Chapter 1

Introduction

In the well-known 1960s article *The unreasonable effectiveness of mathematics in the natural sciences* by E.P. Wigner [225], and in the 1980 article *The unreasonable effectiveness of mathematics* by R.W. Hamming [106], the authors argue that mathematics allows for the formulation of theories—useful for making predictions in the natural sciences, such as physics and astronomy—and as such is the correct language in which to express concepts of regularity in patterns of observations (see also [41]). We, humans, tend to prefer those theories that can elegantly and beautifully explain many phenomena, accurately. Wigner modestly suggests that every theory is wrong in *some* way, and that it is just a matter of time until it is discovered *what* is wrong about any theory; eventually the better theories supersede the worse ones. Simply put, this allowed us to progress from Galileo's falling bodies, to Newton's mechanics, to Einstein's relativity. The reason for the success of mathematics was 'mysterious' for Wigner, and remained elusive for Hamming.

In 2001, in a similar vein as Wigner and Hamming, J.Y. Halpern, R. Harper, et alia [105] argue in their article *On the unusual effectiveness of logic in computer science*, that it is not elegant mathematics, per se, that drives progress; the driving force of progress in (theoretical) computer science, they argue, is *mathematical logic*, as it developed from the foundational crisis of mathematics late 19th century until early 20th century: "logic has had a definite and lasting impact" on computational complexity, relational databases, typed programming languages, knowledge representation in distributed systems, verification of semiconductor designs, among many other areas in computer science. Contrary to Wigner and Hamming, these authors *do* offer an explanation for the unusual effectiveness of logic. Essentially, their argument boils down to recognizing the difference between the natural sciences and computer science—in the sense that most knowledge in computer science is 'synthetic': the organization of computing, down from the design of semiconductors, processor architectures, programming languages, operating systems, all the way up to the working of the Internet and the world-wide applications running atop: everything is entirely man-made. Simply put, since logic is a way of organizing human thought, it as such offers a suitable framework in which the development of computer science can progress.

One successful application of logic in computer science is in the subject of *program verification*,[1] in which one proves properties of programs in a similar way as one would do deductive reasoning in logic. Program verification rapidly gained interest by the seminal articles *Assigning meanings to programs* by R.W. Floyd in 1967, and *An axiomatic basis for computer programming* by C.A.R. Hoare in 1969, where in the latter "sets of axioms and rules of inference which can be used in proofs of the properties of computer programs" are introduced [84, 118]. Also many books on this subject have been written in the past fifty years, see e.g. [71, 61, 100, 86, 10]. By verifying the *correctness* of a program one establishes that a program's behavior has certain desirable properties. The correctness of a program also depends on the correctness of the compiler or interpreter necessary for running the program, the operating system on which the (compiled) program depends, the processor architecture, and ultimately the semiconductor design of all the components in a computing machine. If all of these man-made entities are correct, established with mathematical certainty, then Hoare claims that "it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics." Thus, what Hoare suggested, in a clear way, was that the subject of program verification is foundational to computer science, since it reduces the question of the reliability of our man-made organization of computing ultimately into theories of nature itself: the physical properties of circuits that realize computing.

Although program verification is a deductive method, there is an alternative—a more experimental method: to try a program on a number of different test cases, and see whether the observed program behavior is desirable. The study of this process, called *program testing*, also developed into a respectable subject of computer science [160, 7, 130]. However, there is a limitation to program testing, since from trying out a (finite) number of test cases one cannot always conclude to have ruled out all programming errors, i.e. testing does not always guarantee correctness. Hence, in practice, some programming errors remain, since the cost of discovering all programming errors by means of testing is too large. Discovering programming errors when programs are already put into use might even be more problematic, since then those errors could be costly to remove [104, 128, 129], and cause (irreversible) effects and damage in the mean time [234]. Hoare posits the important role that program verification can play, by prophesying that "the cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed airplane, or a world war." [118] Hoare wrote this in the middle of the cold war, around the same time the U.S. accomplished the first manned moon landing.[2]

A crisis in computer science emerged, later dubbed the *software crisis*, that started late 1960s [183] and lasted until the turn of the millennium [117]. Although computing machinery was improving at an incredibly rapid pace, problems surfaced with the development of software: early estimates indicated that professionally developed programs could contain many programming errors, between one in every

[1]Alternatively: program correctness, software correctness, software verification

[2]See also `https://www.vpro.nl/speel~POMS_VPRO_212868~denken-als-discipline~.html`

hundred to one in every thousand lines of code [117]; recent estimates confirm this still holds today [233, 198]. Such estimates are especially worrisome in safety-critical software, applied in important and impactful high-tech industries such as energy, transportation, and military [144]. The idea that a small programming error in these kinds of applications can have devastating and far-reaching consequences is surely frightening. To treat the ailments of the software crisis, new programming languages and techniques were developed, resulting in the so-called high-level programming languages [30]. The design of high-level programming languages incorporated (lightweight) verification techniques [226, 124, 80], such as type checkers and user-definable data types. Also new ways of organizing the software production endeavor were introduced, and the people involved in these new software development processes, taking care of producing high-quality software, were called *software engineers*.

And with great success: at the end of the software crisis, also Hoare recognized that most safety-critical software had acceptable reliability even though most software engineers did not apply program verification as he envisioned it in 1969. In fact, while reflecting on the role program verification can play, Hoare openly admitted that there was a "non-fulfillment of prophecies of doom. The history of science [...] is littered with false predictions and broken promises; indeed they seem to serve as an essential spur to the advancement of human knowledge; and nowadays, they are needed just to maintain a declining flow of funds for research." [117] Hoare wrote this in the middle of the roaring nineties, in stark contrast with his earlier sentiment.

However, if we have a look at the case for program verification today, have there been fundamental shifts in the playing field of software development in the past thirty years or so? Should we take novel 'prophecies of doom' [204] with a grain of salt? Or, are we on the verge of collapse, and is there again a need to reinvigorate the practice of rigorous program verification [103]? Can we be realistic about the use of program verification—neither too optimistic nor too pessimistic [137, 92]? We can not expect to obtain reasonable answers to all of these questions, but we can recognize the following rough strokes on the canvas of recent computer history:

- In the last thirty years, we have seen the immense growth of the Internet (including its associated costs such as energy usage and corresponding CO_2 emissions [195]), and with it the deployment of many world-wide applications running on top such as the Web (including associated downsides, which are hard to specify formally, such as the spreading of misinformation [60]). As a result, societies in developed countries have witnessed, especially now we are beyond the horizon of 2020, the so-called *digital transformation* of the past decades. This incredible development was in part caused by, but also affected, advancement in software development. Whereas previously it was costly to replace faulty software, nowadays software is easily *updated* through the Internet, leading to reckless development strategies such as "move fast and break things." [219] In fact, many, often centralized, applications of today critically rely on the correct functioning of the Internet. However, this brings new risks too [150]: what if part of the Internet is shut down—either accidentally due to a misconfiguration [146], due to (geo)political pressures [221, 76], or maliciously due to a coordinated attack on critical routing infrastructure [126]? Can program verification play a role in the discovery of fundamental flaws in critical software that is essential to keep the Internet connected and its applications running?

