strictly, a pointer is the address of some object in memory. A pointer-variable (usually, unhelpfully, just 'pointer') is a kind of variable that contains an address; in particular, the address where some other variable's value is stored [Kernighan and Ritchie, 1988, p. 93]. Pointers are well-known in computer science to be both 'extremely powerful' and 'extremely dangerous' [Ishtiaq and O'Hearn, 2001, p.1]. Pointers are powerful because they allow calculation over items in memory without expensive duplication or moving of the actual chunks of data in memory that would otherwise be necessary. Figure 1 demonstrates pointer basics. Each variable is represented by a square. Its name is above the square, the contents are inside. If the content is a pointer, it is represented as a arrow to its target. A pointer may be declared, to reserve its name, without a target, which is represented by a wavy arrow without a target. A pointer with no target has the special value *NULL*, and is called a null pointer. One common memory management error which we can find with Separation Logic is if a program will attempt to use a null pointer in a situation that requires a pointer with a valid value.

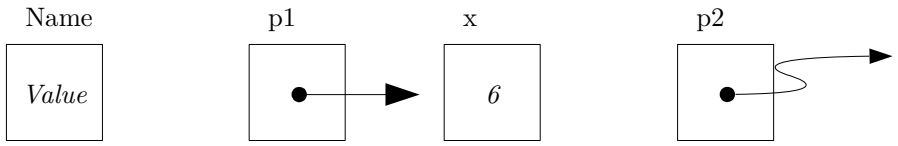

Figure 1 Anatomy of a pointer. Pointer $p1$ points to the variable x, whose value is 64. The name for pointer $p2$ is reserved, but it does not point anywhere; such a pointer has a special value called NULL.

A more subtle pointer error is to lose track of items in memory. An item in memory is only accessible if there is a pointer to it. Garbage is the official term for memory which is allocated (i.e., reserved for use) but not accessible because all pointers to it have been removed. If memory garbage is not explicitly cleaned up by the programmer, memory eventually gets clogged by allocated but inaccessible chunks of garbage data. This slow exhaustion of reserved memory by failure to clean up is called a memory leak. Figure 2 demonstrates one way a memory leak may occur. The technical term for cleaning up memory is to free it; that is, release reservation on its use. Unfortunately, it is not so simple as to just ensure the program frees all memory eventually. Errors when freeing memory also lead to dangerous behaviour. If the program maintains and uses a pointer to a memory location after freeing the memory, the location could have been used by another program to store different data.

We glibly termed these sorts of errors catastrophic. A program with a memory management error will behave erratically or fail suddenly. Whether this behaviour is catastrophic in a human sense depends on the importance of the program. If a word processor has a memory leak which means it cannot run for more than four hours, this is probably fine. If the software is for an air traffic control radar facility, it is more severe.[4]

Memory management errors are also security problems. The security community describes a reference list of canonical types of flaws that lead to security vulnerabilities, called

[4] The FAA press release on such an ATC failure does not specifically identify the software flaw type; however, the description suggests that it was a memory leak [Federal Aviation Administration, 2015].

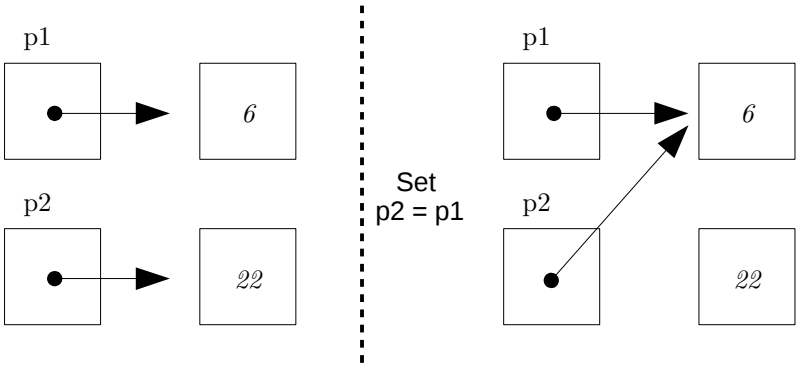

Figure 2 One example error involving pointers. The data element 22 is no longer accessible, because there is no pointer to it. That memory space has 'leaked' and cannot be freed (released back to the computer) or accessed. Memory leaks lead to resource exhaustion, as it is a one-way process and eventually the whole memory is full of leaked garbage which crowds out all useful programs.

the Common Weakness Enumeration. A quick survey of the entries for null pointer exceptions, resource leaks, and memory leaks (which are CWE-476, CWE-402, and CWE-401, respectively) provides a long list of software that has been vulnerable to a hostile takeover by an adversary due to these memory management errors [MITRE, 2015]. Again the amount of harm depends on the importance of the victimized computer. However, criminals can use and resell the electricity and network connection of any computer, to either hide more sinister attacks or rent as infrastructure for less technologically capable criminals [Sood and Enbody, 2013]. Thus, it is important to prevent vulnerabilities such as memory management errors in all computers.

We have elaborated two reasons memory management errors are problematic. They cause instability and make a program crash, which is bad for functionality and usability. They also frequently lead to security vulnerabilities which are exploitable by adversaries and criminals. There are two classes of methods to find flaws in computer software: static and dynamic. In static analysis, we analyse symbolic and structural features but do not run

the code, thus ‘static’ as the code does not move, so to speak. In dynamic analysis, we run the code and measure what happens. We use Separation Logic to find these errors statically, without running the program.

Success in program verification can be measured in at least four ways: reduced software flaws, accuracy of findings, speed of analysis, or reduced human time to fix software flaws. In practice, measuring how many flaws a technique finds is easy but hard to interpret. The total number of flaws remains unknown in principle using static analysis, because of Turing’s halting result, as discussed in Section 1. In practice, it is simply hard and too costly to exercise all the possible execution paths a program might take using dynamic analysis. Therefore, we cannot know for certain how many flaws remain undetected, which makes calculating accuracy or relative reduction in flaws impracticable. The problem is, essentially, that we cannot tell if finding 99 flaws is 99% effective or 2% effective. A more interpretable measure for the software industry is the rate at which found flaws are able to be fixed. This measure relates to analysis speed, because humans fix software better if given fast feedback. These desirable engineering outcomes suggest static analysis.

To make progress with static analysis, one must take a defined subset of the general problem of all software flaws. Since memory management causes such headaches in practice, Separation Logic was developed towards targeting them. To target memory in particular, elements of the logic faithfully incorporate the engineer’s model of how the computer manages memory; that is, pointers, as previously described. In practice, what we will arrive at is a program to check other programs statically. This checker makes use of Separation Logic and an engineer’s model of pointers. Within this checking software, Infer, the logic model and the engineering model will coincide. This confluence overcomes several of the challenges described in this section to more effectively prevent memory management errors, leading to more stable and secure code.

Naturally, for any given deployment of logic there are considerations for success beyond pure technical logical matters. Brooks' essay [Brooks Jr, 1995] is an example starting point for general software engineering considerations. For a program analysis tool such as Infer, integration with an organization's programming culture and process is significant work; see O'Hearn [2015]. For proofs of properties closer to functional correctness or operating system or crypto code, effective integration within a powerful proof assistant is critical [Appel et al., 2014]. While these contextual questions are important, this paper will focus on the features of the logic and its model that contribute to its success, not on the additional contextual factors which are nonetheless important.

In this section, we have introduced pointer management in computer memory. Pointer mismanagement can lead to serious stability and security flaws. Such flaws are hard to find dynamically during run-time. However, finding such flaws statically, based on the program's source code, has historically been too hard to do well enough to be useful. In the following sections we describe how Separation Logic succeeds at this hard task. Section 3 introduces all the properties of the logic that contribute to success. Separation Logic's properties that make it useful can be tackled in two broad categories: semantics (Section 4) and proof theory (Section 5). We will see that the logic undergoes a holistic adaptation to meet the practicalities of the engineering task.

3 Why Separation Logic Works

Separation Logic works for solving this problem of reasoning about memory allocation because of a group of features:

- A useful engineering model of computer memory;

- A logical model and language that are grounded, respectively, in an interpretation of or semantics for exhaustible resources;
- The productive overlap of these two (types of) models;
- The use of the connective $*$ for 'and, separately', from the bunched logic BI, to facilitate the formulation, in the setting Separation Logic's Floyd–Hoare-style formulation, of a 'Frame Rule' to support compositional local reasoning; and
- Scalable pre- and post-conditions.

The first three elements are modelling choices that provide a powerful capacity for prediction of computer memory usage which is not otherwise available. The latter two elements provide a scalable algorithm for calculating and proving these predictions for a given computer program.

