New Foundations for Separation Logic

H.A. Hiep

May 23, 2024

ISBN 978-90-831826-1-2 (paperback)
ISBN 978-90-831826-2-9 (e-book, PDF without DRM)
ISBN 978-90-831826-3-6 (digital artifact)
NUR 123 Exacte vakken en informatica (hoger onderwijs)
Engelstalig

Omslag: Ulysses and the Sirens, John William Waterhouse (Google Art Project)
Typografie door auteur zelf met behulp van LaTeX 2ε.

Uitgave: eerste uitgave
Oplage: 64
Aantal pagina's: 256
Drukkerij: proefschriftenprinten.nl (Kelvinstraat 27, 6716 BV, Ede)

New Foundations for Separation Logic

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op donderdag 23 mei 2024
klokke 13:45 uur

door

Hans-Dieter Anton Hiep
geboren te Hoorn
in 1991

Promotor:
prof.dr. F.S. de Boer

Co-promotores:
dr. C.P.T. de Gouw (Open Universiteit)
dr. A.W. Laarman

Promotiecommissie:
prof.dr. J.-P. Katoen (Universiteit Twente, RWTH Aken)
dr. J.A. Pérez (Rijksuniversiteit Groningen)
dr. H. Basold
prof.dr. M.M. Bonsangue
prof.dr. H.C.M. Kleijn
prof.dr. A. Plaat

Part of the IPA Dissertation Series: No. 2024-04

The research presented in this dissertation was carried out at the Dutch national research laboratory for mathematics and computer science Centrum Wiskunde & Informatica (CWI) in Amsterdam, and the Leiden Institute of Advanced Computer Science (LIACS) of Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

The author was partially supported by funding from NGI ASSURE, a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No. 957073.

Abstract

This thesis presents new foundations for separation logic, an important field within the formal sciences such as theoretical computer science. Around the turn of the millennium, separation logic was introduced by J.C. Reynolds with the goal to make reasoning, about the correctness of computer programs that work with so-called 'pointers', more efficient than earlier formal methods. Reynolds' method and the other earlier methods are both extensions of the proof system introduced by C.A.R. Hoare for reasoning about the correctness of simple while-programs. The essence of Reynolds' initial work, which was researched and put into practical use by many other scientists, consists of two extensions of the work initiated by Hoare: the first extension adds to first-order predicate logic two new propositional connectives (the so-called separating conjunction and separating implication); the second extension adds to Hoare's program logic new proof rules for reasoning about (the primitive operations of) pointer programs. These primitive operations are used for reading memory ('lookup'), writing memory ('mutation'), reserving memory ('allocation'), or freeing memory ('deallocation').

The new foundations are presented in two parts. The following paragraphs summarize the contents of these two parts. The first part contains a model-theoretic and proof-theoretic exploration of the classical interpretation of separation logic, the logic used in Reynolds' assertion language. This first part proves a result for separation logic, that is as fundamental as the corresponding result by Gödel in first-order logic—the completeness theorem. The second part contains a new interpretation of Reynolds' program logic, and introduces—for the first time—so-called dynamic separation logic. Dynamic separation logic is an extension of dynamic logic by D. Harel. Using dynamic separation logic, an alternative weakest precondition axiomatization and strongest postcondition axiomatization is given. These alternative axiomatizations, in contrast to earlier axiomatizations of Reynolds' program logic, do have the property of *gracefulness*: the earlier axiomatizations of Reynolds' program logic are not graceful, because they unnecessarily increase the complexity in the use of separating connectives when generating weakest preconditions or strongest postconditions.

Chapter 2 of the first part demonstrates the inadequacy of the standard interpretation of separation logic, because it lacks compactness. This chapter also introduces a new interpretation, called the full interpretation of separation logic, that is based on the possibility of evaluating formulas also with respect to infinite heaps. However, also this full interpretation is inadequate. We continue with a search for necessary and sufficient conditions for embedding the standard interpretation into the full interpretation, and we introduce so-called relational separation logic with the goal to compare separation logic with second-order predicate logic. It is an interesting fact that the full interpretation of separation logic lies close to the standard interpretation of second-order logic: expressivity of a so-called binding operator is sufficient for showing that these two logics coincide.

Chapter 3 of the first part introduces a proof theory with a corresponding new interpretation of separation logic that is based on first-order definable heaps. This new interpretation allows us to show that the resulting proof system is in fact adequate. The proof system is presented as a sequent calculus, but also a second proof system is introduced in the natural deduction style. The sequent calculus is sound and complete with respect to first-order definable heaps. The natural deduction calculus is sound and complete with respect to structures that satisfy a semantic comprehension condition. The second proof system works with more general formulas than what is allowed in separation logic, by introducing a connective that is closely related to the binding operator of the previous chapter.

Chapter 4 of the second part comprises: general interpretations of separation logic, and a class of structures based on so-called memory models. The latter class is used in the proof of soundness and relative completeness of Reynolds' program logic. We arrive at dynamic separation logic by introducing a program modality in the assertion language. We research an alternative weakest precondition axiomatization and strongest postcondition axiomatization for classical separation logic. This work directly leads us to solving an open problem, where it is the question whether the global axioms can be inferred from the local axioms and the frame rule of Reynolds' program logic but without additionally using the separating implication connective. Our method is robust, since it can also be applied to obtain a weakest precondition axiomatization and strongest postcondition axiomatization for intuitionistic separation logic (this result is not yet published).

The thesis also includes an extensive appendix with background material, concerning higher-order predicate logic and Hoare's program logic, that is needed to understand and appreciate the above novel results. The appendix also describes an accompanying Coq formalization of the soundness and completeness of the alternative axiomatizations of Reynolds' program logic, that aims to increase the trust one may place in the validity of the results.

Preface

This thesis is the result of my promotion trajectory, that ran from 1 November 2018 until 31 October 2023 (5 years), executed at two institutes: Centrum Wiskunde & Informatica (CWI) in Amsterdam from 1 November 2018 until 31 October 2020 (2 years); and Leiden Institute of Advanced Computer Science (LIACS) in Leiden from 1 November 2020 until 31 October 2023 (3 years). At the CWI, I was part of the Formal Methods group (FM), and at LIACS, I was part of the theory group. The CWI provided me with office space for the entire duration of the trajectory. The promotion was initiated by Frank de Boer (promotor) and Stijn de Gouw (co-promotor), with the initial goal of verifying standard libraries of the Java programming language using the KeY system.

The first years were quite productive and this would not be possible without Frank giving me a lot of freedom to explore, to develop independently, and to take initiative. We often had productive meetings, and structured our collaboration by means of writing papers together. I also collaborated with Stijn's master student, Olaf Maathuis, while we were both learning how to use the KeY system to verify Java's `LinkedList` class. Jinting Bian joined our group at CWI, and I helped her with learning how to use KeY so we could collaborate on Java library verification.