- Complementary to increases in connectivity, we have also seen increases in usage and complexity of software in many different industries, known by the popular phrase "software is eating the world." [8] As such, the potential impact of programming errors is only increasing. A new dimension of interest is *cyber security* [197], and in particular finding (and sometimes correcting) so-called zero-day exploits [224]. An exploit allows malicious parties to severely disrupt software-dependent industries, such as finance, healthcare, and education [184], and well-known examples of abusing programming errors are ransomware attacks [25]. Especially when the development of software is *open-source*—allowing anyone to study the source code of software—there may be an increased risk that programming errors are known to attackers but not to the authors of the software [116], while for open-source software critical patches are released faster than for closed-source software [194]. Can program verification play a role in discovering these zero-day exploits?

- With the uptake of large language models for generating programs [229], the question of correctness may become more important than before: since no longer an educated, intelligent human—who is concerned about qualities such as correctness—is writing programs, but instead an artificial intelligent machine is writing it. This may lead to an increase of productivity in software development, but also to a decline of the quality of programs, in the coming decade. Can program verification play a role in the post-hoc verification of programs [102], e.g. generated by large language models [72]?

- As a consequence of the digital transformation, also critical governmental processes are increasingly handled by software, such as software that is used to count votes and compute results for elections,[1] and software used in processing taxes and detecting fraudulent citizens. It is of great public interest that all software involved in these critical processes behave correctly with respect to specifications that formalize the relevant laws. Since governments bear responsibility to the public, correctness of such software should be established beyond any rational doubt. Can program verification play a role in critical governmental software, such as election software or software that is used to process taxes?

Program verification can play a (modest) role in all of these issues, but its success may vary: although there are numerous success stories [101, 123], we shall discuss some technical challenges which have to be tackled before program verification can be actually applied in solving these issues, and discuss some limitations inherent to program verification in general [193, 174].

[1] See `https://nos.nl/artikel/2490218`

In this thesis, we study *program verification systems*, also called *theories of program correctness*: a systematic approach to study the correctness of programs. We distinguish the following components of a program verification system:

- the proper *programming language* is used for describing programs themselves,
- the proper *specification language* is used for specifying positive or negative properties of program behavior,
- the proper *deduction system* is used to verify whether a program is correct, by annotating programs with specifications.

Program verification amounts to showing that the correctness (with respect to specifications in the specification language) of a program (in the programming language) can be deduced (in the deduction system). For a program verification system to be adequate, it needs to have the following qualities:

- all the properties that are potentially observable in reality can be described as well (*expressiveness*),
- the properties that are deductively verifiable of a program also hold of the program when really executed (*soundness*),
- all the properties that are observed of all programs that are really executed can also be deductively verified (*completeness*).

To be able to assert that a program verification system has these qualities, one would need to investigate:

- the programming language and specification language have an unambiguous and formalized meaning (*semantics*),
- the verification system is based on an accurate theory of program behavior as would be observed when executing programs in the real world (*modeling*),
- the verification system itself contains no error (*consistency*).

Furthermore, for a program verification system to be usable, it needs to have the following qualities:

- many important classes of properties can be described and reasoned about in a short and straightforward way (*complexity*),
- a largest as possible part of the verification process can be performed automatically (*effectiveness*).

Recalling E.P. Wigner, that every theory is wrong in some way: in fact, no program verification system exists that satisfies all these qualities perfectly. V.R. Pratt investigated the semantics underlying program verification systems [179]. Already the choice of programming language and specification language have significant influence on the success of a program verification system. A too simplistic specification language may not be expressive enough to state the properties of a program, as investigated by S. Kamin [133]. A programming language may also be too powerful and have a combination of features that leads to incomplete program verification systems, as investigated by E.M. Clarke, Jr. [51, 52]. In fact, many program verification systems can not be fully automated [22, 53], or only automated in certain circumstances [99], but there are also program verification systems which can be fully automated [127]. Sometimes there are subtle interactions between different qualities, such as expressiveness and completeness, as investigated by J.A. Bergstra and J.V. Tucker [21], M.R. Artalejo [13], and others [149, 14]. Going further, a small mistake in modeling the behavior of programs could lead to a theory that makes predictions about program behavior that no longer has a useful relationship with the behavior of programs in the real world.

The task of designing a program verification system is thus difficult. We first discuss the design decisions taken in this thesis: we focus on pointer programs as programming language (Section 1.1), and focus on separation logic as specification language (Section 1.2). As soon as we have settled on these components, we discuss the deduction systems and the aforementioned qualities: expressiveness, soundness, completeness, and complexity (Section 1.2). However, there are significant gaps in the academic literature, and we argue why new foundations of separation logic are needed (Section 1.3). Finally, the scientific contribution, of filling in these gaps, and the larger scientific context of the work presented in this thesis is given (Section 1.4).

1.1 Pointer programs

One of the positive outcomes of the software crisis was the development of the so-called high-level programming languages. For example, in the early 1970s, the C programming language was designed [192] to be a high-level programming language, and it—together with the operating system Unix that was rewritten in it—became widely used in the decades that followed. Also in other operating systems, such as the kernel of Windows, Linux, and (forks of) BSD, the C language is (predominantly) used. Later programming languages, such as C++ [206] around 1980, Python [218] around 1990, and Java [175] around 1995, were even higher-level, by introducing features such as object-orientation and garbage collection, and also became well known and widely used—especially in application programming and on the Web. These four are the topmost widely-used programming languages [141], according to TIOBE Programming Community index in 2023.

To apply techniques of program verification to these practical languages, one has to design a program verification system that is tailored to each specific feature of each specific programming language. But, the syntax and semantics of each programming language differ significantly and require extensive modeling. In fact, even between different versions of the same language there could be many, possibly incompatible, differences in syntax and semantics. For example, the C++ language originally was an extension of the C language, but in decades time the two languages have significantly diverged in their semantics of those parts of the syntax that is still shared by both languages. As such, a verification system is tailored to a specific version of a programming language, and it is costly to keep up with the rapid onset of new versions [123].

On top of that, high-level programming languages have many diverse features:

- type systems with primitive types, reference types, subtypes, etc.
- object-oriented features such as (partial) encapsulation and reflection,
- static typing, dynamic typing, interfaces, dynamic dispatch, etc.
- complex rules for type coercions or conversions,
- expressions with unspecified evaluation order that can have side-effects,
- complex control flow and constructs for dealing with exceptions,
- manual, semi-automatic or garbage-collected memory management,
- concurrency features and multi-threading,
- raw or safe or lock-protected memory access,
- interaction with input/output devices and volatile memory,
- a large standard library.

These language features are not perfectly orthogonal, and may interact with each other in often subtle ways. Ideally, every widely-used programming language should also come with a fully-fledged program verification system, a so-called 'verification-oriented programming language' [157]. But this is not yet the case today: the widely-used programming languages lack such system. As such, pragmatically, in practical program verification systems for widely-used programming languages one restricts attention only to a subset of features, notably those features that are the cause of the most problematic programming errors [95, 2].

One such practical verification system is the KeY system that can be used to verify the correctness of Java programs [3]. The KeY system extensively models many features of the Java programming language, but it only supports single-threaded Java programs written in an older Java version that does not have generic types. Although not feature-complete, the verification system is still useful in practice, since, for example, it can successfully be used to discover bugs in Java's standard library: uncovering a crashing bug in the sorting algorithm [65, 66, 64], a twenty year old overflow bug in the linked list data structure [115, 114], and a bug in the BitSet class [208]. However, a known limitation of KeY is that, while in principle possible, practical reasoning about complex pointer structures is often difficult and time-consuming [113, 24, 207].