The bunched logic BI can be interpreted as a logic of exhaustible resources [Galmiche et al., 2005]. For example, if one has 10 coins, it is certainly true that one has the capacity to buy a red widget that costs 4 coins. It is also true that one has the capacity to buy a yellow widget that costs 5 coins and the capacity to buy a blue widget that costs 7 coins. It is not, however, true that one has the capacity to buy both a yellow widget and a blue widget — that would require a total of 12 coins — but one does have the capacity to buy both a red widget and and a yellow widget — requiring a total of 9 coins — or two yellow widgets — requiring exactly 10 coins. The resource-interpretation of BI's semantics provides a precise interpretation of formal logical statements of all of these cases. More specifically, BI makes a distinction usual logical 'sharing' conjunction, for example,

'10 coins is enough for a yellow widget and is enough for a blue widget'

which is true only if the resource of 10 coins can be shared by the two parts of the statement, and the separating conjunction, for example,

‘10 coins is enough for both a red widget and a yellow widget’

where the resource of 10 coins must be divided, or separated, into those required for each part of the statement.

Computer memory, like money, is an example of an exhaustible resource. Though a computer is, basically, electricity and magnetism in silicon, computer programmers[5] do not write software as if they were individually manipulating millions of tiny magnets. Like most engineers, a programmer works with a model[6] of what she is building. To the programmer, the model of computer’s memory is provided by the stack and heap. In a violently simplified analogy, the stack is what you’re doing, and the heap is what you’re working on. These metaphorical names are evocative of their actual function. The stack is an ordered array of data elements, and the computer can only put elements on the top, and take them off the top. This structure ensures an orderliness and efficiency good for sequential recursive instructions but not good for big chunks of data. In the heap elements can be accessed in any order but only so long as the program’s stack has a pointer, or index, to the information’s location in memory. Note that the stack maps variables into values, whereas the heap maps addresses into values [Reynolds, 2002]. Though the programmer’s model abstracts away from it, location here has a physical interpretation. Computer memory is an apparatus with a numerical address for each microscopic individuated bit in its vast silicon plane. Like the structural engineer who has mathematical equations that inform her choices of bridge design, the programmer uses the model of the stack and heap to inform software development. In

[5] Instead of ‘programmer’, one may find ‘(software) developer’, ‘coder’, or ‘software engineer’. These terms have differing connotations across various communities, which are not relevant here. We just mean anyone who writes software.

[6] The details of what is or is not a model are subtle. We gloss over the subtleties one may find in Giere [2004] or Illari and Williamson [2012], for example, because these subtleties among models in science and engineering are not necessary to differentiate them from models in logic.

both cases, the engineer's model's prediction is not perfect, and the bridge or the program could collapse despite best efforts.

One success of Separation Logic is to merge the logic-model and the engineering-model. The stack and the heap have formal representations in Separation Logic, with the heap as a resource. A programmer's models of memory can be expressed as sentences within Separation Logic without unacceptable loss of applicability to the real world. Sentences in Separation Logic have deductible consequences, and can be proved. In the computing context, this amounts to proving properties of future behaviour of possible program executions. Such proofs enhance the engineering-model of the program directly. If the logic deduces a section of the program code will make a memory-usage error, the code can be tested empirically to verify the error and gather information about the mechanism by which the error is committed. These logical and empirical results update the programmer's model of the software, and she fixes the code accordingly. Such remediation is not possible without logical tools in which the model of the program in an scientific-engineering sense meaningfully overlaps with the logic's model.

An engineer's model commonly merges mathematical modelling with some subject-matter expertise to make predictions. For example, a structural engineer can use mathematical models to predict when stress on a bridge element will exceed its shear strength because we have accurate physical measurement of each material's properties, gravity, etc. But computers are devices made up of logic more-so than metal. However, just as when we build a bridge, if we build a computer and write software for it, we do not know everything that the computer will do just because we designed it. There are interactions with the world that are unpredictable. Logic is one of the subject-matter expertise areas we use as programmers, as a structural engineer uses materials science. Also similar to other engineering or science disciplines, using the correct logic is important. The correct logic for a programming task is

determined empirically; in our experience with Separation Logic, the process seems similar to the usual scientific model-building.

There are three distinct logical elements which have made separation logic successful: the connective $*$, the Frame Rule, and the specification of automatable abduction rules.

The connective 'and, separately' ($*$) is related to the familiar connective for conjunction ($\wedge$). In the familiar case, we write $\phi \wedge \psi$ for the situation $w \models \phi \wedge \psi$ (read $w \models ...$ as 'the world w "supports" or "satisfies" ... ') iff $w \models \phi$ and $w \models \psi$. We can use this sort of structure to make a different conjunction, 'and, separately' to capture the resource interpretation for reasoning about exhaustible resources such as computer memory. We need to know a little more about the world w that supports $\phi * \psi$ to say when $w \models \phi * \psi$. We need to be able to break the world up into disjoint parts, which we represent as $w_1 \cdot w_2 = w$ to say w_1 composed with w_2 is w. If we have this decomposition, then $w \models \phi * \psi$ iff there are $w_1 \cdot w_2 = w$ such that $w_1 \models \phi$ and $w_2 \models \psi$ [Galmiche et al., 2005].[7]

The difference between $w \models \phi \wedge \psi$ and $w \models \phi * \psi$ is just that aspects of the world can be reused to satisfy conjunction, but not with the separating conjunction. This difference is most obvious in that if $w \models \phi$, then $w \models \phi \wedge \phi$ is always true, but $w \models \phi * \phi$ need not be true, because it may be the case that there is one part of the world that satisfies ϕ ($w_1 \models \phi$), but the rest of the world does not ($w_2 \not\models \phi$). If ϕ is 'I have enough money to buy a drink', then $w \models \phi \wedge \phi$ says nothing new, but $w \models \phi * \phi$ says I have enough money to buy two drinks.

The second technical element that enables the success of Separation Logic is the Frame Rule. Separation Logic builds on Floyd–Hoare logic [Apt, 1981] (henceforth, 'Hoare logic'). Hoare logic developed through the 1970s specifically to reason about the execution of com-

[7] The treatment of $*$ described here is for Boolean BI, where the set of worlds is not ordered. Intuitionistically (see Section 4.2), we require that $w_1 \cdot w_2 \sqsubseteq w$, where $\sqsubseteq$ is a preorder that is defined on the set of worlds and which satisfies *monotonicity*.

puter programs. The intuition is straightforward: proving the relevant properties of a program C amounts to proving that whenever a certain precondition holds before executing C, a certain postcondition holds afterwards. This statement,known as a Hoare triple, is written formally as

$$\{\phi\}\, C\, \{\psi\},$$

where ϕ is the precondition and ψ is the postcondition. Hoare logic provides various proof rules for manipulating triples. For example, composing two program fragments if the postcondition of the first is the precondition of the second. Such deductions are written as

$$\frac{\{\phi\}\, C_1\, \{\chi\} \quad \{\chi\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1 ; C_2\, \{\psi\}}$$

with the given statements on top and the deduced statement on the bottom.

The Frame Rule lets us combine a Hoare triple with $*$ — for 'and, separately' — to reason about just the local context of a program fragment. This support for local reasoning is critical, supporting *compositional* reasoning about large programs by facilitating their *decomposition* into many smaller programs that can be analysed and verified *independently*. This analysis relies on the compliance of resource semantics with Frege's principle that the meaning of composite expression be determined by the meanings of its constituent parts.

We write the Frame Rule as

$$\frac{\{\phi\}\, C\, \{\psi\}}{\{\phi * \chi\}\, C\, \{\psi * \chi\}}$$

provided χ does not include any free variables modified by the program C (that is, formally, $\text{Modifies}(C) \cap \text{Free}(\chi) = \emptyset$). The Frame Rule is powerful because it lets us ignore context we have not changed. Reasoning locally, as opposed to globally, is vital when analysing large programs. Normally, a program verification technique would have to re-evaluate a whole program if one line changes. When the program has even ten thousand lines of code this is

prohibitively inefficient. Industrial scale program analysis must analyse millions of lines of code, and so without the ability to reason locally any approach will fail.

The sorts of preconditions and postconditions that we are interested in for separation logic are directly related to the programmer's goals for modelling. Abstracting away from the details of a computer, the precondition may be something like 'there exists an available resource not currently in use' and the postcondition may specify the details of 'nothing bad happened' or 'the program worked'. The frame rule is powerful because we can break the program up into many disjoint parts, and once we have proved $\{\phi\}C\{\psi\}$ for one of the parts, we can take χ to be the union of all the pre- and post-conditions for all the other disjoint parts of the program and know that $\{\phi * \chi\}C\{\psi * \chi\}$ will hold without having to re-prove the statement in the new context. Thus, if a million lines of code can be broken up in to ten thousand disjoint fragments, then when we change code in one of the fragments we only need to prove $\{\phi\}C\{\psi\}$ for that fragment and not the 9,999 others.

Local reasoning is helpful for reasoning about programs at scale, but a human still has to be rather clever and expert to choose exactly the right pre- and post-conditions to prove facts about C. There are simply not enough clever, expert people to do this at scale. Software development at firms like Amazon or Google involves many hundreds of developers who each need their code checked and analysed within a few hours of making complex changes. A human logician might take days to figure out the right conditions for the Hoare triples for each code change. Even if that many experts could be trained, no company is likely to pay for that sort of increased labour cost. Separation logic works because we are able to abduce potential pre- and post-conditions to test.