In these initial years I also submitted grant proposals, and collaborated with Benjamin Lion, Kasper Dokter, Roy Overbeek, and Farhad Arbab. Some of the grant proposals were accepted: a new project with the code name Reowolf started that was later extended in a project named Reowolf 2.0. As part of these projects we were able to hire scientific programmers, Christopher Esterhuyse and Max Henger, with whom I collaborated on completing the deliverables of the projects. Christopher convinced me to use the Rust programming language for the project, which I had not used before. This allowed me to gain more practical experience with programming under a linear typing discipline. Max has taught me to take the concerns of efficiency and scalability more seriously than I did before.

From the start, I was involved in teaching in the Program Correctness course at Leiden University. In this course we explain Hoare's logic to bachelor students, both for simple while-programs and programs with recursive procedures, and we practice with a simplified version of the KeY system. Later on, teaching both the Program Correctness and Concepts of Programming Languages courses also became my responsibility. For the latter course, I redeveloped the course material and recorded an on-line lecture video series.

After years of progress in this initial direction and several semesters of involvement in teaching, Frank invited me to collaborate on an article, submitted to the Association for Computing Machinery (ACM) journal Transactions on Programming Languages and Systems (TOPLAS), on the completeness and complexity of a Hoare-like logic for reasoning about call-by-value procedures. For that article, I contributed the Coq formalization proving several tricky supporting lemmas and came up with the idea of applying techniques from proof theory to do proof normalization in Hoare's logic to prove the complexity result that correct programs have linear proofs. On my own initiative, I presented the basis of this work at the PhD Day organized by the VvL (the Dutch Association for Logic and Philosophy of the Exact Sciences) on July 1st, 2022.

Afterwards, I wanted to change the direction of my own research towards the investigation of the foundations of separation logic (while continuing collaboration with Jinting Bian on verifying Java libraries, continuing work on the Reowolf project, and continuing teaching Concepts of Programming Languages and Program Correctness). There were several reasons for considering this change of direction: firstly, we received numerous anonymous reviews in response to our earlier articles about Java program verification that mentioned separation logic as related work. Secondly, on several occasions Frank indicated he was a contrarian in this subfield, of separation logic, so I was inclined to become a *meta-contrarian*[1]. Lastly, I had many interesting discussions about separation logic during the PhD Day organized by the VvL. Although at that time I had only superficial knowledge, I started to wonder: what the hell is separation logic, really? Already, I had done several years of work of a practical nature, in actual Java program verification, and in the mean time I had learned more about higher-order logic, set theory, model theory, proof theory, and foundational issues in mathematics. Now I wanted to do more work of a theoretical nature in mathematical logic, and continue my work on formalizing Hoare's logic.

So I convinced Frank that it was a good idea to investigate separation logic. Our approach would be from a foundational point of view, and my goal was to understand what were the issues in separation logic that Frank refrained from articulating in the past twenty years or so. What emerged was a symbiotic relationship between me and my promotor: I gladly took Frank as a confident oracle, and saw myself as a skeptical verifier. In this period we worked together intensively, often spending many hours a day discussing next to a whiteboard. Also I used the Coq proof assistant to meticulously check my work. But at other times, the roles reversed, and I saw myself as the oracle while Frank was verifying my 'nonsense', critically and skeptically questioning my position until we obtained something reasonable. The benefit of our symbiosis was that we discovered many of our own mistakes, that we were able to repair ourselves. As such, I was deeply involved in the discovery, the refinement, the verification, and the presentation of the subject matter that is presented in this thesis. Frank and I collaborated on a

[1] Thanks to Benjamin for explaining to me why I am an 'intellectual hipster': whereas Frank is a contrarian (i.e. opposing separation logic), I took an opposite position in Frank's contrariness (thus opposing Frank's opposition, in defense of separation logic).

paper until no longer there would be any ground to oppose each other, and then we involved my first co-promotor, Stijn, to check the intermediate paper—whether what we did made sense. Finally, after this thesis was written, also my second co-promotor, Alfons Laarman, was involved to check whether the thesis as a whole made sense. I found this way of working to be very productive.

<rant> Whereas the collaboration between me and my promotor and co-promotors was very productive, I found that there was also a source of counter-productivity: the anonymous reviewers of our articles about separation logic. As mentioned earlier, we had structured our collaboration by means of writing articles that were submitted for presentation at several conferences. However, that last part, submission to conferences and subjecting our articles to objective anonymous reviewers, was severely frustrating our productivity. Contrary to the positive symbiotic relation between me and my promotor, I had experienced the relation between me and my anonymous reviewers as negative, even alienating. It felt I was in a toxic burn pit that slowly burned me out. The epitome of toxicity was when a reviewer was rejecting, insistingly, our article on the basis of a counter-example to our result, that was also a counter-example to Gödel's completeness theorem! Communication with anonymous reviewers was very limited and the reviewers did not respond to requests to discuss the matter further.</rant> In an attempt to prevent any confusion about what the background material is, I spent several months writing the appendix.

In the end, I am deeply indebted to Frank for his encouragement: to continue to defend these new foundations for separation logic, and to regard less of negative and discouraging comments by anonymous reviewers. I am also delighted by the fact that Stijn and Alfons were always able to give useful and constructive feedback on papers or this thesis. I am grateful to all the members of the PhD committee for reading a preliminary version of this thesis and giving valuable feedback that has lead to an improved and final version.

x

Acknowledgments

I wish to thank, besides the people already mentioned in the preface, also the following people for providing a most pleasant working environment and/or their (direct or indirect) support of me while I was working on this thesis:

Vlad Serbanescu, Keyvan Azadbakht, Jana Wagemaker, Jurriaan Rot, Jan Rutten, Luc Edixhoven, Sung-Shik Jongmans, Carl Schulz, Maarten Dijkema, Marco Floor, Henk Roose, Emil Gorter, Vera Sarkol, Annette Kik, Minnie Middelberg, Nada Mitrovic, Doutzen Abma, Dick Broekhuis, Margriet Brouwer, Martine Anholt Gunzeln, Susanne van Dam, Ramona Rij□, Remco Westra, Krzysztof Apt, Jos Baeten, Erik de Vink, Marten van Dijk, Chenglu Jin, Niloufar Sayadi, Chao Yin, Sirui Shen, Steven Pemberton, and all other collegues at the CWI;

Mike Preuss, Walter Kosters, Hendrik Jan Hoogeboom, Rudy van Vliet, Jeannette de Graaf, Mitra Baratchi, Frank Takes, York-kam Kwok, Esme Caubo, Joyce Glerum, Riet Derogee, Chris Flinterman, Caroline de Bruin, Hui Feng, Lieuwe Vinkhuijzen, Luc Edixhoven, Dalia Papuc, Tanjona Ralaivaosaona, Miguel Blom, Tim Coopmans, Sebastiaan Brand, and all other collegues at LIACS;