Such pointer structures are a common aspect of all of today's widely-used programming languages. That programs necessarily interact with working memory is a consequence of current computer architectures [200], in which the digital representation of values are stored at locations in random access memory (RAM). Pointer structures emerge as soon as the location (or address) of values are treated as values that can be stored in memory. The difficulty of reasoning about pointer structures comes from the fact that the same location in memory, where a value is stored, may have different names, i.e. can be identified by different expressions [32]. In that case, all the (different) names of the same location are called *aliases*. Every time the value of a location is updated, one would need to analyze every expression to see whether it refers to the updated location or not. This process is called *alias analysis*, which is the main source of difficulty in reasoning about the correctness in widely-used programming languages, such as Java.

As such, in this thesis we focus on the design of a system that is suitable for reasoning about pointer structures, but without covering all the specific features of each individual programming language or architecture. We investigate an *abstract* program verification system, that works well with *abstract* programs that manipulate pointer structures, in such a way to satisfy as many qualities of program verification systems as possible—expressivity, soundness, completeness, and complexity. The abstract system can be further tailored or integrated into practical verification systems for verifying properties of concrete programs. We refrain from doing the latter in this thesis, and shall not study the design or implementation of concrete program verification systems: rather, by focusing on the abstract, we offer a solid foundation for the future development of practical tools, thereby allowing those tools to benefit from satisfying the qualities, including completeness, too. For example, it is envisioned that the results of this thesis could serve as a foundation of the next version of the KeY system, KeY 3.0, to make reasoning about Java programs simpler.

Following along the tradition of Hoare, we focus on abstract programs. The abstract programs we study operate on (an abstraction of) random access memory in which locations are values too, which we call *pointer programs*. In such programs there are three different storage locations of values: the *registers*, the *stack*, and the *heap*. The value of every variable in a program can be thought of as if being stored in some register; one can introduce local variables and pass along values as parameters in *procedure calls* (also called function calls or method calls, depending on circumstances) by making use of the stack; and finally one can create, manipulate, and destroy so-called *objects* which are stored in another part of the memory called the heap.

For example, see the program in the C programming language in Figure 1.1. After compiling the program, and running it with the command line argument `5`, we can see the following happening:

(a) the procedure `main` is called, with $argc = 2$ and $argv$ pointing to some array of strings on the heap;

(b) the standard library function `atoi` is called, which converts the contents of the string pointed to by $argv + 1$ into an integer;

(c) we allocate a dynamically sized array of n integers with unknown initial values, and let x point to the start of that array;

```c
#include <stdio.h>
#include <stdlib.h>
int* x;                                       // global variable
int n;                                        // global variable
int main(int argc, char** argv) {             // formal parameters
  if (argc != 2) return 1;
  n = atoi(argv[1]);                          // assignment, procedure call
  x = (int*) malloc(n * sizeof(int));         // allocation
  if (x == NULL) return 1;
  // Create table
  for (int i = 0; i < n; i++) {               // local variable (i)
    int j = n - i;                            // local variable (j)
    *(x + i) = (i * j) + (j - i);             // mutation
  }
  // Find the largest number
  for (int i = 1; i < n; i++)                 // local variable (i)
    if (*x < *(x + i))                        // lookups
      *x = *(x + i);                          // lookup, mutation
  // Print output
  printf("%d\n", *x);                         // lookup, procedure call
  free(x);                                    // deallocation
  return 0;
}
```

(a)

Location	$argv + 0$	$argv + 1$
Value	–	z

(b)

Location	$z + 0$	$z + 1$
Value	`'5'`	`'\0'`

(c)

Location	$x + 0$	$x + 1$	$x + 2$	$x + 3$	$x + 4$
Value	–	–	–	–	–

(d)

Location	$x + 0$	$x + 1$	$x + 2$	$x + 3$	$x + 4$
Value	5	7	7	5	1

(e)

Location	$x + 0$	$x + 1$	$x + 2$	$x + 3$	$x + 4$
Value	7	7	7	5	1

Figure 1.1: A C program that computes a table, finds the maximum, and prints the result.

(d) we initialize the values of the array that x points to;

(e) we search the array from the left to right, eventually storing the largest number at the beginning of the array.

This example illustrates the following programming concepts that we investigate:

- *local variables* and *global variables*,
- basic sequential programming structures such as loops,
- *procedure calls* with call-by-value parameters,
- dynamic *allocation* of memory,
- pointer dereferencing—also known as *lookup*,
- assignment through pointer dereferencing—also known as *mutation*,
- *deallocation* of dynamically allocated memory.

However, the example also illustrates many features from which we abstract: complex expressions, types and data structures, value representation and memory layout, standard libraries, procedures with return values, non-local control flow, input/output, et cetera. We abstract from these features to instead focus on the foundational issues.

One could argue that it is not necessary to study abstract pointer programs, since already the abstract programming language of simple **while**-programs operating on integers, as studied by Hoare, is Turing-complete. Hence every pointer program can be turned into such a simpler **while**-program that does not use pointers at all, and then one could use the existing program verification system introduced by Hoare, to indirectly reason about the correctness of (translated) pointer programs. However, such approach is not natural, since the formulation of correctness specifications then depends on the chosen translation (of which there are many variations). Instead, we want to keep close to a natural programming model, that is close to how one would informally reason about pointer programs.

Furthermore, we focus in the main matter of this thesis on the primitive operations of pointer programs. The control structures such as conditional branching, looping, and recursive procedures are orthogonal to our concern: these complex control structures are discussed in the appendix.

1.2 Why separation logic?

The next important design choice is what specification language to employ. The quest for finding a suitable specification language for pointer programs is long. Traditionally, Hoare used so-called *first-order logic* as a specification language for program verification. Early on, in the 1970s, the problem of finding a suitable specification language for describing the behavior of pointer programs was explored, for different classes of data structures, by R.M. Burstall [42], T. Kowaltowski [138], M.S. Laventhal [142], and others. The problem with these approaches, however, was that they were restricted to classes of data structures that were all tree-like. The main difficulty of giving specifications to pointer programs is, however, in dealing with the cyclic nature of the data structures stored on the heap.

In 1975, two papers appeared by S.A. Cook and D.C. Oppen [56, 169]. In these papers, the matter of proving correctness of pointer programs was settled—and for all data structures in full generality, including cyclic data structures: first-order logic was chosen as a specification language, in the tradition of Hoare, and a soundness and completeness proof was given.[1] Unfortunately, these papers did not become widely known. In fact, later on, it was believed that first-order logic is not suitable as a specification language for pointer programs, since the axioms of Cook and Oppen were deemed 'extremely complicated' [187]. Although it is undeniable that Cook and Oppen have demonstrated completeness, hence that this 'extreme' complexity is *necessary*, what remains is the strive for the best of both worlds: completeness *and* simplicity.