Industrial-scale use of logic for proving program properties requires a deployable proof theory. The combination of local reasoning and abduction support this deployable proof theory for separation logic. Abduction, as introduced by Peirce [Bergman and Paavola, 2016],

is akin to hypothesis generation. Initial implementations of Separation Logic to analyse programs required the human analyst to provide the pre- and post-conditions. However, we have been able to automate abduction because the scope of problems we attempt to solve is well-defined and because computer code is reasonably well-structured [O'Hearn, 2015]. Automation means writing a computer program which is able to analyse other computer programs. One such analysis program, Infer, was recently published freely as open-source code for anyone to use [Calcagno et al., 2015b].

Computer code is not arranged into Hoare triples, so verification tools must create that logical structure as they read and analyse the program. A pre- or post-condition may be established in a segment of the code far distant from where they are needed or checked. We cannot build a table of all possible combinations of legal Hoare triples to solve this problem, the number is astronomically large for even modest programs. Abduction makes this problem manageable by dynamically determining what conditions a segment of code might expect. Each abductive hypothesis is not perfect. But each hypothesis can be tested quickly and the results reused. The analysis program can quickly and soundly check each hypothesized pre- and post-condition; because reasoning is local the result can be stored and reused easily.

Separation logic is useful because it calculates predictions of hard-to-handle computer program execution errors; this is well known. Why separation logic is so effective at this useful task was not deeply questioned; to some extent one does not question how the goose lays the golden eggs. Yet, understanding successful tactics will help reproduce success in different areas of inquiry, of which there are many in logic and computer science. The thesis we are advancing is that the useful predictions are generated by the convergence of two senses of the word model — the logic model designed specifically for the task and the programmatic-

engineering model of what the computer and its software actually do — together with a proof theory that meets the engineer's goals of timely and explanatory prediction of errors.[8]

This section has only briefly introduced the features of Separation Logic that are adapted for it to become an adequate engineering model. An appreciation of the extent to which the details of the logic model and the engineering model come together requires a more technical exposition. To this end, we introduce the semantics of separation logic is Section 4. Those less inclined to logics can skim these details without loss of continuity. Furthermore, to be successful on the necessary large scales, our engineered logic requires a proof theory that is deployable. Section 5 discusses two features that contribute to making the proof theory 'deployable' — local reasoning, as supported by the Frame Rule, and automated abduction. These details demonstrate directly our argument that the logic model and the engineering model are inescapably and intricately intertwined. Our positive thesis concludes that this is no accident, but rather the source of separation logic's success in analysing programs.

4 The Semantics of Separation Logic

The history of Separation Logic in particular stretches back from the foundations of programming in the 1970s through to practical changes in the way tech giants produce computer programs today. The first piece of the history is Hoare's development of assertion programs, with the insight that valid program execution can be interpreted as a logical proof from the

[8] That results must be timely is straightforward; clearly a programmer cannot wait 100 years for the analysis to complete. That the result also provides satisfactory explanation of the error is equally important. Explanation requires a practical and a human sense. Practically, the programmer must receive enough detail to locate and fix the error. Psychologically, programmers are less likely to trust an arcane or unintelligible report than a transparent documentation of the entities and activities responsible for the error. This transparency merges a sense of adequate mechanistic explanation [Illari and Williamson, 2012] with the logical community's sense of when a proof is both convincing and elegant.

preconditions to the postconditions [Apt, 1981]. However, the familiar classical logical connectives — $\neg$, $\vee$, $\wedge$, and $\rightarrow$ — and quantifiers — $\exists$ and $\forall$ — did not capture the resource management problems that computer science then found intractable. Linear Logic [Girard, 1987], originally developed as a tool in proof theory, introduced an explicit single-use resource interpretation and a modality (!) to mark resources as being usable as many times as needed. Although linear logic has enjoyed much success, the resource-management problem remained out of its reach.

With the benefit of hindsight, we can see that what was necessary was a logic of resources with a structure that was composable and decomposable in a way that mirrors the composability of resources in the physical world. Resources in linear logic are usable once, or infinitely many times. This pattern does not match real-world resources like sandwiches or money. How many hungry people a sandwich satisfies depends on how many parts it can be decomposed into that independently satisfy a hungry person. This number is often more than one but less than 'as many as needed'. We want a logical structure that mirrors this behaviour. We will arrive at such a structure in three historical stages of exposition: first, bunched logic, then the semantics of bunched logic, and finally the semantics for resources in separation logic.

4.1 Bunched Logic

Towards the end of the twentieth century, O'Hearn and Pym [1999] introduced BI, the 'logic of bunched implications'. In its initial form, BI can be understood as freely combining the intuitionistic propositional connectives (BI's additive connectives) with the multiplicative fragment of intuitionistic linear logic (BI's multiplicative connectives).

The idea of bunching — an older idea from relevant logic; see, for example, Read [1988], Dunn and Restall [2002] — is used to formulate natural deduction and sequent calculus proof systems for BI. The key point is that proof-theoretic contexts are constructed using two operations, one corresponding to the additive conjunction, $\wedge$, and one corresponding to the multiplicative conjunction, $*$.

To see how this works, consider the natural deduction rules for introducing the additive and multiplicative conjunctions, $\wedge$ and $*$, respectively. If $\Gamma \vdash \phi$ is read as 'ϕ is provable from assumptions Γ, then these are the following:

$$\frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{\Gamma ; \Delta \vdash \phi \wedge \psi} \wedge \mathrm{I} \quad \text{and} \quad \frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{\Gamma , \Delta \vdash \phi * \psi} * \mathrm{I}.$$

Notice that the $\wedge$I rule combines the contexts Γ and Δ using semi-colon, corresponding to $\wedge$, whereas the $*$I rule combines them using the comma. The key difference is that the semi-colon admits the contraction and weakening rules,

$$\frac{\Theta(\Gamma ; \Gamma) \vdash \phi}{\Theta(\Gamma) \vdash \phi} \mathrm{C} \quad \text{and} \quad \frac{\Theta(\Gamma) \vdash \phi}{\Theta(\Gamma ; \Delta) \vdash \phi} \mathrm{W},$$

respectively, whereas the comma does not. The form of these rules draws attention to a key point about bunches: they are trees, with leaves labelled by propositions and internal vertices labelled with ' ; ' and ' , '.

A key consequence of the availability of contraction for $\wedge$, for example, is that the simple additive form of the $\wedge$I rule, in which the context Γ is shared between the two components of the conjunction, is recovered when $\Delta = \Gamma$.

4.2 The Semantics of Bunched Logic

In the spirit of this paper, the semantics of BI can be seen as being based on the notion of resource. Specifically, adopting the approach of constructing a engineering model of resource,

and bearing in mind our examples of interest as discussed above, we can observe that two properties of resource are of central importance.

- Given two elements of a given type of resource, it should be possible, subject to an observation spelled out below, to combine them to form a new element of that type of resource. In the case of the example of coins mentioned in Section 3, we consider combination to be addition of numbers of coins.
- Given two elements of a given type of resource, it should be possible to compare them. Again, in the case of the example of coins mentioned in Section 3, we compare the number of coins available (10) with the number required to buy *both* a yellow widget and blue widget (12).

Mathematically, these 'axioms' for resource can be captured conveniently by requiring that a given type of resource carry the structure of a preordered partial commutative monoid.[9] That is, a (given type of) resource R is given as

$$R = (R, \cdot, e, \sqsubseteq),$$

where R is the set of resource elements of the given type, $\cdot : R \times R \rightharpoonup R$ is a partial function, e is a unit (or identity) element for $\cdot$ such that, for all $r \in R$, $r \cdot e = r = e \cdot r$, and $\sqsubseteq$ is a preorder on R. In the case of the example of coins, the monoid of resources can be taken to be the ordered monoid of natural numbers,

$$(\mathbb{N}, +, 0, \leq).$$

The partiality — in general, addition of natural numbers happens to be total — of $\cdot$ reflects that in many natural examples of resource, such as computer memory, not all combinations of resource elements will be defined. Where necessary for clarity, we write $r\downarrow$ to denote that a resource r is defined.

[9] A preorder $\sqsubseteq$ on a set S is required to be reflexive and transitive. It is not a total order.

Finally, for technical mathematical reasons, we require that the combination $\cdot$ and comparison $\sqsubseteq$ of resources should interact conveniently. Specifically, we require the following functoriality condition: for all r_1, r_2, s_1, s_2,

$$r_1 \sqsubseteq r_2 \text{ and } s_1 \sqsubseteq s_2 \text{ implies } r_1 \cdot s_1 \sqsubseteq r_2 \cdot s_2.$$

For example, in the ordered monoid of natural numbers, $(\mathbb{N}, +, 0, \leq)$, if $m_1 \leq m_2$ and $n_1 \leq n_2$ implies $m_1 + n_1 \leq m_2 + n_2$.

This set-up is known as *resource semantics*.

So far, we have described a model of resource quite simply in the style of an engineering model. However, the mathematical structure we have obtained is exactly what is required to defined a formal logical model of BI.