Wan Fokkink, Jasmin Blanchette, Herbert Bos, Robbert Krebbers, Frits Vaandrager, Marieke Huisman, Loek Cleophas, Thomas Neele, the late Eelco Visser, Wolfgang Ahrendt, Reiner Hähnle, Bernhard Beckert, Richard Bubel, Mattias Ulbrich, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, Volker Stolz, Violet Ka I Pun, Crystal Chang Din, Dominic Steinhöfel, Eduard Kamburjan, Michael Kirsten, Alexander Weigl, Lars Tveito, Asmae Heydari Tabar, Alexander Knüppel, Michiel Leenaars, Mirko Ross, Stephen Farrell, Chris Verhoef, Bas van Bockel, and all other (international) collegues I worked with;

Wesley Shann, Daniel Roos, Lazlo de Wijs, Oualid Azzeggarh, Zahir Bingen, Andy Tatman, Wessel van der Goot, Renz Roos, Roos Wensveen, Dominique Lawson, Perri van den Berg, Dirck van den Ende, and all my other students;

Joris Bierkens, David van Oldenhof, Anders Rehult, Diederik Malien, Xavier Boot, Jim Lemmers, Suzanne Kraaij, Laurens van Kempen, Micha Klamer, Koen van Veen, Eric Ruts, Jacob Kooijman, Alan Hopman, Thijs Louwman, Maarten Dinkelberg, Thijs Dekker, and all my other friends;

Mara and Pandora Visser, Jorien van den Heuvel, my brother, and my parents; and all others I forgot to mention here.

Contents

Chapter 1

Introduction

In the well-known 1960s article *The unreasonable e ectiveness of mathematics in the natural sciences* by E.P. Wigner [225], and in the 1980 article *The unreasonable e ectiveness 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 e ectiveness 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* o er an explanation for the unusual e ectiveness of logic. Essentially, their argument boils down to recognizing the di erence 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 o ers 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 a□ected, 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 a□ected 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 a□ected by the operation. Explicit alias analysis may complicate practical reasoning for two main reasons:

- it is di□cult to modularize reasoning about fragments of the heap—that is, it is di□cult 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 di□cult 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 di□erent 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 di□culty, encountered during practical verification e□orts 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 $_1 \ ::: \ _n$ [in the notation of this thesis], where each [component] $_i$ described a distinct region of storage, so that an assignment to a single location could change only one of the $_i$. I believe that this idea of organizing assertions to localize the e□ect 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 di□erent components, and by allowing pointers both to and from di□erent 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 o□ers 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 a□ected 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 ($\dagger$) are prime subjects of this thesis, where novel contributions are made. The logic marked by a double dagger ($\ddagger$) 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 *, !* , 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 *f g S f g*, where the *precondition* and *postcondition* 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 describes the expected final state after running the program *S* from an initial state satisfying the precondition . We focus on partial correctness semantics of Hoare's logic.

   In *Reynolds' logic* one reasons also about triples *f g S f g*, 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*] for every given program *S* and postcondition . The Hoare triples *f g S f g* of Hoare's logic (of the level above) are embedded in dynamic logic as the implication *!* [*S*] : 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 *f g S f g* of Reynolds' logic (of the level above) can be embedded into dynamic separation logic as the implication *!* [*S*] , 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 su□cient, 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 di□erent 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):

$$(x \mapsto \_) \quad ((x \mapsto 0) \quad (y \hookrightarrow z)) \tag{1.1}$$

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

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

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 su□cient 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 is a (syntactic or semantic) consequence of a theory in two essentially different ways: either the semantic way, by showing a counter-model that satisfies the theory but not , or the syntactic way, by showing there is a proof with premises in and conclusion . 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 is a (syntactic or semantic) consequence of a theory of separation logic formulas 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

Chapter 2

Model theory of separation logic

In this chapter, we introduce the syntax and semantics of separation logic. We shall present the standard interpretation of separation logic, but also investigate a new interpretation of separation logic and the relations between the different interpretations of separation logic, and between separation logic and second-order logic. The first result is 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. An interesting result is that the full interpretation of separation logic is close to the standard interpretation of second-order logic, in the sense that the expressivity of a binding operator is sufficient for the two logics to coincide. As such, this chapter is a model theoretic investigation of separation logic.

Informally, the purpose of separation logic is to formalize and allow reasoning about the notion of spatial separation. This intuition can best be elucidated by the following examples, in which we see the different meaning in natural language of the conjunction of two facts:

- "I know the moon orbits the earth."
- "I have a Euro in one of my pockets."

Notice how the meaning of conjunction differs in the following sentences:

- "I know the moon orbits the earth, and I know the moon orbits the earth."
- "I have a Euro in one of my pockets, and I have a Euro in one of my pockets."

In the first sentence, describing the knowledge that the moon orbits the earth twice does not change the way we could interpret the overall sentence. Once you know

something, it does not matter how often you think of it: we have the classical propositional law that states that $A \wedge A$ is equivalent to A for any proposition A. This works for any 'pure' proposition.

However, in the second sentence there is a difference between our classical intuition, and our spatial intuition. Classically, we are able to substitute any sentence for the proposition A, and hence the propositional law should also apply here. But this does not entirely capture our spatial intuition. The second sentence is not precise: is the Euro in the same pocket, or in different pockets? Phrased differently, is the expression 'one of my pockets' in both conjuncts referring to the same pocket or to two different pockets? Classically, to be precise, one would have to explicitly describe this situation: "I have a Euro in one of my pockets, and I have a Euro in another of my pockets."

Notice how the explicit difference in location, which we classically need to describe to be precise, can also be resolved differently: by changing the way we interpret the conjunction—no longer classically, but spatially: "I have a Euro in one of my pockets, and separately, I have a Euro in one of my pockets." Surely the two pockets must now be different pockets, because how could otherwise one fact be separate from the other fact?

We introduce the following symbols to abbreviate our intuition. We write $(x \hookrightarrow y)$ to express that the location x has the value y. If we interpret x as being the location of one of my pockets and y as the value of one Euro, then $(x \hookrightarrow y)$ expresses that "I have a Euro in one of my pockets". Further, we introduce the spatial conjunction, or separating conjunction, by using $*$ as a connective. Symbolically, we can evaluate the following formulas:

1. $\exists x(x \hookrightarrow y)$,
2. $(\exists x(x \hookrightarrow y)) \wedge (\exists x(x \hookrightarrow y))$,
3. $(\exists x(x \hookrightarrow y)) * (\exists x(x \hookrightarrow y))$,
4. $\exists x((x \hookrightarrow y) \wedge \exists z(z \neq x \wedge (z \hookrightarrow y)))$.

Notice how the first two formulas are equivalent (due to the classical law that $A \wedge A$ is equivalent to A). However, the second and third formulas are not equivalent. The second formula expresses that at least one pocket has a Euro in it. The third formula expresses that at least two (different) pockets have a Euro in them. Intuitively, the third and fourth formula are again equivalent, where in the fourth formula we classically express that the second pocket is different from the first pocket.