Next to the approach by Cook and Oppen, there is an alternative approach for reasoning about pointer programs. Essentially, one could treat the heap as if it were one large array. One can look up the value stored at a location by simply treating the location as an index into the array representing the heap. Allocation then amounts to searching that array for a free location, and mutation amounts to assigning a value to a particular index in the array. A special marker value is needed to indicate the absence of a value, representing a free location on the heap: deallocation then simply assigns that location in the array this special value. To reason about allocations, mutations, and deallocation, one is required to perform an alias analysis on *every* reference to the heap in a specification. This approach to axiomatization, in which one uses an assignment axiom for arrays that deals with complex index expressions, was first described [11] in detail in 1980 in the book *Mathematical Theory of Program Correctness* by J.W. de Bakker [61] (see also the work by J.M. Morris [158]). Although De Bakker did not explicitly mention pointer programs, he did mention in the first paragraph of Chapter 4, that introduces the assignment axiom for arrays, that it 'constitutes a very modest venture into the realm of data structures.' Maybe he was too modest, since—similar to Cook and Oppen's approach—also De Bakker's approach is sound and complete.

This approach, reasoning about aliasing as in the case of updating an array, is also taken in proof systems for reasoning about the correctness of object-oriented programming languages, as was first done in the papers by P.H.M. America and F.S. de Boer, collected in the 1991 Ph.D. thesis of De Boer [63], and later refined by C. Pierik and De Boer [176]. In object-oriented languages one abstracts from locations of objects as manipulable addresses, and instead treats objects as abstract identities [43, 155, 173]. By doing so, object-oriented languages can be equipped with a garbage collector, removing the need for programmers to manually deallocate unreachable memory. De Boer et al. axiomatized the operation of object allocation

[1]The completeness result was relative to an expressivity condition, and later this became known as 'relative completeness' in the sense of Cook [55].

by introducing a substitution-like operator, analyzing formulas by their logical structure and performing alias analysis in a special sense—by answering the question: does a term potentially refer to a newly created object or not [4]? Since their work focused on garbage-collected object-oriented languages, this line of research lacked axiomatization of the operation of deallocation.

These approaches, as described by Cook and Oppen, De Bakker, and De Boer et al., are similar by recognizing they all perform *explicit alias analysis.* In the axiomatization of operations such as allocation, mutation, or deallocation, one analyzes every reference to the heap and decide whether it is affected by the operation or not. Whether aliasing occurs or not is made explicit in the axiomatization by logically making a case distinction for each reference to the heap that potentially is an alias with the location affected by the operation. Explicit alias analysis may complicate practical reasoning for two main reasons:

- it is difficult to modularize reasoning about fragments of the heap—that is, it is difficult to *adapt* specifications that specify 'local' properties of the heap into specifications that specify 'global' properties of the heap;
- alias analysis is required for *every* reference to the heap—and since the alias analysis results in a case distinction (alias or not), reasoning about non-trivial specifications quickly becomes complex.

In practice, explicit alias analysis is also performed by the KeY system [3]. In KeY, program specifications describe properties of the entire 'global' heap. Although KeY employs techniques that allow for modular reasoning [196], by declaring what locations of the heap can be *changed* and *accessed*, this still yields complex proof obligations and difficult to automate proofs [113, 207].[1]

With explicit alias analysis, complexity arises from reducing the fact that an alias occurs or not to an equational property, that is, whether two expressions refer to the same location or not. There is also a different approach, which one could call *implicit alias analysis.* With implicit alias analysis, one avoids such reduction to equational properties, and thereby avoid the need to perform many case distinctions. The approach of implicit alias analysis can be thought of as more 'topological' in nature, in the sense that it is possible to guarantee two expressions are not referring to the same location by their spatial properties. Two expressions are equal, and refer to the same location, if they have every property in common (sometimes known as Leibniz' law). With implicit alias analysis, one guarantees the absence of aliasing by declaring two expressions denote a location that is necessarily separate in space, and thus have not every property in common.

Around the turn of the millennium, J.C. Reynolds wrote the article *Intuitionistic Reasoning about Shared Mutable Data Structure* [187]. Herein, Reynolds returns to the original idea set out by Burstall [42]: the specification language should not only describe the state of the memory as one could do in first-order logic, but also capture certain spatial aspects of the *layout* of the memory. He writes:

[1]This difficulty, encountered during practical verification efforts of the Java collection framework, was one of the motivations for starting the research described in this thesis.

> "Burstall's 'distinct non-repeating tree system' was a sequence of assertions, written $\phi_1 * \ldots * \phi_n$ [in the notation of this thesis], where each [component] ϕ_i described a distinct region of storage, so that an assignment to a single location could change only one of the ϕ_i. I believe that this idea of organizing assertions to localize the effect of a mutation may be the key to scalability in reasoning about shared mutable data structure." [187]

Reynolds significantly improves the approach by Burstall, as described in several papers [187, 188, 189], by allowing for sharing of substructures from within different components, and by allowing pointers both to and from different components. Together with P.W. O'Hearn, S.S. Ishtiaq [125], and H. Yang [230], Reynolds thus introduced what later became known as *separation logic* (see also [168, 67]). Note that, already from the start of the investigation into, and later development of, separation logic, there has been a focus on the scalability of the approach, and practical usability of the specification language. In fact, in the paper *Why separation logic works*, D. Pym, J.M. Spring, and O'Hearn [180] argue that

> "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 has a proof theory which aids in the crucial problem of scaling the reasoning task. Scalability is a central problem, and some would even say the central problem, in applications of logic in computer science."

Separation logic became hugely successful and influential. Over the past twenty years, separation logic developed into a serious academic research subject [125, 89, 215, 140, 73, 161, 165], and has numerous applications in practical program verification [139, 132]. In 2016, the European Association for Theoretical Computer Science (EATCS) awarded the Gödel prize to S. Brookes and O'Hearn for introducing concurrent separation logic [36, 163], an extension of Reynolds' program logic to reason about concurrent pointer programs. Also in practice, separation logic is successful, since it forms the basis for practical program verification systems [131, 228] for proving correctness of modern programs, e.g. written in Mozilla's Rust and Google's Go.

Separation logic is the specification language that we settle on in this thesis: we study pointer programs and describe program behavior using the language of separation logic. Separation logic has many benefits: since the heap is not represented by any variable, the language offers modular descriptions of fragments of the heap—and as such allows easy adaptation of 'local' specifications into 'global' specifications. Due to the main idea of Burstall, improved by Reynolds, it is not always the case that alias analysis is necessary for *every* reference to the heap in specifications of pointer programs, but alias analysis is restricted to only those components that are actually affected by memory manipulating operations: this is the essence of the scalability argument of separation logic.

1. Assertion language	First-order logic	Separation logic†
2. Program logic	Hoare's logic	*Reynolds' logic*†
3. WP-calculus	Dynamic logic	Dynamic separation logic††

Table 1.1: Terminology as used in this thesis. The first column shows different levels of the logics studied. The logics marked by a dagger (†) are prime subjects of this thesis, where novel contributions are made. The logic marked by a double dagger (††) is entirely novel.

In becoming such a mature subject of study, it is also increasingly more important that its foundations are properly understood. However, as often happens in periods of enthusiastic scientific advancement, and as the title of the paper *Bringing order to the separation logic jungle* by Q. Cao, S. Cuellar, and A.W. Appel [46] may suggest, no longer it is clear what one means by 'separation logic': some authors use the it to mean (a variant of) the assertion language, being an extension of first-order logic; whereas other authors use it to mean (a variant of) the program logic, being an extension of Hoare's logic.