The starting point for this is intuitionistic logic [Kripke, 1965] and its Kripke semantics in which an implication $\phi \to \psi$ is interpreted as a function, or procedure, that converts evidence for the truth of ϕ into evidence for the truth of ψ. Technically, this is achieved using a preorder on the set of possible worlds, or states of knowledge, [van Dalen, 2004]: if an observer can establish the truth of ψ from the truth of ϕ at its current state of knowledge, then it must also be able to do so at any greater state of knowledge; this is called *monotonicity*.

A similar interpretation can be applied to the separating conjunction, $*$, described above in Section 3: If $r \models \phi * \phi$ says I have enough money to buy two drinks, then, if $r \sqsubseteq s$, $s \models \phi * \phi$ also says I have enough money to buy two drinks.

Monotonicity is defined formally below.

With these interpretations in mind, and assuming (i) a 'resource monoid' $R = (R, \cdot, e, \sqsubseteq)$, (ii) that $r \models \phi$ is read as 'the resource r is sufficient for ϕ to be true', and (iii) for each atomic proposition p, a set $\mathscr{V}(\mathrm{p})$ of resource elements that are sufficient for $\mathscr{V}(\mathrm{p})$ to be true, we can give a formal semantics to BI as follows, where $r \models \phi$ is read, as before, as 'the world r

supports, or satisfies, the proposition ψ':

$$
\begin{array}{rcl}
r \models \mathrm{p} & \text{iff} & r \in \mathscr{V}(\mathrm{p}) \\
r \models \bot & \text{never} & \\
r \models \top & \text{always} & \\
r \models \phi \vee \psi & \text{iff} & r \models \phi \text{ or } r \models \psi \\
r \models \phi \wedge \psi & \text{iff} & r \models \phi \text{ and } r \models \psi \\
r \models \phi \to \psi & \text{iff} & \text{for all } r \sqsubseteq s,\ s \models \phi \text{ implies } s \models \psi \\
 & & \\
r \models \mathrm{I} & \text{iff} & r \sqsubseteq e \\
r \models \phi * \psi & \text{iff} & \text{there are worlds } s \text{ and } t \text{ such that } (s \cdot t)\downarrow \sqsubseteq r \text{ and} \\
 & & s \models \phi \text{ and } t \models \psi \\
r \models \phi \mathrel{-\!\!*} \psi & \text{iff} & \text{for all } s \text{ such that } s \models \phi \text{ and } (r \cdot s)\downarrow,\ r \cdot s \models \psi.
\end{array}
$$

All propositions ϕ are required to satisfy *monotonicity*: if $r \models \phi$ and $r \sqsubseteq r'$, then $r' \models \phi$.

With this semantics and with a system of rules of inference along the lines of the ones sketched above, we can obtain soundness and completeness theorems for BI: the propositions that are provable using the inference rules correspond exactly to the ones that are true according to the semantics [Galmiche et al., 2005].

In the context of this semantics, the significance of the contraction and weakening rules can now be seem: they explain how the semi-colon combines properties of resources that may be shared whereas the comma combines properties of resources that must be separated.

Although we have described the original, intuitionistic formulation of BI, Separation Logic in fact uses the classical or 'Boolean' variant [Reynolds, 2002, Ishtiaq and O'Hearn, 2001]. Boolean BI is based on classical logic, so that the implication $\phi \to \psi$ is defined to be $(\neg\phi) \vee \psi$, where the negation satisfies the classical 'law of the excluded middle'. Technically, we work now with a resource semantics based simply partial commutative monoids,

without including a preorder; that is,

$$R = (R, \cdot, e),$$

where R is the set of resource elements of the given type, $\cdot : R \times R \rightharpoonup R$ is a partial function, e is a unit (or identity) element for $\cdot$ such that, for all $r \in R$, $r \cdot e = r = e \cdot r$.

With models of this form, the semantics of Boolean BI is given as above, but with the following variations:

$$r \models \phi \to \psi \quad \text{iff} \quad r \models \phi \text{ implies } r \models \psi$$

$$r \models \mathrm{I} \quad \text{iff} \quad r = e$$

$$r \models \phi * \psi \quad \text{iff} \quad \text{there are worlds } s \text{ and } t \text{ such that } (s \cdot t) \downarrow = r \text{ and } s \models \phi \text{ and } t \models \psi.$$

Notice that the separating conjunction now divides the resources exactly.

4.3 The Resource Semantics of Separation Logic

The resource semantics described above, much richer than that which is available in linear logic [Girard, 1987], allows the construction of specific logical models for a characterization of computer memory. Characterizing memory addressed challenging problems in program verification [Ishtiaq and O'Hearn, 2001]. Over the following 15 years, this logic — called Separation Logic [Reynolds, 2002, O'Hearn, 2007] — developed into a verification tool successfully deployed at large technology firms like Facebook [O'Hearn, 2015] and Spotify [Vuillard, 2016]. In this section, we explain how the semantics of (Boolean) BI as described above forms the basis of separation logic.

Ishtiaq and O'Hearn [2001] introduced 'BI Pointer Logic', based on a specific example of Boolean BI's resource semantics. Three points about BI Pointer Logic are key.

- First, its resource semantics is constructed using the stack, used for static, compile-time memory allocation, and the heap, used for dynamic, run-time memory allocation:
- Second, the semantics of the separating conjunction, $*$, splits the heap, but not the stack: the stack contains the allocations required to define the program, which are unchanged at run-time; the heap contains the allocations made during computation.
- Third, it employs a special class of atomic propositions constructed using the 'points to' relation, $\mapsto$: $E \mapsto E_1, E_2$ means that expression E points to a cons cell E_1 and E_2. (It also employs a class of atomic propositions which assert the equality of program expressions, but this is a standard formulation.)

These factors combine to give an expressive and convenient tool for making statements about the contexts of heap (cons) cells. For example, the separating conjunction

$$(x \mapsto 3, y) * (y \mapsto 4, x)$$

says that x and y denote distinct locations. Further, x is a structured variable with two data types; the first, an integer, is 3, and the second is a pointer to y. The variable y denotes a location with a similar two-part structure in which the first part, also called the car, contains 4 and the second part, sometimes called the cdr ('could-er'), contains a pointer back to x [Ishtiaq and O'Hearn, 2001]. Note that the pointers identify the whole two-part variable, not just the car. Figure 3 displays this linked list relationship in pictures.

Separation Logic can usefully and safely be seen (see O'Hearn and Yang [2002] for the details) as a presentation of BI Pointer Logic [Ishtiaq and O'Hearn, 2001]. The semantics of BI Pointer Logic, a theory of (first-order) Boolean BI (BBI), is an instance of BBI's resource semantics in which the monoid of resources is constructed from the program's heap. In detail, this model has two components, the store and the heap. The store is a partial function mapping from variables to values, $a \in \mathrm{Val}$, such as integers, and the heap is a partial

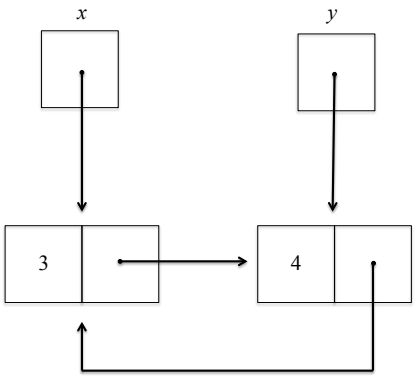

Figure 3 As in Figure 1, variable names are listed above their square, and contents of the variable are inside the square. The diagram represents the logical statement $(x \mapsto 3, y) * (y \mapsto 4, x)$.

function from natural numbers to values. In logic, the store is often called the valuation, and the heap is a possible world. In programming languages, the store is sometimes called the environment. Within this set-up, the atomic formulae of BI Pointer Logic include equality between expressions, $E = E'$, and, crucially, the points-to relation, $E \mapsto F$. To set all this up, we need some additional notation. $dom(h)$ denotes the domain of definition of a heap h and $dom(s)$ is the domain of a store s; $h\#h'$ denotes that $dom(h) \cap dom(h') = \emptyset$; $h \cdot h'$ denotes the union of functions with disjoint domains, which is undefined if the domains overlap; $[f \mid v \mapsto a]$ is the partial function that is equal to f except that v maps to a; expressions E are built up from variables and constants, and so determine denotations $[\![E]\!]s \in \mathrm{Val}$. With this basic data, the satisfaction relation for BI Pointer Logic is defined as in Figure 4.

The judgement, $s, h \vDash \phi$, says that the assertion ϕ holds for a given store and heap, assuming that the free variables of ϕ are contained in the domain of s.