If we scale up our argument to more and more pockets (for example, imagine the many pockets of a handyman), we observe the following pairs of equivalences:

- $(\exists x(x \hookrightarrow y)) * (\exists x(x \hookrightarrow y)) * (\exists x(x \hookrightarrow y))$,
- $\exists x((x \hookrightarrow y) \wedge \exists z(z \neq x \wedge (z \hookrightarrow y) \wedge \exists w(w \neq x \wedge w \neq z \wedge (w \hookrightarrow y))))$.

and

- $(9x(x \not\to y)) \quad (9x(x \not\to y)) \quad (9x(x \not\to y)) \quad (9x(x \not\to y))$,
- $9x((x \not\to y) \wedge 9z(z \in x \wedge (z \not\to y) \wedge 9w(w \in x \wedge w \in z \wedge (w \not\to y) \wedge 9v(v \in x \wedge v \in z \wedge v \in w \wedge (v \not\to y)))))$.

We see how, classically, we need to describe more and more facts that state that the locations are all pairwise different, whereas with our spatial intuition we simply declare these locations to be separate by the separating conjunction connective. The separation is either described 'bottom up' using explicit equational facts, or 'top down' using separating connectives.

Especially in the setting of programming with pointers, this spatial intuition is natural to reason about. Often, data structures are laid out in separate parts of the memory, and describing explicitly that these parts of the memory are separate quickly grows in complexity. Consider, for example, the circular singly-linked list of Figure 2.1. Whenever we informally reason about the data structure of a linked list, we mentally model the memory state by means of a picture in which each box represents some storage space in memory, and pointers to boxes represent the address value of the location of that memory space. Already by choosing such a picturesque model, we have the graphical intuition that locations in space are separate: drawing two boxes on paper means the two boxes have to be separate. However, classically, one would have to explicitly describe that to be precise.

Consider running through the execution of a program that manipulates the memory states of a linked list:

(a) In the initial state we have one item which is linked back to itself. This is the so-called head of the list. In Figure 2.1 we see this situation, where x is a variable that points to the location of the box, and the value of the box points to itself. Symbolically, we would describe this by stating:

$$(x \not\to x) \wedge 8z((z \not\to \quad) \not\to z = x)$$

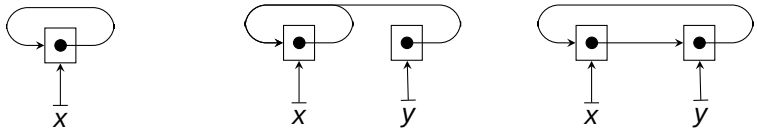
where $(z \not\to \quad)$ means $9y(z \not\to y)$. This describes that the value of x represents a location, which is allocated since we see it points to a box, and the value of the location is the address of itself, since we see a pointer from inside the box to the edge of the box itself. The second conjunct expresses that there are no other locations. We can abbreviate this formula, by simply writing:

$$(x \not\to x)$$

where the symbol $\not\to$ indicates that the location is the sole location that is allocated.

(b) Next, we allocate new space. So y now points to a location that was previously unallocated, but now it is allocated since it takes up space as a box. We obtain the 'paperclip' state, in which the box to which x points still points to itself, but also the box to which y points points to the box to which x points. Symbolically we have

$$(x \not\to x) \quad (y \not\to x)$$



(a) Initial state (b) After creation (c) Relinking

Figure 2.1: Three different memory states of a circular singly-linked list.

since we can imagine to split the memory in two parts: the box on the left, and the new box on the right. The box on the left still points to itself. The box on the right points to the box on the left.

(c) Finally, we relink the previous box to the new box: the box to which x points itself now points to the box to which y points. Symbolically, we express this situation as:

$$(x \mapsto y) \quad (y \mapsto x)$$

and this description is quite precise. We know, from the picture, that x and y must point to *different* boxes, because we have drawn these boxes apart from each other. Notice that the two components each point to each other. The components, when viewed isolated from each other, have *dangling pointers*. However, by putting the components in separating conjunction, these former dangling pointers now resolve to the proper location, being the box in each component.

This simple example convincingly shows that pointer structures can be described, component-wise, using the connective of separating conjunction. Separating conjunction directly captures the intuition that the locations to which one refers to are separated among the components of the conjunction. Further examples of cyclic data structures and containers can be given: doubly-linked lists, tree structures with pointers from parents to children but also back links from children to parents. In fact, our intuition of these pointer structures quite naturally transfers to the memory states of object-oriented programs, in which the precise identity of objects is abstracted away, where different pointers coming out of a box represent the different fields of an object.

Another important aspect of spatial intuition is hypothetical space: the question what happens if one would allocate locations according to a description, where those locations are previously not allocated. To describe hypothetical situations we introduce the connective $-\!\!*$ that is called separating implication, or the magic wand. Consider Figure 2.2 in which there are two situations depicted. In the situation on right we are dealing with a hypothetical extension of the situation depicted on the left, as represented by a dashed arrow. We can describe these situations as follows:

(a) This situation is described precisely by the formula

$$(x \mapsto y) \wedge (y \not\mapsto \_) :$$

(a) Original state (b) Original state and extended state

Figure 2.2: Hypothetical extension of locations.

The location pointed to by y is not allocated, as depicted by a crossed-out box. Hence both the box pointed to by x, and to which y points, are dangling pointers: the location to which is pointed is not allocated.

(b) Now, we could imagine a hypothetical extension of the state, in which the box pointed to by y actually exists and points back to x. This hypothetical situation can be described by the formula

$$(x \mapsto y) \wedge (y \mapsto \quad) \wedge ((y \mapsto x) \quad ((x \mapsto y) \quad (y \mapsto x))).$$

The connective describes on the right of the connective what holds of the resulting, hypothetical memory state after the current memory state is extended by any separate part of the memory for which the left of the connective holds.

In this chapter we shall formally introduce *separation logic*. In separation logic we aim to formally capture our intuition as given above. We introduce the language in which formulas are described, their interpretation by means of structures. In this chapter we focus on the model theoretic development of the semantics of separation logic. Next chapter we develop the proof theory for reasoning about formulas of separation logic.

Technically, we restrict ourselves to classical first-order separation logic. By classical we mean that we interpret the formulas of first-order logic embedded in separation logic classically. Although the language of separation logic extends the language of first-order logic, not all classical laws such as the law of excluded middle hold for all classical separation logic formulas. The embedding of first-order logic in separation logic is also called the 'pure' part of separation logic; it is pure since it does not depend on our spatial intuition. Further, we restrict to first-order logic and do not develop higher-order separation logic, to keep the presentation light. In principle, nothing restrains us from allowing higher-order variables and higher-order quantification in separation logic too. Furthermore, just like in Chapter A that introduced first-order logic, we keep terms orthogonal to our discussion. Although terms can be easily added to the syntax, and we can interpret constant symbols and function symbols specially as individuals and functions, it is not necessary to do so—we do not miss out any expressivity of the logic, but we get simpler definitions by leaving them out.

Throughout this chapter, we will introduce different classes of separation logic formulas. An overview of these classes of formulas is given in Figure 2.3.