In fact, the 2002 paper by Reynolds [188] introduced two systems: an assertion language and a program logic. To avoid confusion, we shall call the assertion language 'separation logic', and we shall coin the name 'Reynolds' logic' to refer to the program logic (and *not* the assertion language).[1] From this point onward, we mean by 'separation logic' only the assertion language that was introduced by Reynolds in 2002 [188]. By introducing this terminology, it is easier to see the difference between 'separation logic' and 'Reynolds' logic', and see how they are related to 'first-order logic' and 'Hoare's logic', respectively. Note that by giving these names to the logical systems we study it remains important to remember that it was not only Hoare or Reynolds, but many whom have contributed to the formulation, semantics, and axiomatization of these program logics.

We recognize three different levels: that of assertion languages, that of program logics, and that of weakest precondition calculi. On the first level, separation logic is used as an assertion language, similar to how first-order logic is used as an assertion language. On the second level we have Hoare's logic, the deductive system introduced by Hoare in 1969 [118, 11]. On the same level, by Reynolds' logic we mean the extension of Hoare's logic to incorporate separation logic in specifications of programs to reason about pointer programs, introduced by Reynolds in 2002 [188]. On the third level, we investigate dynamic separation logic—an extension of dynamic logic. First-order dynamic logic was introduced by D. Harel in 1979 [107]. Dynamic separation logic is novel and not studied before[2]. By dynamic separation logic we mean first-order dynamic separation logic: in 2020, propositional dynamic separation logic was introduced by Maratovich [147].

An overview of the new terminology is shown in Table 1.1. The relation between the different logics is the following:

[1]The only previous occurrence of the name 'Reynolds' logic' was found in an unpublished note by David A. Naumann.

[2]Credit is due to Einar Broch Johnsen for suggesting the name 'dynamic separation logic'.

1. The level of the *assertion language*: we consider the formulas of *first-order logic* (with a built-in equality predicate), and the formulas of *separation logic*. Separation logic is an extension of first-order logic, in the sense that every formula of first-order logic is also a formula of separation logic. Separation logic adds two new connectives, the separating conjunction $*$ and the separating implication $-\!\!*$, and a built-in predicate, the 'points to' predicate $\hookrightarrow$, to the syntax of first-order logic. In the semantics of these assertion languages, we focus on classical first-order logic, and classical separation logic [188].

2. The level of the *program logic*: in *Hoare's logic* one reasons about Hoare triples $\{\phi\}\ S\ \{\psi\}$, where the *precondition* ϕ and *postcondition* ψ are first-order formulas (of the level above) and S a program in a simple **while**-language. A Hoare triple is a specification of the program S, where the postcondition ψ describes the expected final state after running the program S from an initial state satisfying the precondition ϕ. We focus on partial correctness semantics of Hoare's logic.

   In *Reynolds' logic* one reasons also about triples $\{\phi\}\ S\ \{\psi\}$, but the assertions are now formulas of separation logic (of the level above) and S is a pointer program (an extension of the **while**-language with heap memory manipulating operations). Almost all axioms and proof rules of Hoare's logic are reusable in Reynolds' logic, except for the *invariance rule* (introduced in the introduction of Chapter 4). Reynolds' logic further includes the so-called *frame rule* and axioms for the primitive operations of pointer programs. We focus on the strong partial correctness semantics of Reynolds' logic.

3. The level of the *WP-calculus*, or *weakest precondition calculus*: we consider the formulas of *dynamic logic*, and the formulas of *dynamic separation logic*. Dynamic logic is an extension of first-order logic by introducing the weakest precondition $[S]\psi$ for every given program S and postcondition ψ. The Hoare triples $\{\phi\}\ S\ \{\psi\}$ of Hoare's logic (of the level above) are embedded in dynamic logic as the implication $\phi \to [S]\psi$: the given implication is valid in dynamic logic if and only if the Hoare triple is valid on the level above.

   Our novel approach is, at this level, to introduce *dynamic separation logic* as an extension of dynamic logic which includes the connectives of separation logic, in such a way that the triples $\{\phi\}\ S\ \{\psi\}$ of Reynolds' logic (of the level above) can be embedded into dynamic separation logic as the implication $\phi \to [S]\psi$, see also Section 4.4. Note that the difference between dynamic logic and dynamic separation logic is that, in the latter, we can use the separating connectives and the built-in 'points to' predicate, in a similar way how separation logic extends first-order logic at the first level. By introducing dynamic separation logic, an alternative axiomatization of Reynolds' logic (of the level above) was discovered (see Section 4.5).

Coming back to the article by Halpern, Harper, among others: that logic is unusually effective in computer science can again be witnessed from Table 1.1. The systems that we study in this thesis all end with 'logic'!

1.3 Why new foundations?

Separation logic is an extension of first-order logic, in the sense that it adds two new connectives: the separating conjunction $*$, and the separating implication $-\!\!*$. The latter is also called the *magic wand*. Already the separating implication is sufficient, in the sense that separation logic without separating conjunction but with separating implication is equally expressive to separation logic with both connectives [35]. However, reasoning about separating implication is complex: separation logic is equally expressive as weak second-order logic, and therefore is undecidable [35]. In practice, tools for automatic reasoning about separation logic either are restricted to the fragment of the language without separating implication or require so-called packing operations to direct the proof search [59].

The research of this thesis started with the discovery of an alternative axiomatization of Reynolds' logic, the program logic used for reasoning about pointer programs. By taking a different approach than what was done previously, a new axiomatization of *exactly* the same theory of pointer program correctness was discovered (see Chapter 4). However, this alternative axiomatization gave rise to a remarkable question: we generate two formulas in separation logic that are necessarily equivalent, since both are the weakest precondition with respect to a given program and postcondition. Symbolically, we have (see also Section 2.1):

$$(x \mapsto -) * ((x \mapsto 0) -\!\!* (y \hookrightarrow z)) \tag{1.1}$$

$$\equiv$$

$$[[x] := 0](y \hookrightarrow z) \tag{1.2}$$

$$\equiv$$

$$(x \hookrightarrow -) \wedge ((y = x \wedge z = 0) \vee (y \neq x \wedge y \hookrightarrow z)) \tag{1.3}$$

where (1.2) is the weakest precondition expressed in dynamic separation logic with respect to the program $[x] := 0$, that assigns the value 0 to location x, and the postcondition $(y \hookrightarrow z)$, that expresses that location y has value z. The approach of Reynolds to generate the weakest precondition leads to the formula (1.1), in which we can see the two separating connectives introduced and two subformulas. The subformula $(x \mapsto -)$, respectively $(x \mapsto 0)$, expresses that strictly the location x has some value, respectively the value 0. The weakest precondition (1.1) follows the *implicit* alias analysis approach. In contrast, our novel approach results in generating the weakest precondition (1.3), following the *explicit* alias analysis approach. Are we now also able to *show* the equivalence of (1.1) and (1.3) using the existing techniques for reasoning about separation logic?

In particular, this question involves establishing sufficient facts that speak about the so-called 'points to' predicates. There is the *weak* (or *loose*) 'points to' predicate $\hookrightarrow$, and the *strict* 'points to' predicate $\mapsto$. In his seminal paper, Reynolds describes a set of necessary truths that hold for these 'points to' predicates. However, Reynolds also writes (emphasis not originally present):

> "Finally, we give axiom schemata for the ['points to'] predicate $\mapsto$. **(Regrettably, these are far from complete.)**" [188]

Nonetheless, this never was a problem for the success of separation logic. Was it simply never the case anyone needed more than the axioms that were given by Reynolds? Or, may our question reveal there is a missing piece?