The remaining classical connectives are defined in the usual way: $\neg\phi = \phi \to \bot$; $\top = \neg\bot$; $\phi \vee \psi = (\neg\phi) \to \psi$; $\phi \wedge \psi = \neg(\neg\phi \vee \neg\psi)$; and $\forall x \,.\, \phi = \neg\exists x \,.\, \neg\phi$.

$$
\begin{array}{rll}
s,h \models E = E' & \text{iff} & [\![E]\!]s = [\![E']\!]s \\
s,h \models E \mapsto (E_1, E_2) & \text{iff} & [\![E]\!]s = \mathrm{dom}(h) \text{ and } h([\![E]\!]s) = \langle [\![E_1]\!]s, [\![E_2]\!]s \rangle \\
 & & \\
s,h \models \mathtt{emp} & \text{iff} & h = [\,] \text{ (the empty heap)} \\
s,h \models \phi * \psi & \text{iff} & \text{there are } h_0, h_1 \text{ s.t. } h_0 \# h_1,\ h_0 \cdot h_1 = h, \\
 & & s,h_0 \models \phi \text{ and } s,h_1 \models \psi \\
s,h \models \phi \mathrel{-\!\!*} \psi & \text{iff} & \text{for all } h', \text{ if } h' \# h \text{ and } s,h' \models \phi, \text{ then } s, h \cdot h' \models \psi \\
 & & \\
s,h \models \bot & \text{never} & \\
s,h \models \phi \to \psi & \text{iff} & s,h \models \phi \text{ implies } s,h \models \psi \\
s,h \models \exists x\,.\,\phi & \text{iff} & \text{for some } v \in \mathrm{Val}, [s \mid x \mapsto v], h \models \phi
\end{array}
$$

Figure 4 The satisfaction relation for BI Pointer Logic [Ishtiaq and O'Hearn, 2001].

The definition of truth for BI Pointer Logic — that is, its satisfaction relation — provides a first clear illustration of argument, made in Section 3, concerning the merging of logic-models and engineering-models. The stack and the heap and the ways in which they are manipulated by programs are considered directly by working programmers: indeed, memory management at this level of abstraction is a key aspect of the C programming language (see Kernighan and Ritchie [1988] for descriptions of the history, definition, and usage of C).

As we have seen, BI Pointer Logic, with its truth-functional semantics of the form

$$s,h \models \phi$$

provides and elegant semantics for reasoning about the correctness of programs that manipulate computer memory. However, as we have seen, for reasoning directly about the behaviour of programs, Hoare logic, based on triples $\{\phi\}\,C\,\{\psi\}$, is both natural and convenient.

The main reason why Hoare triple are so convenient is that they include directly code, C, whereas BI Pointer Logic is formulated wholly in terms of properties of the contents of

memory. We connect these two points of view by providing a semantics of Hoare triples in terms of BI Pointer Logic [Calcagno et al., 2007]. There are essentially two ways of going about this, depending upon on the strength of requirements on the behaviour of the code. The behaviour of code is expressed in terms of the evaluation of a program C — using stack s and heap h — with respect to sequences of steps defined by its operational semantics, $\rightsquigarrow$, and essentially denoted by $C,s,h \rightsquigarrow^* s',h'$, read as 'the program C transforms the memory configuration s,h into the memory configuration s',h'. There is a special configuration, *fault*, indicating a memory fault or abnormality.

The first semantics for Hoare triples [O'Hearn and Yang, 2002], called partial correctness, relies on the notion of safety,

$$C,s,h \text{ is safe if } C,s,h \not\rightsquigarrow^* \mathit{fault}$$

and is the 'fault-avoiding' interpretation, as explained in [O'Hearn and Yang, 2002]:

> *Partial correctness semantics*: $\{\phi\}C\{\psi\}$ is true in a model of Pointer Logic if, for all s,h, $s,h \models p$ implies
>
> – C,s,h is safe, and
>
> – if $C,s,h \rightsquigarrow^* s',h'$, then $s',h' \models q$.

The second, called total correctness [O'Hearn and Yang, 2002], does not require the safety condition because it requires the 'stronger' property of 'normal' termination; that is, the program returns a value that lies within its intended range of outputs:

> *Total correctness semantics*: $\{\phi\}C\{\psi\}$ is true in a model of Pointer Logic if, for all s,h, $s,h \models p$ implies
>
> – C,s,h must terminate normally, and
>
> – if $C,s,h \rightsquigarrow^* s',h'$, then $s',h' \models q$.

With these definitions, and some non-trivial technical development, soundness (that the rule transforms true properties into true properties) and completeness (that the rule derives one specification statement from another just when this inference holds semantically) theorems for the Frame Rule,

$$\frac{\{\phi\}\, C\, \{\psi\}}{\{\phi * \chi\}\, C\, \{\psi * \chi\}} \ \text{Modifies}(C) \cap \text{Free}(\chi) = \emptyset,$$

can be established [O'Hearn and Yang, 2002]. These theorems give precise mathematical expression to the coincidence of the logical and engineering models of computer memory allocation.

In this section, we have provided some detail on the novel aspects of Separation Logic's semantics, and how they support reasoning about computer memory as a resource. At heart, the atoms of the logic are composable in a way that mirrors the way that the physical substrate is composable. The physical transistors come apart, and one can make meaningful claims about affixing or pulling apart bits of silicon that have reliable impacts on the changes to the electrical and computational properties of the physical system. The structure of the logical model using partial commutative monoids and $*$ that we have introduced allows for logical claims to naturally mirror this physical fact.

The following section details the cluster of properties surrounding the proof theory of Separation Logic that make it a successful engineering tool. Part of these also relate to the composability of $*$ through the Frame Rule, as it is leveraged for efficient computation of results. Equally important to the deployability of the proof theory is the automation of bi-abduction for generating hypothetical pre- and post-conditions to drive proof solutions. The abductive rules we use are essentially encodings of engineer's heuristics when reasoning about computer memory usage, further demonstrating the deep ways in which the logical and engineering aspects of the task merge in Separation Logic.

5 Deployable Proof Theory for Separation Logic

In Section 4.3, above, we explained that we can obtain soundness and completeness properties for the Frame Rule; that is, the Frame Rule exactly characterizes logical truth for local reasoning about memory allocation.

An important consequence of a system of logic having a completeness theorem is that its a proof system can be used as a basis for formal reasoning within it. Consequently, the study of the automation of proof systems — that is, the provision of computationally feasible presentations of proof systems — is a widely studied topic in modern logic. Perhaps the most famous example is the provision of resolution systems for the Horn clause fragment of first-order predicate logic [Robinson, 1965, Van Emden and Kowalski, 1976], the basis of the programming language Prolog [Hodgson, 1999].

Such a proof system might be described as *deployable*. That is, the search for, and construction of, proofs in the system is computationally tractable. For a problem to be tractable, the compute resources (these days, mainly time) required to make the calculations are acceptable for their intended use. This section discusses the intellectual choices that make Separation Logic deployable. Actual deployment requires integration with software engineering practices. Integration is not trivial; the deployment challenges associated with Infer are described by O'Hearn [2015].

In the setting of Separation Logic, we have a deployable proof system for a semantics that captures simultaneously the engineering model of computer memory allocation and its logical interpretation. There are two key properties that a proof theory ought to have to be deployable: scalability and automation. Separation Logic achieves these engineering-type implementation goals through features built in to the logic. Scalability comes mainly from access to the Frame Rule, and the parallel computation which it enables. Automation of

proofs is a large topic on its own, with tools such as Coq. However, Separation Logic has also been used to make systems that can automate reasoning about what is relevant to attempt to prove — that is, abduction. Automating abduction in this context means formalising heuristics engineers use to diagnose errors. The logical system must be tailored to accomplish this task.

5.1 Separation Logic and the Frame Rule

The formal definition of the Frame Rule for Separation Logic was introduced in Section 4.3. The 'frame' in the Frame Rule is essentially a context; formally a set of logical statements; and, in the practice of software engineering, it is the variables and memory resources that a program modifies. The Frame Rule lets the analyst break a program into disjoint fragments, analyse them separately, and cleanly and quickly conjoin the results. This is because, as long as the frame and the program do not modify each other's variables, the Frame Rule tells us that we can freely conjoin the frame to the pre- and post-conditions for the program.

Let us return to our drinks-as-resources analogy. If the 'program' we are interested in is I drink my drink, a sensible pre-condition is that I have a full drink. The post-condition is, let's say, that I have an empty glass. The frame then is all the other drinks in the restaurant, as well as the food, and the sunshine outside, as long as there is no joker in the place going about pouring people's drinks into one another's glasses. In computer programming, we can check rigorously for such jokers because we can check what variables (in this example, the glasses) different programs can access.

The benefits for scalability, and therefore deployability, are immediate. Imagine if one had to re-analyse one's 'program' for drinking a glass of water every time another patron entered or exited the restaurant, or any time any other patron refilled or finished their own

drink. In program verification, this is a serious threat to viable tools. Programs change often, and are expensive to analyse wholesale. It is not plausible to reanalyse a whole program for each minor change. The Frame Rule gives Separation Logic a deployable proof theory for two reasons. First is the facility it provides for the saving of results from past analyses of unchanged program fragments and applying them quickly to analyse of small changed fragments. The second reason is perhaps more subtle, but more powerful. Modern computing is done largely in clouds owned by giant tech companies. The benefit of cloud computing is that hundreds of thousands of processors can work on a computation in parallel and merge their results. Without the Frame Rule, Separation Logic would not be able to take advantage of the massive computational resources of cloud computing; parallelization requires fragmentation of a problem into smaller parts and sound merging of results.