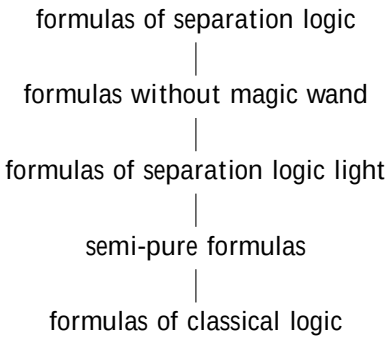


Figure 2.3: Overview of the classes of formulas of separation logic. Formulas are included from bottom to top.

2.1 Syntax of separation logic

The language of separation logic imports many of the concepts already present in classical logic, such as variables and signatures. Formulas, or synonymously assertions, of separation logic extend the formulas of classical logic in two aspects: first, we add a new primitive formula, called *points-to*, denoted by $\_ \mapsto \_$, and, secondly, we add two new connectives, called *separating conjunction* and *separating implication*, denoted by $*$ and $-\!\!*$, respectively. After extending formulas, some concepts imported from classical logic require some adjustments to the setting of separation logic. We also introduce new concepts that are were not yet present in classical logic.

In our presentation of the assertions of separation logic, we shall not introduce terms at first. This makes a comparison with our presentation of classical logic easier, and also makes the technical development easier to follow. However, by taking this approach, we do not lose any expressive power: later in this section we add terms in an orthogonal way, regardless of whether the assertion language of classical logic or separation logic is used.

Based on a given signature (see Definition A.1.2), we construct the formulas of separation logic. Although we follow [125] in the definition of the syntax (and later also the standard semantics) of the assertion language of separation logic, we now work in this more general setting with signatures and we use a different atomic 'weak points to' formula. For the remainder of this section, we fix a first-order signature Σ that consists of constant symbols, similar to what we did for classical logic. It should be clear from context whether one speaks of formulas of separation logic or formulas from classical logic. In this chapter, 'formula' refers to 'formula of separation logic' unless explicitly mentioned otherwise.

Definition 2.1.1 (Formulas). A *formula* is constructed inductively as follows:

1. $\bot$ is a formula,
2. $(x \doteq y)$ is a formula if x and y are individual variables,

3. $(x \mapsto y)$ is a formula if x and y are individual variables,
4. $C(x_1, \ldots, x_n)$ is a formula if C is a constant symbol of arity n and $x_1, \ldots, x_n$ are individual variables,
5. $(\quad \Rightarrow \quad)$ is a formula if and are formulas,
6. $(\forall x \ )$ is a formula if is a formula and x an individual variable,
7. $(\qquad)$ is a formula if and are formulas,
8. $(\qquad)$ is a formula if and are formulas.

All formulas are constructed by one of these eight clauses. Alternatively, we can define formulas by the following abstract grammar:

$$\ ::= \bot \mid (x \doteq y) \mid (x \mapsto y) \mid C(x_1, \ldots, x_n) \mid$$
$$(\qquad) \mid (\qquad) \mid (\quad \Rightarrow \quad) \mid (\forall x \ )$$

Note that in separation logic, we prefer using false and true instead of $\bot$ and $\top$, where false abbreviates $\bot$, and true abbreviates $(\bot \Rightarrow \bot)$.

The first four clauses construct primitive formulas, the last four clauses construct complex formulas. Formulas can still be regarded as finite sequences of symbols, but we consider the newly introduced symbols $\mapsto$, $\Rightarrow$, , to be *separation symbols* disjoint from the earlier *logical symbols* and *non-logical symbols*. It is easy to verify that the set of formulas, as defined above, is recursive.

We speak of formulas in the usual way, except for the new clauses. The primitive formula $(x \mapsto y)$ is called *points-to* (as in 'x points to y'). As complex formulas, two separating connectives are given: $(\qquad)$ is a *separating conjunction*, and $(\qquad)$ is a *separating implication*. The latter connective is also called the *magic wand* by some authors.

Again, when proving meta-properties of formulas, we proceed by induction on the *complexity* of formulas. There are different obvious measures of complexity: the *height of a formula*, by viewing the formula as a parse tree and taking the height of that tree, or the *length of a formula*, by viewing a formula as a sequence of symbols and taking the length of that sequence.

In a certain sense, Definition 2.1.1 includes all first-order formulas of classical logic (cf. Definition A.1.4). Regarding formulas as sequences of symbols, if in that sequence there is no separation symbols present then the formula must also be a (first-order) formula of classical logic. The latter formulas are also called *pure* formulas or *heap-independent* formulas. If a formula, seen as a sequence of symbols, contains a separation symbol we call it *impure* or *heap-dependent*. If a formula is impure but does not contain separating connectives (i.e. contains primitive points-to constructs) then we call it *semi-pure*.

As such, we may use the usual classical abbreviations, given in Definition A.1.5, too in separation logic. This means we have access to (logical) disjunction, conjunction, bi-implication and existential quantification. Further, we also use these

abbreviations in the case where they are used to compose heap-dependent formulas. Specifically, we may use separating connectives nested under logical connectives or quantifiers.

We further have the separation symbols $\not\mapsto$, emp, $\not\rightarrow$ and we use the symbol $\_$ as a placeholder, by introducing the following abbreviations:

$$
\begin{aligned}
(x \not\mapsto y) &\text{ abbreviates } (\neg(x \mapsto y)) \\
(x \mapsto \_) &\text{ abbreviates } (\exists z(x \mapsto z)) \\
(x \not\mapsto \_) &\text{ abbreviates } (\neg(x \mapsto \_)) \\
\mathsf{emp} &\text{ abbreviates } (\forall x(x \not\mapsto \_)) \\
(x \not\rightarrow y) &\text{ abbreviates } ((x \mapsto y) \wedge (\forall z, w((z \mapsto w) \Rightarrow x \doteq z))) \\
(x \not\rightarrow \_) &\text{ abbreviates } (\exists z(x \not\rightarrow z))
\end{aligned}
$$

where z is a fresh individual variable (i.e. not the same as x). We may speak of 'locations' being on the left-hand side of either $\mapsto$ or $\not\rightarrow$, and 'values' being on the right-hand side of either $\mapsto$ or $\not\rightarrow$. We may speak about these formulas in the following way:

- for $(x \mapsto y)$ we say 'x points to y' or 'location x has value y',
- for $(x \mapsto \_)$ we say '(at least) x is allocated',
- for $(x \not\rightarrow y)$ we say 'strictly x points to y' or we may say 'x points to y and only x is allocated',
- for $(x \not\rightarrow \_)$ we say 'x is solely allocated' or 'x is allocated alone',
- for emp we say 'nothing is allocated'.

When speaking of the negated forms, some care is needed:

- for $(x \not\mapsto y)$ we say 'x does not point to y' or 'x has not the value y', but note that does not necessarily mean x is not allocated,
- for $(\forall y(x \not\mapsto y))$ we say 'x has no value' or 'x is not allocated',
- equivalently, for $(x \not\mapsto \_)$ we say 'x is not allocated' (but that does not say anything about other allocations).