Part of the problem may have come from the fact that 'separation logic' was ambiguous. As alluded to previously, we distinguish 'separation logic' from 'Reynolds' logic'. Surely, Reynolds' logic is complete in a special sense: it was established multiple times in different settings that the program logic is sound and relatively complete. For example, in the work by M.F. Al Ameen, W.-N. Chin, M. Tatsuta [81, 209], the authors show relative completeness of Reynolds' logic based on a weakest precondition axiomatization, a result that is on the same level as the well-known result by Cook that proves the relative completeness of Hoare's logic [55] that in part appeared earlier in the Ph.D. thesis of Clarke Jr. [50] and the M.Sc. thesis of G.A. Gorelick [96]. Also there is a strongest postcondition axiomatization of Reynolds' logic, by C. Bannister, P. Höfner, and G. Klein [15]. However, in the work by Al Ameen et al. that gives a weakest precondition axiomatization, the frame rule is not needed to obtain relative completeness: for the primitive operations, a direct weakest precondition can be given, and in the case of recursive procedures one could employ an encoding of the heap to obtain most general specifications.

However, the frame rule is a crucial aspect of Reynolds' logic. That it is possible to reason locally about the correctness of pointer programs was investigated in detail in the Ph.D. thesis of Yang, where he shows that the frame rule can be used to obtain modular relative completeness [235, 230]. Furthermore, different axioms for the primitive operations of pointer programs can be given and these axioms are interderivable by use of the frame rule (see also [45, 190, 58, 164]). In the case of the mutation operation that modifies the heap, the frame rule is instantiated with a formula involving the magic wand. In practice, many tools for automatic reasoning in separation logic are restricted to the $-\!\!*$-free fragment of the language, due to the complexity of reasoning about the magic wand [148, 75]. Since using the magic wand is complex to reason about, we thus wish to obtain a relative completeness result without using the magic wand connective [145, 19]. This question shows an interesting connection between complexity (avoiding the magic wand) and relative completeness of Reynolds' logic.

It turns out that our novel approach leading to an alternative axiomatization also solves this open problem: we can show that the local axioms are relatively complete by instantiating the frame rule *without* introducing the magic wand (see Section 4.5). This means that any valid Hoare triple of primitive pointer programs, which are specified without using the magic wand, can also be proven correct without using the magic wand in the frame rule.

All these relative completeness results of Reynolds' logic work on top of the assumption that there is an oracle which provides the valid formulas of separation logic. This assumption is similar to the one made in the relative completeness result of Hoare's logic: the question of program correctness is reduced to questions of validity in the underlying logic [149]. This process, of reducing the question of program correctness to questions of logical validity, is called *verification condition* *generation* [151]. In the case of Hoare's logic, the underlying logic is first-order logic. Since there are proof systems for first-order logic (e.g. Hilbert systems, natural deduction, sequent calculus) which are sound and complete, it is actually possible to prove the true verification conditions, and thus to prove that correct programs are indeed correct.

In the case of Reynolds' logic, the underlying logic is separation logic. However, what about the question whether separation logic—the logic used to reason about the assertion language—is complete, similar to how Gödel proved completeness of first-order logic in his Ph.D. thesis [136, 34]? Again there may be some ambiguity involved: it is easy to make the mistake to think that first-order logic must be incomplete, since Gödel proved the famous incompleteness theorems [202, 182]. However, the incompleteness theorems state something different than the negation of Gödel's completeness theorem.

Ironically, the 2016 Gödel prize winners, Brookes and O'Hearn, wrote [38]:

> "It is all too easy to get caught up in completeness and related issues for formal systems that turn out to be too complicated when humans try to apply them; it is more important first to get a sense for the extent to which simple reasoning is or is not supported."

This thesis is summarized as such: separation logic has passed the phase in which 'a sense for the extent to which simple reasoning is supported' is obtained, and now it is time 'to get caught up in completeness and related issues'. Although there were earlier attempts to give a completeness result for separation logic, either by restricting to a fragment of the language [20], by abstracting away from the 'points to' predicate [121], or by looking at a restricted form of completeness called weak completeness in which one reasons only about universal validity [143]:[1] the completeness of separation logic has not yet been established. Soundness and completeness of a logic (also called its *adequacy*) is an important matter, as can be illustrated by considering their practical applications.

Practical tools for reasoning about separation logic can be grouped in two fundamentally different approaches: satisfiability checking and theorem proving. This follows the same two approaches in first-order logic. In the case of first-order logic, these different approaches are possible due the adequacy of first-order logic: the set-theoretic semantics underlying first-order logic is sound and complete with respect to its syntactic proof system, as was proven by Gödel in his completeness theorem. Due to the adequacy of first-order logic, we can *try* to answer the question whether a formula ϕ is a (syntactic or semantic) consequence of a theory Γ in two essentially different ways: either the semantic way, by showing a counter-model that satisfies the theory Γ but not ϕ, or the syntactic way, by showing there is a proof with premises in Γ and conclusion ϕ. However, this analysis requires human ingenuity: the question whether a formula follows from a given theory in general is undecidable, as was established by Church and Turing [23], both inspired by Gödel's incompleteness theorems. Despite this undecidability, in the years that

[1]The authors of [143] acknowledged on their website that one of their proof rules is unsound for the standard semantics, and removing that rule yields an incomplete proof system.

followed, a rich model theory and proof theory developed for first-order logic, leading to a great many techniques on either side, and often transporting results from one side to the other side due to completeness.

However, in the case of separation logic, there is no analogue to Gödel's completeness theorem. The goal of this thesis, and leading to new foundations for separation logic, is to be able to provide such an analog to the completeness theorem, or—at least—make the path towards it clear. However, why are new foundations needed? To motivate our goal, we revisit the equivalence of the generated weakest preconditions (1.1) and (1.3) discovered earlier.

In satisfiability checking, one is interested in automatically checking the satisfiability of separation logic sentences. To do so, it is useful to consider a semantics that is based on finite or finitary structures that can be enumerated. Many fragments of separation logic were isolated, decision procedures for some of them were developed and compared in benchmarks, and undecidability results for other fragments were proven [40]. There are fragments called propositional separation logic [44], set separation logic [110], separation logic of linked lists [177], and there are different subsets of the language which restricts the use of certain connectives [69]. However, none of the current satisfiability checking tools of separation logic are able to show our equivalence of (1.1) and (1.3), either because the equivalence falls outside the supported fragment or due to a failure to produce an output. This shows that the tools are incomplete, but this is not due to deep results such as Gödel's incompleteness as one might expect, but instead due to an inadequate semantics.

On the other hand, in interactive theorem proving, one is interested in constructing finitary proofs (also called certificates) that witness the validity of separation logic sentences. To do so, logical frameworks or embeddings in type theories can be employed, as they help with the construction of proofs. However, by focusing on the proof system only, we suffer from inadequacy of the logic due to an underdeveloped model theory: if one fails to prove something, one can always be blamed for not looking far enough, since there are no semantic means by which one can convincingly show that there is no proof to be found in the first place. One may argue that, due to the rich structures in which one is reasoning, we are already incomplete (in Gödel's incompleteness sense). But does that fact alone justify a lack of interest in the adequacy of the logic of separation logic itself?