5.2 Deployability via Contextual Refinement

The Frame Rule is not the only method of developing a deployable proof theory for Separation Logic. Xu et al. [2016] describe an extension of Concurrent Separation Logic [O'Hearn, 2007] that uses contextual refinement between implementation and specification of a program to prove the correctness of the program. Contextual refinement is a formal specification of the following relationship: the implementation, i.e., the actual written computer code, does not have any observable behaviours that the abstract design specification of the system does not have.

Xu et al. [2016] deploy Separation Logic to verify the scheduling behaviour of operating system kernels. The kernel is the most trusted, central part of the operating system that coordinates all other applications' access to the physical hardware. This application of Separation Logic, like Infer, treats computer memory as a resource. However, the relevant

property of memory in this application is unique ownership by a task, rather than unique identification of a memory location by a pointer. This distinction aims to overcome the main difficulty in scheduler design, which is ensuring that two programs that both hold the same pointer do not interfere with each other. The technical details are out of scope; however, this is a common and challenging computer science problem. In order to make efficient use of hardware resources, complex scheduling has been common in operating systems since the mid 1990s. Deployable verification of the scheduling for a real-world (preemptive) operating system kernel uses Separation Logic [Xu et al., 2016].

The design of a logic to verify operating system scheduling contains many of the same strategic features as are evidenced in the development of Infer. The logic is tailored to the problem at hand to the extent that 'the interrupt mechanism in our operational semantics is modeled specifically based on the Intel 8259 A interrupt controller, and the program logic rules for interrupts are designed accordingly' [Xu et al., 2016, p. 77]. In order to arrive at a satisfactory semantics, the authors modelled the behaviour of a specific processor on specific Intel hardware. This quite clearly demonstrates the merging of the logical model and the engineering model. The inference rules Xu et al. [2016] uses are quite different from those used by Calcagno et al. [2011]. In the section that follows, we focus on the use of inference rules over memory allocation via Calcagno et al. [2011]; there are analogous rules for operating system scheduling which we elide [Xu et al., 2016, p. 72].

5.3 Bi-abduction

We briefly discussed, in Section 3, the importance for the effectiveness of Separation Logic of the concept of abduction. In this section, we give a introduction to how it is integrated into

the logic. Abduction was introduced by Charles Peirce around 1900, when writing about the scientific process, and explained by Peirce as follows:

> 'Abduction is the process of forming an explanatory hypothesis. It is the only logical operation which introduces any new idea' [Bergman and Paavola, 2016, CP 5.171].

Consider a non-technical example. A baby, perhaps yours, is crying for no obvious reason. Approaching the problem like an engineer, we should like to know the source of the baby's distress, so that we can devise a method to allay it. But as we did not see the child begin to cry, we must guess at, or abduce, the source. Perhaps we abduce that a malicious Cartesian demon is making the baby believe it is being viciously pinched. Or perhaps we guess hunger is the source. Neither are entirely new ideas, both suggested by our past experience with and structure of the world. Yet we prefer the abduction of hunger, if for no other reason than we have a ready method to allay hunger on hand, and none such for demons. That is, we can test whether the baby is hungry by feeding it. We can guess at the post-condition we should reach from this intervention if the precondition is true: if the baby is hungry, and we feed it, then the baby will stop crying. If we feed the baby and it does not stop, we surmise our guess failed and we must abduce something else. Thus, even though there are incalculably many conceivable causes of the baby's crying, the structure of the situation suggests certain abductions. Knowing, or abducing, what should or might be true of conditions after a process or intervention puts constructive constraints on our abductions of prior conditions.[10]

[10] Of course, what we describe here is not solely abduction. Our description also relies on some sort of structured knowledge and the ability to manipulate models of how parts of the world might interact. Structural reasoning [Swoyer, 1991] and mechanism discovery [Bechtel and Richardson, 1993] perhaps play a role. However, we focus only on abduction as that is the feature that Separation Logic clearly identifies as automated.

There is not a general process by which one generates useful new ideas. However, if one has both a precise language and a detailed conception of the mechanisms of interest in the system, abduction becomes more tractable. Since we have these in our logic and in our engineering model of computer memory, respectively, and further we have a fast and composable method for soundly checking the correctness of the guesses from abduction, we can automate abduction in the case of looking for pre-conditions and post-conditions that lead to memory errors in computer code.

The formalization of abduction in classical logic is, deceptively simply, as follows:

Given: assumption ϕ and goal ψ;

Find: additional assumptions χ such that $\phi \wedge \chi \vdash \psi$.

In this expression it is customary to disallow trivial solutions, such as $\phi \rightarrow \psi$. When reasoning about computer memory and pointers, we use the separating conjunction in the obvious analogue:

Given: assumption ϕ and goal ψ;

Find: additional assumptions χ such that $\phi * \chi \vdash \psi$.

Because our problem domain is program analysis and specifically the program's use of memory, we constrain χ to be a formula representing a heap. This constraint disallows trivial solutions such as $\phi \mathrel{-\!\!*} \psi$ [Calcagno et al., 2011, p. 6].

To contribute genuinely to a deployable proof theory, we need to know both the pre-conditions necessary for the piece of code to run safely and also all the logical conditions that will be true after the piece of code finishes. Post-conditions for a single piece of code do not help to verify that particular piece of code. However, computer programs are complex arrangements of separable but interrelated pieces of code. The post-conditions of one segment are good candidate guesses for pre-conditions of other segments. Calcagno et al. [2011]

coin the term bi-abduction for finding both pre- and post-conditions. In program analysis, the pre-conditions are the anti-frame and the post-conditions are the frame, so bi-abduction is formalized as follows:

Given: assumption ϕ and goal ψ;

Find: additional assumptions ?anti-frame and ?frame such that $\phi * ?\text{anti-frame} \vdash \psi * ?\text{frame}$.

The statement's specific logical form, our model of the mechanism of computer memory use by programs, and the machine-readable nature of our domain of interest—computer programs, all combine to allow us to automatically generate potential solutions to the frame and anti-frame. The result of this synthesis of features makes bi-abduction 'an inference technique to realize the principle of local reasoning' [Calcagno et al., 2011, p. 8].

Let us step through bi-abduction in some examples. First we discuss ascertaining pre-conditions in some detail; post-conditions we touch more lightly. We do not assume any familiarity with C or with programming, so we explain the target program segment in English detail.

The example used by Calcagno et al. [2011, p. 8] to explain abduction is:

```
void free_list(struct node *x) {
  while (x!=0) {
  t=x;
  x=x->tl;
  free(t);
}
```

Our example program steps through or traverses all the elements of a list and removes them. Literally, it frees the memory used to store each element.

Let's use the example of a shopping list, for concreteness. Traversing a list is just to read all the elements in order. For a paper list, this ordering is handled by the physical layout on the paper. Eggs are first if they are on the first line. In computer memory, a directly analogous physical layout is difficult and inefficient for technical reasons. Instead each list element contains two parts. First, its contents, say 'eggs', and second, a pointer to the next element. Pointers, as discussed in Section 2, can cause myriad problems during a program's execution. Such linked lists are a common place to find pointers, and so a common place to find memory management errors.

When verification tools encounter a program like `free_list`, they start off assuming an empty heap (*emp*) and that the variable x has some value X. However, at the line '`x=x->tl`' the reasoning stalls. There needs to be some X' to which X points. Using abduction, the tool guesses that such an X' exists. Another step is required. In the general case, we will hit an infinite regress of assuming ever-more X'', X''', and so on. Separation Logic requires an abstraction step, which Calcagno et al. [2011, p. 9] link to a scientific induction step. The abstraction step is to posit a list of arbitrary length from X to X' and to assert or abduce that a program that works of lists of length 4 probably works on lists of length 6. The trick is to encode these heuristics, such as the guess that an X' exists, into formal proof rules that can be applied automatically. Abduction and abstraction potentially weaken preconditions. Weakening may be unsound, and must be tested. But such tests can also be automated in Separation Logic. Calcagno et al. [2011, p. 10] describe perhaps 50 pages of their article as 'devoted to filling out this basic idea [of using abduction to guess what good preconditions might be]'. We discuss one further example to illustrate some of the complications that can arise in the task.

Lists can get more complicated. For example, the last element can link back to the first. Imagine taping a shopping list into a loop, so that 'eggs', our first element, came on the line

after our last element, ‘chocolate’. The C syntax of such a program is [Calcagno et al., 2011, p. 53]:

```
void traverse-circ(struct node *c) {
  struct node *h;
  h=c; c=c->tl;
  while (c!=h) { c=c->t1;}
}
```

We human shoppers would not start over and traverse the list again, picking up a second copy of everything on the list. And then a third, looping through the list until our cart overflowed. However, `free_list` would naïvely enter such an infinite loop. So `traverse-circ` not only reads an element and goes to the next one, but remembers where it started so that it can stop after going through once. Since the program is designed to read circular lists, we should expect our logic to produce a circular list as a pre-condition. This is the case. Specifically, we abduce the precondition [Calcagno et al., 2011, p. 52]

$$c \mapsto c\_ * list(c\_, c)$$

That is, for the program to run safely, the input (c) must be a pointer to a valid element of memory ($c\_$), and separately there must be a linked list going from that valid element back to the initial element.