Remark 2.1.2. In some texts on separation logic, the symbols emp and $\not\rightarrow$ are taken as primitive formula (instead of $\mapsto$). In this case we recover the same language as we have here by taking $(x \mapsto y)$ as an abbreviation of $(\mathsf{true} \quad (x \not\rightarrow y))$. In other texts the 'weak' points to is taken as primitive (as in [187] and [70]). The reason we take $\mapsto$ as primitive is to avoid the use of separating connectives in expressing our abbreviations, while still being able to express that an element is not allocated: all abbreviations are semi-pure.

The same conventions for reducing parentheses are employed, with the following two additions to precedence: separating conjunction precedes logical conjunction (and so also disjunction and logical implication), and separating implication precedes logical implication (and so also bi-implication). All separating connectives also associate to the right. For example, $P(x)\ \ Q(x) \wedge P(x)$ disambiguates to $((P(x)\ \ Q(x)) \wedge P(x))$, and $P(x) \mathrel{!} Q(x)\ \ P(x) \mathrel{!} Q(x)$ disambiguates to $(P(x) \mathrel{!} ((Q(x)\ \ P(x)) \mathrel{!} Q(x)))$. However, we shall try to use parentheses, even if not necessary by these disambiguation rules, to present formulas as clearly as possible also for readers less familiar with separation logic.

The concept of variable occurrences in formulas can be imported in a straightforward manner. Formulas in separation logic can also be viewed as a parse tree, in which variables occurs at leaves. We also have the sets $FV(\ )$, and $BV(\ )$, that denote the variables that occur free in , and the variables that occur bound in , respectively. Revisiting Definition A.1.6 (Free and bound variables), we need to add the following clauses:

- $FV(x, \mathrel{!}\ y) = \{x; y\}$ and $BV(x, \mathrel{!}\ y) = ;$,
- $FV(\qquad) = FV(\ ) \cup FV(\ )$ and $BV(\qquad) = BV(\ ) \cup BV(\ )$,
- $FV(\qquad) = FV(\ ) \cup FV(\ )$ and $BV(\qquad) = BV(\ ) \cup BV(\ )$.

As such, we also import the following concepts: a formula without free variables is a *sentence*, a *context* is a list of formulas, and a *theory* is a set of sentences.

Similarly, we have also the application $(\ )$ of a renaming to a formula . Revisiting Definition A.1.9 (Variable renaming), we need to add the following:

- $(x \doteq y) = (\ (x) \doteq\ (y))$,
- $(\qquad) = (\ (\ )\quad(\ ))$,
- $(\qquad) = (\ (\ )\quad(\ ))$.

Also (capture-avoiding) substitution of variables for variables works in the same way as in classical logic. We can also add terms to separation logic in the same manner as is done in Section A.5.

2.2 Standard semantics

The standard semantics of separation logic formula is given in the style of Tarski, extending the semantics of classical logic. There are two important aspects to consider before we define the satisfaction relation formally.

The first aspect is that we employ the same structures that we used for giving semantics to classical logic: this ensures that the semantics of the heap-independent formulas of separation logic coincides with their classical semantics. Further, terms are evaluated without depending on the heap. So the semantics of terms in separation logic completely coincides with the semantics of terms in classical logic.

The second aspect is the context in which we define the satisfaction relation. In separation logic, we employ another concept, next to structures and valuations, called the *heap*. A heap is represented by a partial function. In the standard semantics we furthermore restrict ourselves to *finitely-based* partial functions, meaning that only finitely many locations are assigned a value by the partial function representing the heap.

Let A = $(A; I)$ be a structure (see Definition A.2.3). A *finite heap* of A is a finitely-based partial function from A to A. A heap thus assigns to finitely many elements of the domain of the structure a *value*, which is also an element of the domain of the structure. Let h be a heap. By dom(h) we denote the *domain* of the heap h, that is, the set of all elements of A on which h is defined. If a is an element for which h is undefined we also write $h(a) = ?$ (where we implicitly know $? \notin A$ since $?$ is some dummy element), and if a is an element for which h is defined we write $h(a) = a'$ for some value $a' \in A$.

We may also speak of *locations* to mean elements that are (possibly) in the domain of a heap. A finite heap thus assigns finitely many locations to values. Note that speaking of just a domain may be unclear: is one speaking of the *domain of a structure*, or the *domain of a heap*? The latter, however, is a (finite) subset of the former.

We define three operations on finite heaps. There is the *empty heap*, denoted by ϵ. The empty heap is undefined on every element of the domain, that is, $\mathrm{dom}(\epsilon) = \emptyset$. Clearly the domain of the empty heap is finite. Given two elements $a, a' \in A$. By $h[a := a']$ we denote the heap obtained after applying a *heap update* operation that sets location a to the element a'. Formally,

$$h[a := a'](e) = \begin{cases} a' & \text{if } a = e \\ h(e) & \text{if } a \neq e \text{ and } h(e) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where e ranges over elements of the domain A of our structure. We now thus have $\mathrm{dom}(h[a := a']) = \mathrm{dom}(h) \cup \{a\}$, and so the domain remains finite. By $h[a := ?]$ we denote the heap obtained after applying a *heap clear* operation that clears location a from the domain. Formally,

$$h[a := ?](e) = \begin{cases} \text{undefined} & \text{if } a = e \\ h(e) & \text{if } a \neq e \text{ and } h(e) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and thus $\mathrm{dom}(h[a := ?]) = \mathrm{dom}(h) \cap \{a\}$, and also here the domain remains finite.

We have the property of *heap extensionality*: given two finite heaps h and g, then $h = g$ if and only if $\mathrm{dom}(h) = \mathrm{dom}(g)$ and $h(a) = g(a)$ for every $a \in \mathrm{dom}(h)$.

Intuitively, we could split and merge heaps. A finite heap that has more than one element in its domain can be partitioned into two finite heaps, by selecting for each element in the domain to what partition it should belong after the split. Similarly, given two finite heaps that have a disjoint domain, we can form a new

finite heap by merging the two. After splitting or merging, the values assigned to locations remain the same. To formalize these intuitions, we introduce the concept of a heap partitioning.