Summarizing the state-of-the-art that is based on an inadequate semantics: a tool based on satisfiability checking can be used to find counter-models, but can never be used to argue that a formula is valid from *the lack of finding any counter-models*. A tool based on (interactive) theorem proving can be used to find proofs of validity, but can never be used to argue that a formula is invalid from *the lack of finding a proof*. We need an adequate logic to connect the two approaches!

The lack of an adequate semantics in current practice leads to workarounds. For example, one would need to introduce ad-hoc theories, e.g. a theory per data structure such as linked lists [48, 211] or trees [178, 166]. Such theories are typically infinite and defined inductively. But, due to the lack of an adequate semantics, it is not clear either what are, or are not, the consequences of each different theory.

New foundations are needed, because in the case of first-order separation logic we already suffer from inadequacy due to non-compactness of the standard interpretation: there is no suitable finitary proof theory in which all valid formulas can be derived. This should not be surprising, since the same also holds for first-order logic in which the semantics is restricted to finite structures—also leading to non-compactness: there is no finitary proof theory in which the valid formulas of first-order logic with respect to finite structures can be derived. Furthermore, an adequate semantics for separation logic may be the first step towards a model theory that helps answering meta-theoretical questions such as consistency and independence of axioms.

Thus, to attain our goal of completeness for separation logic, we design a model theory (Chapter 2) and proof theory (Chapter 3) that is adequate for separation logic. Surely, we do not strive for decidability: that is an unattainable goal for the same reasons as for first-order logic. However, from a methodological point of view, an adequate semantics gives again two essentially different ways to establish whether a formula of separation logic ϕ is a (syntactic or semantic) consequence of a theory of separation logic formulas Γ or not: the syntactic way of showing a proof, or the semantic way of showing a counter-model. We furthermore desire that the proofs in our proof theory are *finitary*, for the following reason: we need an effective procedure for deciding whether some object is acceptable as proof or not. The decidability and complexity of the proof checking procedure is important, since otherwise an unfair and high amount of effort is required of the proof checker. In proof theories where the proof checking procedure is undecidable, or is unreasonably complex, it becomes possible to give a *proof by intimidation* where all the resources of the proof checker are exhausted while no new knowledge is gained (see also [1, 223]).

Such new foundations requires revision of the basic assumptions underlying the existing standard interpretation of separation logic, but it also requires the design of a new proof system. The first crucial point in revising the basic assumptions is to drop a *finiteness condition* on the heaps with respect to which separation logic formulas are evaluated, thereby generalizing the interpretation of separation logic and include the possibility of *infinite heaps*. The second crucial point is restraining the expressive power of higher-orderedness: in this thesis it is shown that the (non-standard) interpretation of separation logic with respect to *all* (finite or infinite) heaps can be considered as an intermediate logic between first-order logic and second-order logic, and might even be equivalent to second-order logic. However, this is also not a suitable interpretation, so we need to follow the footsteps of Henkin [108, 109] to obtain a suitable interpretation for which a soundness and completeness result can be obtained, and restrict ourselves to particular sets of heaps: those sets of heaps which includes the first-order definable heaps.

Within the separation logic community there seems to be a wide-spread belief that finiteness is fundamental assumption. Many widely-cited papers on separation logic work with finitely-based heaps, see e.g. [188, 20, 74, 27, 39, 90, 68, 185, 77, 78], although there are also some authors who drop the finiteness condition [222, 121]. In fact, after submission of a paper of some of the results presented in this thesis, with the finiteness condition dropped, some of the anonymous reviewers remarked:

> "By extending the semantics to infinite and first-order definable heaps, sure, we obtain a sound and complete axiomatization. However, what is this useful for? Programs most definitely operate only on finite heaps; so how useful (sound?) is it to use the proof system obtained from an extension of the semantics of separation logic to infinite heaps?"

and

> "For me, the finiteness of the heap is one of the fundamental decisions about separation logic. Of course, it is very natural to try to see what happens if some assumption is weakened or removed. Sometimes one finds something very interesting, sometimes less so. [...] My main objection is about motivation. It is not clear to me why these variations on separation logic are interesting. It is of course good to explore all variants of a standard definition. This paper does it well, but the results it obtains are maybe not important enough for it to be accepted at [conference]."

From a practical standpoint, one may object to the generalization to infinite heaps by arguing that infinite heaps do not exist in practice ("programs most definitely operate only on finite heaps"). From a theoretical standpoint, one may object to the generalization to infinite heaps by arguing that the class of valid formulas changes accordingly. Against this and similar objections one can put forward a philosophical argument, a mathematical argument, a semantical argument, a correctness argument, a computational argument, and a pragmatical argument—all in *favor* of allowing infinite heaps.

The philosophical argument. The concepts of 'finite' and 'infinite' belongs to mathematics and not to logic, since finiteness is a predicate that speaks about the cardinality of a set and thus requires, in the background, knowledge of sets (e.g. as axiomatized by classical Zermelo-Fraenkel set theory). It seems good philosophical practice to eliminate as many assumptions, or ontological commitments, as possible from a logic.

The mathematical argument. The first-order theory of real closed fields in particular has the real numbers as a model, and is an elegant theory with nice meta-theoretical properties such as the decidability of its first-order properties. The objects of this theory are infinitary too, such as the real algebraic number $\sqrt{2}$ when viewed as an infinite decimal expansion. Going further, it is known that, for example, E.W. Dijkstra did not limit himself to integer programs, but also proved correctness of programs that operate on (mathematical) real numbers.[1] To the ultrafinitist it may seem defensible to say that such 'real numbers', viewed as infinitary mathematical objects, do not 'really' exists. However, real numbers are a *useful fiction*: there are many

[1]See `https://www.youtube.com/watch?v=GX3URhx6i2E`

benefits from using real numbers in applications such as analysis, probability, physics, et cetera, and this wide applicability follows from its well-understood theory. Similarly, by considering the possibility of infinite heaps, as a useful fiction, we shall see in this thesis that we obtain an elegant meta-theory, which may lead to practical benefits too.

The semantical argument. In the semantics of programs we also deal with potentially infinite sequences of successive configurations. In fact, for non-terminating executions, we have an infinite sequence of configurations. It seems unfair to, one the one hand, allow this possibility of infinity in the semantics of programs, but, on the other hand, deny the possibility of infinite heaps in the semantics of separation logic. In fact, in practice, non-terminating programs are useful too: many reactive systems are specified by non-terminating programs that react to input events by generating output events.

The correctness argument. It turns out that we are able to show that the proof rules and axioms of Reynolds' logic are all sound (and relatively complete): both in the standard interpretation where we restrict to finite heaps, but also the full interpretation that is based on all and potentially infinite heaps, or any intermediary interpretation that fixes a set of heaps that satisfy modest closure conditions. It is quite remarkable that the program logic remains sound and relatively complete, even when the interpretation of the assertion language can be changed *ad libitum*.

The computational argument. Infinite heaps can represent potentially infinite data, such as input/output streams. One could realize potentially infinite data by a *lazy* computation strategy, in which the value of a location of an infinite heap is computed on-the-fly. Such potentially infinite data structures could be specified co-inductively. Alternatively, potentially infinite data could result from interaction with an external environment, e.g. in computer networks. Hence it is not the case that "programs most definitely operate only on finite heaps".