Let us explore in more detail the formal form of this abduction, which is Algorithm 4 in [Calcagno et al., 2011, p. 37]. The algorithm is run (by another computer program) with our small program of interest as input, along with a guess at the starting state. The first steps of the algorithm build a logical model of the program’s interaction with memory. The logical model takes the form of Hoare triples. How exactly a computer program is soundly converted into Hoare triples is a matter of shape analysis, or ‘determining “shape invariants"

for programs that perform destructive updating on dynamically allocated storage' [Sagiv et al., 2002]. There are technical details about converting the program to a logical model that are out of scope here, but note that our logical model and language are purpose-built tools for this task. Going back to Hoare's explicit axiomatization of programs [Hoare, 1969] through to the definition of $\mapsto$ for the function of a stack element pointing to a location in the heap, both broad strokes and finer details of the logic are responsive to the problem at hand.

After constructing the logical model, Algorithm 4 iterates through all of the Hoare triples and calls, AbduceAndAdapt [Calcagno et al., 2011, p. 43]. This function has two main purposes: to do bi-abduction, and to take any successful results from bi-abduction and 'perform essential but intricate trickery with variables' to maintain precise results. The abduction aspect of the algorithm is specified in Algorithm 1. This algorithm, in turn, depends upon a set of proof rules used in reverse as abduction heuristics [Calcagno et al., 2011, p. 15-17]. The rules are all of a special form,

$$\frac{H_1' * [M'] \triangleright H_2' \quad Cond}{H_1 * [M] \triangleright H_2}$$

Here *Cond* is a condition on the application of the rule based on parts of H_1 and H_2. The proof rules can thus be read backwards to create a recursive algorithm that will eventually abduce pre- and post-conditions. To read them in this manner, the algorithm checks that the condition holds. If so, instead of answering the (harder) question $H_1 * ?? \vdash H_2$, the algorithm goes on to search for the answer to the (simpler) abduction question $H_1' * ?? \vdash H_2'$ [Calcagno et al., 2011, p. 17].

The example at hand, traverse-circ, will hit the heuristic 'ls-right' until the list loops, generating the precondition that there is a list from $c\_$. The other precondition is generated

by the heuristic '$\mapsto$-match'. These are linked in the 'intricate trickery' done in the algorithmic step to keep results precise.

The details of which proof rules are chosen as abduction heuristics is important and non-trivial. The choice is based on decades of prior experience and empirical results on the effectiveness of different modelling choices. Our main point at present is to remark on the extent to which the logic has been shaped to be a tool to solve the engineering problem at hand such that the proof rules are chosen empirically.

The postconditions of this example seem less exciting. The program only reads the list, it does not output any contents nor change it. Therefore, the abduced post-conditions will be the same as the preconditions. While this initially seems unenlightening, remember that bi-abduction is on program segments, not whole stand-alone programs. So if a larger, more realistic program runs this traverse-circ process successfully, and it had the necessary preconditions, we can be sure that there is a circular linked list in memory. This information may be very helpful for determining whether another program segment runs safely. For example, a process that deletes elements of a list one at a time often has the flaw that it will not check for circular lists. When such a delete process cycles, it will try to delete the now non-existent first list-element, causing a memory error that can result in a crash. In such a situation, this post-condition of a circular linked list would be informative. For more details on how to abduce postconditions, see Algorithm 6 in Calcagno et al. [2011].

Abduction is automatable in this situation because the problem space investigated by the engineering/scientific model is quite precisely defined. Instead one might say that abduction is automatable here because the logical model sufficiently accurately represents the behaviour of real computer programs. These two assessments are both true, and amount to the same thing: effective merging of the features of the logical model and the conceptual model. Automated abduction is a striking example of the benefits of such a confluence.

The best measure of whether a proof theory is deployable for finding errors in software is whether programmers in fact fix the errors it finds. For programmers to fix errors, the tool must provide a combination of timely results, precise results, and clear explanations. These are part of usefulness requirements within the industrial software engineering setting that are essentially social or organizational [Calcagno et al., 2015a]. Therefore, what counts as a satisfactory fix-rate may change from one organization to another. Infer is open-source and used by many organizations. Separation Logic is measured as deployable in some sense because it is deployed in these contexts. In this paper we focus on the technical aspects of the logic that have made it deployable. For an account of the social and practical environment necessary to shepherd Infer to deployment, see Calcagno et al. [2015a].

In Section 2 we detailed why finding memory usage flaws is an important task in computer programming. Programmers make these errors, and in products that are widely used. Further, these kinds of errors impact stability and security in costly ways that are hard to catch and handle during execution. Separation Logic has been tailored to this problem specifically, through adaptations to both its semantics (detailed in Section 4) and proof theory. In this section, we have detailed how the proof theory has been made deployable, to meet the needs of industrial application. It is deployable because (1) its reasoning is scalable and fast, using the compositionality of the Frame Rule; and (2) its generation of hypothetical pre- and post-conditions is automated using encoded discovery heuristics and bi-abduction.

6 Conclusion

We have introduced Separation Logic as a tool for reasoning about computer programs, specifically their use of memory as a resource. This history provides insight to philosophers of science, logicians, and computer scientists based on the methodology that makes Sepa-

ration Logic successful. Namely, that the logic model overlaps with the conceptual model of a practical problem and the proof theory is usefully deployable. Philosophers of science may view this convergence as a tactic for model building. There are benefits to both the logical and practical problems by working towards tightly integrated logical-cum-engineering solutions.

The type of errors that Separation Logic is currently used to find are constrained to a specific, though important, type of catastrophic run-time error. We have identified two types of run-time errors — memory allocation [Calcagno et al., 2011] and task scheduling [Xu et al., 2016] — that have been addressed with Separation Logic. These types of errors arise in a variety of applications, from hardware synthesis [Winterstein et al., 2016] to computer security [Appel, 2015] to popular phone apps. Separation Logic is not the solution to all computer science problems, but it is not so specific as to be uninteresting.

Other specific problems will very likely require logics tailored to them. As one example, Lamport [2002] details temporal logic which is used by Amazon for its network architecture [Newcombe et al., 2015]. Another aspect of assuring memory, called shared memory consistency, used yet a different logic model to address its programming problem [Adve and Gharachorloo, 1996]. These other examples of success by bringing a programming/engineering model into close contact with an adequately designed logic model strengthen our conclusion. The history of Separation Logic, through to its implementation in deployed verification tools, demonstrates that such overlap is an effective strategy for reasoning about the behaviour of computer systems. See O'Hearn [2015] and Calcagno et al. [2015a] for accounts of the software-engineering effort involved in deploying one such tool.

It is important to understand the extent to which the case of Separation Logic is relevant to both computer-science models and science more generally. Model-based reasoning in computer science seems to come in at least two flavors. Some parts of computer science,

like human-computer interaction and usable security, have methodologies that are closely adapted from established fields like psychology [Krol et al., 2016]. However, in other parts of the field, computer-science methods are distinctly developed within the discipline. Even so, Hatleback and Spring [2014] argue that experiments and model-building in computing are not so different from other sciences, after accounting for the unique challenges of the fields. Separation Logic provides a good example of this second type; the above examples of temporal logic and shared memory consistency indicate it is not alone. Hatleback and Spring [2014] argue that reasoning about objects that can purposefully change at the same pace as the observer can interact with them, namely software, is a particular challenge in computer science. Separation Logic is an example of how computer scientists overcome this problem. Reasoning at the appropriate level of abstraction produces stable representations of the phenomenon so that conclusions are reliable. The challenge of making reliable generalizations is not unique to computing; Spring and Illari [2017] argues that computer security, at least, handles the challenge in substantively the same mode of reasoning as biology does. In all of these disciplines, reasoning about exhaustible resources often matters; in this regard, beyond any similarities with mode of reasoning, the mechanics of Separation Logic may be applicable. Therefore, the case of Separation Logic is similar to many other aspects of computing, and computing is likely similar enough science more generally, that this instance of reliable model-building by combining logic models and conceptual models may carry widely-applicable lessons.

Our approach would not get off the ground without a deployable proof theory, no matter how nice the overlap between the model of computer memory and the logical interpretation of exhaustible resources. In fact, exploiting the model structure for some practical benefit, such as timely parallel computation, is perhaps more rare — and more important — than devising a model that is simultaneously a logic and an engineering model. Verification tools

using Separation Logic reach a deployable proof theory due to a constrained domain that permits the automation of abduction combined with a composable logic that permits reuse of results. In this regard, the logical machinery we have detailed that enables these features should be of technical interest to logicians outside computer science. We have focused this technical development in Section 4. The main points are (1) the introduction of the logic of bunched implications, which admits the usual conjunction with contraction and weakening rules and a different conjunction that does not; (2) the semantics of a resource as a preordered partial commutative monoid; (3) a full definition of the connectives $*$ and $-\!\!*$.