Let h_1 and h_2 be finite heaps with disjoint domains, $\mathrm{dom}(h_1) \setminus \mathrm{dom}(h_2) = ;$. We sometimes write $h_1 \mathbin{?} h_2$ to abbreviate $\mathrm{dom}(h_1) \setminus \mathrm{dom}(h_2) = ;$. Now $h_1 \mathbin{]} h_2$ denotes a finite heap that can be split into two parts, h_1 and h_2, so has as domain the union of the underlying domains, $\mathrm{dom}(h_1 \mathbin{]} h_2) = \mathrm{dom}(h_1) \mathbin{[} \mathrm{dom}(h_2)$. Every location in the resulting heap has the value of the corresponding underlying heap, $(h_1 \mathbin{]} h_2)(e) = h_1(e)$ if $e \in \mathrm{dom}(h_1)$ and $(h_1 \mathbin{]} h_2)(e) = h_2(e)$ if $e \in \mathrm{dom}(h_2)$. Locations outside of the domain remain undefined, $(h_1 \mathbin{]} h_2)(e) = ?$ if $e \notin \mathrm{dom}(h_1)$ and $e \notin \mathrm{dom}(h_2)$. Thus one can think of $h_1 \mathbin{]} h_2$ as a merged heap. It does not exists when h_1 and h_2 both assign a value to the same location, even when both h_1 and h_2 assign the same value to shared locations. Although the latter makes sense when merging heaps, its fails our intuition in the other direction, when splitting a heap in two parts. Thus, we write $h \quad h_1 \mathbin{]} h_2$ to mean $h_1 \mathbin{?} h_2$, that h_1 and h_2 have disjoint domains—and so the finite heap $h_1 \mathbin{]} h_2$ exists, and $h = h_1 \mathbin{]} h_2$, and we say that there is a heap partitioning.

We are now able to give the formal definition of the satisfaction relation. Our definition is an extension of Definition A.2.5 in two ways: we additionally consider a finite heap h, and we have new clauses corresponding to points-to, separating conjunction, and separating implication.

Definition 2.2.1 (Satisfaction relation). Given a structure $\mathsf{A} = (A; I)$, a valuation of A, a finite heap h of A, and a separation logic formula . The satisfaction relation $\mathsf{A}; h; \models^{\mathbf{SSL}}$ is defined inductively on the structure of :

- $\mathsf{A}; h; \models^{\mathbf{SSL}} ?$ never holds,
- $\mathsf{A}; h; \models^{\mathbf{SSL}} (x \doteq y)$ iff $(x) = (y)$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}} (x \mathbin{,\!\to} y)$ iff $h(\ (x))$ is defined and $h(\ (x)) = (y)$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}} C(x_1; \ldots; x_n)$ iff $(\ (x_1); \ldots; \ (x_n)) \in C^I$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}} \ \to$ iff $\mathsf{A}; h; \models^{\mathbf{SSL}}$ implies $\mathsf{A}; h; \models^{\mathbf{SSL}}$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}} \forall x$ iff $\mathsf{A}; h; [x := a] \models^{\mathbf{SSL}}$ for every $a \in A$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}}$ iff $\mathsf{A}; h_1; \models^{\mathbf{SSL}}$ and $\mathsf{A}; h_2; \models^{\mathbf{SSL}}$ for some $h_1; h_2$ such that $h \quad h_1 \mathbin{]} h_2$,
- $\mathsf{A}; h; \models^{\mathbf{SSL}}$ iff $\mathsf{A}; h'; \models^{\mathbf{SSL}}$ implies $\mathsf{A}; h''; \models^{\mathbf{SSL}}$ for every $h'; h''$ such that $h'' \quad h \mathbin{]} h'$.

The superscript SSL stands for Standard Separation Logic. In the third clause it is superfluous to state that $h(\ (x))$ is defined, since we know that (y) cannot be the dummy element $?$ and hence $(x) \in \mathrm{dom}(h)$. Further, since we restrict ourselves to first-order signatures, the valuation of A only assigns individual

variables a value. We shall leave out the discussion how the satisfaction relation is defined for empty structures, since it is similar to classical logic.

Based on this definition it is now also possible to give semantics of abbreviations, similar to what we did in the case of classical logic. Also similar to classical logic, we have the coincidence condition and invariance under renaming. Both propositions are with respect to a fixed heap.

Proposition 2.2.2 (Coincidence condition). *Given that* $[FV(\ )] = {}^{0}[FV(\ )]$, *it follows that* $\mathcal{A}; h; \ j=^{\mathbf{SSL}}$ *if and only if* $\mathcal{A}; h; {}^{0} j=^{\mathbf{SSL}}$.

Proposition 2.2.3 (Invariance under renaming). *Given a renaming* $\ $ *such that all free variables of* $\ $ *stay the same, i.e.* $(v) = v$ *for all* $v \in FV(\ )$. *It follows that* $\mathcal{A}; h; \ j=^{\mathbf{SSL}}$ *if and only if* $\mathcal{A}; h; \ j=^{\mathbf{SSL}} (\ )$.

Sometimes, it is more convenient to work with the set of heaps and valuations by which a formula is satisfied given a particular structure.

Definition 2.2.4 (Denotation). The *denotation* of a formula $\mathcal{A}J\ \mathbb{K}^{\mathbf{SSL}}$ is defined:

$$\mathcal{A}J\ \mathbb{K}^{\mathbf{SSL}} = \{(h; \ ) \mid \mathcal{A}; h; \ j=^{\mathbf{SSL}} \ \}.$$

Similar as before, we may drop SSL if clear from context. We write $\ _{\mathcal{A}}$ for $\mathcal{A}J\ \mathbb{K} = \mathcal{A}J\ \mathbb{K}$, and say that $\ $ and $\ $ are equivalent.

Note that the we can also add terms to separation logic, completely analogous to what we did in Section A.5. An important result also holds for separation logic:

Lemma 2.2.5 (Substitution lemma).

$$\mathcal{A}; h; \ j=^{\mathbf{SSL}} [x := t] \text{ if and only if } \mathcal{A}; h; \ [x := (t)] \ j=^{\mathbf{SSL}} \ :$$

We write $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ to mean $\mathcal{A}; h; \ j=^{\mathbf{SSL}}$ for all valuations $\ $, and we write $\mathcal{A} \ j=^{\mathbf{SSL}}$ to mean $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ for all finite heaps h. Given a sentence that is satisfied, using the coincidence condition we can obtain that it is also satisfied by the same structure but with any other valuation: the valuation has no influence on whether a sentence is satisfied by the structure, but the heap does have such influence. So if $\ $ is a sentence, $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ if and only if $\mathcal{A}; h; \ j=^{\mathbf{SSL}}$ for some valuation $\ $.

Given a sentence $\ $, we write $j=^{\mathbf{SSL}}$ to mean that $\mathcal{A} \ j=^{\mathbf{SSL}}$ for all structures $\mathcal{A}$, and we then say that $\ $ is *valid*. Valid sentences in separation logic thus are properties that hold for all finite heaps.

Given a theory, i.e. a set of sentences $\ $, we write $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ to mean that all sentences in $\ $ are satisfied by $\mathcal{A}$ and finite heap h, that is, $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ for all $\ \in \ $. We may also speak of '$\ $ is satisfied by $\mathcal{A}$ and h'. A theory $\ $ is *satisfiable* if there exists a structure $\mathcal{A}$ and heap h such that $\mathcal{A}; h \ j=^{\mathbf{SSL}}$. A theory $\ $ is *finitely satisfiable* if every finite subset of $\ $ is satisfiable. Note that the finite heap is considered existentially when speaking of (finite) satisfiability in separation logic.