The pragmatical argument. Even when one restricts to heaps with a finite domain, there remains the difference between heaps which have a bound on the size of their domain or whether the (finite) domain is unbounded. In practice, and especially in non-terminating programs, one has to work with a bound on the maximum available free locations on the heap, e.g. there is only finitely much memory available. Such heaps can be modeled by an infinite, co-finite heap, in which there are only finitely many locations not allocated.

However, as soon as one is committed to the possibility of infinite heaps, one should not overshoot and *fully* embrace infinite heaps. The full interpretation of separation logic, in which we consider *all* (finite or infinite) heaps is just as non-compact as the standard interpretation, and thus is not suitable for attaining our goal of an adequate logic.

The second crucial point is that we introduce an interpretation of separation logic akin to Henkin's general interpretation of higher-order logic, in which the formulas of separation logic are interpreted with respect to a given set of (finite or infinite) heaps. We further restrict ourselves to general interpretations in which the set of heaps satisfy a so-called semantic comprehension condition. One model of that interpretation consists precisely of those heaps that are definable by a formula (which themselves are recursively enumerable). This latter model is central in the completeness proof (of separation logic). We also introduce another class of general interpretations, in which the set of heaps satisfy a number of closure properties: these sets of heaps are called memory models. Memory models are central in the relative completeness proof (of Reynolds' logic).

1.4 Scientific contributions

In this thesis, we focus sharply on the subject: classical separation logic. The thesis consists of two parts: in the first part, we study the logic of separation logic, and in the second part we study Reynolds' logic.

The results of this thesis are primarily based on the following two publications:

- *The logic of separation logic: models and proofs*
  Frank S. de Boer, Hans-Dieter A. Hiep, Stijn de Gouw
  In: *Automated Reasoning with Analytic Tableaux and Related Methods: 32nd International Conference, TABLEAUX, Proceedings*
  Lecture Notes in Computer Science, volume 14278
  Springer, 2023

- *Dynamic separation logic*
  Frank S. de Boer, Hans-Dieter A. Hiep, Stijn de Gouw
  In: *Proceedings of MFPS XXXIX*
  Electronic Notes in Theoretical Informatics and Computer Science, volume 3
  Episciences, 2023

These publications also correspond to the two parts of this thesis. The first part comprises a model theoretic and proof theoretic investigation of classical separation logic—the logic. The second part comprises a novel interpretation of Reynolds' logic, the introduction of dynamic separation logic, and an alternative weakest precondition and strongest postcondition axiomatization.

In Chapter 2 (of the first part) we show the inadequacy of the standard of separation logic, by showing it is non-compact. We then introduce a new interpretation, the full interpretation of separation logic based on the possibility of infinite heaps, and show it is inadequate too. We investigate the sufficient and necessary conditions for an embedding of the standard interpretation into the full interpretation, and we introduce relational separation logic to compare separation logic to second-order logic. Interestingly, the full interpretation of separation logic is close to the standard interpretation of second-order logic, and we see that expressivity of a binding operator is sufficient for the two logics to coincide.

In Chapter 3 (of the first part) we introduce a proof theory with respect to a novel interpretation of separation logic that is based on first-order definable heaps. As such, the resulting proof system and interpretation are shown to be adequate. The proof system is presented as a sequent calculus, but also a second proof system is introduced in the style of natural deduction. The sequent calculus is shown to be sound and complete with respect to first-order definable heaps, and the natural deduction calculus is shown to be sound and its completeness is with respect to structures satisfying a semantic comprehension condition. The latter proof system operates on more general formulas than those of separation logic, by introducing a connective that is closely related to the binding operator of the previous chapter.

The approach of Chapter 4 (of the second part) first introduces general interpretations of separation logic, and a class of structures based on so-called memory models, which are necessary for showing soundness and relative completeness of Reynolds' logic. We introduce a program modality to obtain dynamic separation logic, a novel logic in the spirit of dynamic logic. We then investigate an alternative weakest precondition axiomatization and strongest postcondition axiomatization of Reynolds' logic with respect to classical separation logic (see Section 4.5). This approach directly leads to solving an open problem, in which the local axioms of Reynolds' logic and the frame rule can be used to derive the global axioms, but without using the magic wand connective. Furthermore, our approach is robust, and as such can also be adapted to intuitionistic separation logic: resulting in an alternative weakest precondition axiomatization and a novel strongest postcondition axiomatization, given in Chapter C of the appendix (these results are not yet published).

Background material is presented in the appendix: Chapter A gives the necessary results from classical logic, such as syntax, semantics, basic results from model theory and proof theory, and soundness and completeness. Chapters 2 and 3 of the first part of this thesis assume that the reader is familiar with this background material. Furthermore, Chapter B gives the necessary background from program verification, such as syntax of programs, operational semantics, denotational semantics, axiomatic semantics, and recursive procedures. Chapter 4 of the second part of this thesis assumes that the reader is familiar with this background material.

The background material does not contain any novel scientific contributions, only the presentation of the material is original. Chapter B is based on the following publication:

- *Completeness and complexity of reasoning about call-by-value in Hoare logic*
  Frank S. de Boer, Hans-Dieter A. Hiep
  In: *ACM Transactions On Programming Languages And Systems*
  Volume 43, Issue 4
  Association for Computing Machinery, 2021

Some of the results in this thesis have an accompanying Coq formalization to increase the confidence in the correctness of the presented results (Chapter D). This is not the first formalization of separation logic in a formal interactive theorem prover (see e.g. [222]), but it does show the correctness and the base case of relative completeness of our novel alternative axiomatization presented in Section 4.5 and the novel axiomatizations for intuitionistic separation logic presented in Chapter C.

Also an alternative logic for describing the state of memory, that is useful for reasoning about pointer programs, has been investigated—related to the work presented in this thesis. In there, abstract object creation is investigated and its connection to a second-order logic as assertion language, resulting in a novel substitution-like operator for computing a weakest precondition. A case study of a linked list data structure is described, where the other approach is compared to separation logic. This work resulted in the following publication, but is not included in this thesis:

- *Footprint logic for object-oriented components*
  Frank S. de Boer, Stijn de Gouw, Hans-Dieter A. Hiep, Jinting Bian
  In: Formal Aspects of Component Software: 18th International Conference, FACS 2022, Proceedings
  Lecture Notes in Computer Science, volume 13712
  Springer, 2022

As mentioned before in a footnote, the motivation for starting the research described in this thesis comes from practical experience with the KeY verification system, in particular the investigation of the correctness of the linked list data structure in the standard library of the object-oriented programming language Java. During these investigations a critical bug was found, thereby demonstrating the usefulness of program verification in practice. These practical experiences, and the approach based on dynamic logic for reasoning about pointer structures, have been published in the following articles:

- *Verifying OpenJDK's LinkedList using KeY*
  Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, Stijn de Gouw
  In: Tools and Algorithms for the Construction and Analysis of Systems, 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Part II
  Lecture Notes in Computer Science, volume 12079
  Springer, 2020

- *Verifying OpenJDK's LinkedList using KeY (extended paper)*
  Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Stijn de Gouw
  International Journal on Software Tools for Technology Transfer, volume 24
  Springer, 2022