Philosophical logic has a long tradition of analysis of arguments and meaning. One message we have for logicians is that it can have more clarity and impact when the model theory is grounded in concrete engineering or scientific problems; that is, where the elements of the model have a clear reading or interpretation apart from their role in defining the semantics of sentences. For example, relevant logicians have admitted to struggles in interpreting the meaning of the elements in their formal semantics based on ternary relations [Beall et al., 2012]. Their semantics enjoys completeness theorems with respect to their proof theories, but the subject matter of the models themselves is not evident. In contrast, as we have shown here there is a nearby semantics, not identical, where the model elements are understood in terms of the structure of computer memory — and more generally of resources [Pym et al., 2004]. These arise independently of the logic, which gives them all the more semantic force. Moreover, by looking at the model, novel proof-theoretic ideas emerge, such as the Frame Rule. In general, when the semantics of logics meets independently-existing science and engineering, a feedback cycle can be set up which impacts both to mutual benefit.

Logic, like any other technology, must be designed to specifications for the task at hand. In concert with design, the logic employed should be empirically tested as to whether it meets specifications. This sort of feedback loop is not so different from the tool-building and

scientific modelling interaction in other fields. However, unlike, say, biology whose tools are often of glass and metal, the tools in computer science are often conceptual or logical tools. Considering computer science as the field that explores the human-created abstractions of mathematics and logic, this tooling change makes sense. Moreover, the understanding that just because we humans have built or defined some system it does not automatically follow that we know all the properties and behaviours of said system perhaps elucidates why computer science can often usefully be considered an experimental science. In this way, Separation Logic is a useful test case for applying concepts from philosophy of science to computer science.

Separation Logic is also a useful case for communicating salient aspects of computer science to the broader philosophy of science community. The technical details of a logic for exhaustible resources is one contribution that logicians in many fields may find applicable. Further, for the debate on model-based reasoning, Separation Logic is an automatable system for model-based reasoning, albeit in a tightly constrained environment. Perhaps such extensive context constraints are necessary to formalize reasoning to the level of detail necessary for automation. However, the case study provides a starting point from which philosophers may be able to generalize broader lessons for model-based reasoning.

Our case study of the success of Separation Logic for reasoning about memory as a resource indicates that further work in the direction of appropriately integrating the right logic as a tool in empirical modelling should bear further fruit. The various deployments of Separation Logic in tools demonstrate the extent to which the conceptual/engineering model and requirements may intertwine with the logic's model and proof theory for great success.

Acknowledgements

Removed for blind review

References

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

Andrew W Appel. Verification of a cryptographic primitive: SHA-256. *Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015.

Andrew W Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program logics for certified compilers*. Cambridge University Press, New York, 2014.

Krzysztof R. Apt. Ten years of Hoare's logic: a survey—Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

Jc Beall, Ross Brady, J. Michael Dunn, Allen P. Hazen, Edwin Mares, Robert K. Meyer, Graham Priest, Greg Restall, David Ripley, John Slaney, and Richard Sylvan. On the ternary relation and conditionality. *Journal of Philosophical Logic*, 41 (3):595–612, 2012.

William Bechtel and Robert C Richardson. *Discovering complexity: Decomposition and localization as strategies in scientific research*. Princeton University Press, Princeton, NJ, 1st edition, 1993.

Mats Bergman and Sami Paavola. 'Abduction': term in The Commens Dictionary: Peirce's Terms in His Own Words. New Edition. http://www.commens.org/dictionary/term/abduction, Jul 14, 2016.

Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, number 1837 in LNCS, pages 102–126. Springer, 2000.

Frederick P Brooks Jr. *The mythical man-month: Essays on software engineering*. Addison Wesley, 2nd edition, 1995.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Logic in Computer Science*, pages 366–378. IEEE, 2007.

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, 2011.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, number 9058 in LNCS, pages 3–11. Springer, 2015a.

Cristiano Calcagno, Dino Distefano, and Peter W. O'Hearn. Open-sourcing Facebook Infer: Identify bugs before you ship, 11 June 2015b. https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *25th Symposium on Operating Systems Principles*, pages 18–37, Monterey, CA, Oct 4, 2015. ACM.

J. Michael Dunn and Greg Restall. Relevance Logic. In Dov M. Gabbay and F. Guenther, editors, *Handbook of philosophical logic*, volume 6, pages 1–128. Springer Netherlands, Dordrecht, 2002.

Federal Aviation Administration. FAA Statement on Automation Problems at Washington Center. https://www.faa.gov/news/press_releases/news_story.cfm?newsId=19354, Aug 17, 2015.

Luciano Floridi, Nir Fresco, and Giuseppe Primiero. "on malfunctioning software". *Synthese*, 192(4):1199–1220, 2015.

Roman Frigg and Stephan Hartmann. Models in science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2012 edition, 2012.

Didier Galmiche, Daniel Méry, and David Pym. The semantics of BI and resource tableaux. *Mathematical Structures in Computer Science*, 15(06):1033–1088, 2005.

Ronald N. Giere. How models are used to represent reality. *Philosophy of science*, 71(5):742–752, 2004.

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

Eric Hatleback and Jonathan M. Spring. Exploring a mechanistic approach to experimentation in computing. *Philosophy & Technology*, 27(3):441–459, 2014.

Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10): 576–580, 1969.

J.P.E. Hodgson. Project "Contraintes" Prolog Web Pages: The ISO Standard, Mar 1, 1999. URL http://http://www.deransart.fr//prolog/overview.html.

Phyllis McKay Illari and Jon Williamson. What is a mechanism? Thinking about mechanisms across the sciences. *European Journal for Philosophy of Science*, 2(1):119–135, 2012.

Samin S. Ishtiaq and Peter W. O'Hearn. BI As an Assertion Language for Mutable Data Structures. *SIGPLAN Not.*, 36(3): 14–26, Jan 2001.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1988.

Saul A. Kripke. Semantical analysis of intuitionistic logic I. *Studies in Logic and the Foundations of Mathematics*, 40: 92–130, 1965.

Kat Krol, Jonathan M. Spring, Simon Parkin, and M. Angela Sasse. Towards robust experimental design for user studies in security and privacy. In *Learning from Authoritative Security Experiment Results (LASER)*, pages 21–31, San Jose, CA, 2016. IEEE.

Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

MITRE. Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types v2.9. http://cwe.mitre.org, Dec 2015.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

Peter W. O'Hearn. From Categorical Logic to Facebook Engineering. In *Logic in Computer Science (LICS)*, pages 17–20. IEEE, 2015.

Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(02):215–244, 1999.

Peter W. O'Hearn and Hongseok Yang. A Semantic Basis for Local Reasoning. In *Proceedings of the 5th FoSSaCS*, number 2303 in LNCS, pages 402–416. Springer, 2002.

Gualtiero Piccinini. Computing mechanisms. *Philosophy of Science*, 74(4):pp. 501–526, 2007. URL http://www.jstor.org/stable/10.1086/522851.

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

Stephen Read. *Relevant Logic: A Philosophical Examination of Inference*. Basil Blackwells, 1988. URL https://www.st-andrews.ac.uk/~slr/Relevant_Logic.pdf.

John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

Viola Schiaffonati and Mario Verdicchio. Computing and experiments: A methodological view on the debate on the scientific nature of computing. *Philosophy & Technology*, 27(3):359–376, 2014.

Herbert A Simon. *The sciences of the artificial*. MIT press, Cambridge, MA, 3rd edition, 1996.

Aditya K. Sood and Richard J. Enbody. Crimeware-as-a-service: A survey of commoditized crimeware in the underground market. *International Journal of Critical Infrastructure Protection*, 6(1):28–38, 2013.

Jonathan M. Spring and Phyllis Illari. Mechanisms and generality in information security. *Under review – Philosophy & Technology*, 2017.

Jonathan M. Spring, Tyler Moore, and David Pym. Practicing a science of security: A philosophy of science perspective. In *New Security Paradigms Workshop*, Santa Cruz, CA, Oct 2-4, 2017.

Mauricio Suárez. Scientific representation. *Philosophy Compass*, 5(1):91–101, 2010.

Chris Swoyer. Structural representation and surrogative reasoning. *Synthese*, 87(3):449–508, Jun 1991.

Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363): 5, 1936.

Dirk van Dalen. *Logic and structure*. Springer-Verlag, 4th edition, 2004.

Maarten H. Van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23 (4):733–742, 1976.

Jules Vuillard. Blog post on Infer-Spotify Collaboration, 17 March 2016. http://fbinfer.com/blog/2016/03/17/collaboration-with-spotify.html.

Felix J Winterstein, Samuel R Bayliss, and George A Constantinides. Separation logic for high-level synthesis. *Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(2):10, 2016.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In *Computer Aided Verification (CAV)*, number 9780 in LNCS, pages 59–79, Toronto, Ontario, Jul 2016. Springer.