We write $\ j=^{\mathbf{SSL}}$ to mean $\mathcal{A}; h \ j=^{\mathbf{SSL}}$ for all structures $\mathcal{A}$ and finite heaps h such that $\mathcal{A}; h \ j=^{\mathbf{SSL}}$, and say that $\ $ is a *semantic consequence* of $\ $.

By $Th^{\mathbf{SSL}}(\mathsf{A})$ we mean the set of all sentences such that $\mathsf{A} \models^{\mathbf{SSL}}$, and we speak of the *separation logic theory of* A. Note that we have $Th_1^{\mathbf{CL}}(\mathsf{A}) \quad Th^{\mathbf{SSL}}(\mathsf{A})$, that is, the first-order theory of A is included in its separation logic theory. Further, a separation logic theory contains only sentences that are universal in the heap, i.e. sentences that hold for every finite heap.

Proposition 2.2.6. *For any sentence we have* $\mathsf{A}; h \models^{\mathbf{SSL}}$ *or* $\mathsf{A}; h \models^{\mathbf{SSL}} \; : \; .$

However, contrasting to the first-order theory of a structure, which is necessarily complete, we do not have that the separation theory of a structure is necessarily complete. To see why, consider the counter-example used in the following proof.

Proposition 2.2.7. $Th^{\mathbf{SSL}}(\mathsf{A})$ *is complete if and only if the domain of* A *is empty.*

Proof. Assume the domain of A is not empty. It is sufficient to show there is a sentence such that there is a heap h_1 such that $\mathsf{A}; h_1 \models^{\mathbf{SSL}}$, and there is a heap h_2 such that $\mathsf{A}; h_2 \models^{\mathbf{SSL}} \; : \; .$ Take to be emp. Now h_1 can be simply , the empty heap. And h_2 is any non-empty heap (which exists since the domain of our structure is non-empty).

Assume $Th^{\mathbf{SSL}}(\mathsf{A})$ is not complete. Thus there is a sentence such that it is not the case that $\mathsf{A} \models^{\mathbf{SSL}}$ or $\mathsf{A} \models^{\mathbf{SSL}} \; : \; .$ So there exists $h_1; h_2$ such that $\mathsf{A}; h_1 \models^{\mathbf{SSL}} \; :$ and $\mathsf{A}; h_2 \models^{\mathbf{SSL}}$. Now h_1 and h_2 are not equal due to Proposition 2.2.6. If, however, A is empty then there is only one heap and h_1 and h_2 thus must be equal. So A must be non-empty. □

Another difference is that the standard semantics of separation logic is not compact, in contrast to classical logic (see Theorem A.2.10). So see why, consider the following counter-example: every finite subset of an infinite set of sentences expressing that the domain of the heap contains at least so many elements is satisfiable, but clearly no finite heap satisfies the entire set.

Lemma 2.2.8 (Non-compactness standard semantics). *It is not the case that is finitely satisfiable implies that is satisfiable.*

Proof. Let $x_0; x_1; x_2; \ldots$ be individual variables. We construct a set of sentences which is finitely satisfiable, but not satisfiable:

$$= \{ \ _n \mid n \in \mathbb{N} \}$$

where $_n$ expresses that there are at least $n + 1$ allocated and distinct elements:

$$_n = \exists x_0, \ldots, x_n . \left(\bigwedge_{0 \ i \ n} (x_i , ! \quad) \right) \wedge \bigwedge_{0 \ i<j \ n} x_i \not\doteq x_j .$$

Clearly, is infinite. For any finite subset of , there must exist a maximum m such that $_m \in$. Then we take a structure with as domain N and a finite heap in which the locations $0, \ldots, m$ are allocated (their value does not matter). Every formula $_i$ for $0 \quad i \quad m$ is satisfied. This construction works for every finite subset of , so is finitely satisfiable. However, is not satisfiable, since for every $_i \in$ there always exists $_j \in$ with $j > i$, and so we cannot construct a finite heap that satisfies all sentences. □

Above it is also possible to give $_n$ using separating conjunctions, namely by

$$_n = 9x_0; : : : ; x_n : (x_0 , ! \quad) \quad : : : \quad (x_n , ! \quad):$$

From the semantics of separating conjunction, it follows that x_i 6= x_j for every 0 $i < j$ n: if it were the case that $x_i = x_j$ for i 6= j then one cannot split the heap into disjoint parts that satisfies all components of the conjunction.

The failure of compactness has important ramifications to the design of a proof system for separation logic.

Corollary 2.2.9. *There is no sound, complete, finitary proof system for* SSL.

Proof. Suppose there would be a finitary proof system SSL that allows us to define what it means that a sentence is a syntactic consequence of a theory , denoted $`^{\mathbf{SSL}}$, and suppose that it is complete with respect to the standard semantics:

$$j=^{\mathbf{SSL}} \quad \text{implies} \quad `^{\mathbf{SSL}} :$$

Now we have that compactness follows from it: is finitely satisfiable if and only if is satisfiable. To see why, it is su□cient to show that if is not satisfiable then is not finitely satisfiable. Suppose some theory is not satisfiable, then $j=^{\mathbf{SSL}} ?$ is the case. By completeness, we then have $`^{\mathbf{SSL}} ?$. By the finitary nature of our proof system, there must only be finitely many sentences in on which the deduction is based. Let $_0$ denote that finite subset of such that the same deduction can be used to witness $_0 `^{\mathbf{SSL}} ?$. Thus is not finitely satisfiable. Hence, the existence of a complete finitary proof system is in contradiction with the above non-compactness proposition. □

2.3 Full semantics

One cause of non-compactness in the previous section is the assumption that we deal with finite heaps only. In this section, we consider a more liberal semantics: the *full semantics* of separation logic. In the full semantics, we leave out the restriction that we only consider heaps that are finite. Recall the discussion in the introduction chapter that motivates our choice (see Section 1.3). Does this modification resolve the problem of non-compactness, or will the resulting semantics also be non-compact? That is the main question we answer in this section.

Again, let A = $(A; I)$ be a structure. A *heap* of A is a partial function from A to A. Every finite heap is a heap, and also every function from A to A is a heap. If A is infinite, then there are heaps that are not finite heaps. In the case A is finite, then every heap is a finite heap. A heap that is not a finite heap is called an infinite heap.

The concepts we have introduced earlier are easily adapted to the new situation. Let h be a (finite or infinite) heap. The domain dom(h) is the set of locations for which h is defined. For infinite heaps, dom(h) is an infinite set. The empty heap remains. The two operations of heap update $h[a := a']$ and heap clear $h[a := ?]$ can be extended to infinite heaps: their definition remains the same. However,

dom($h[a := a^0]$) and dom($h[a := ?]$) remain infinite if dom(h) is infinite. Also the concept of heap partitioning can be extended to infinite heaps: we have that either h_1 or h_2 is an infinite heap if $h \quad h_1 \;] \; h_2$ and h is an infinite heap.

Similarly, we can adapt the satisfaction relation, which for the full semantics we denote by $A; h; \; j=^{\mathbf{FSL}}$, where the superscript FSL stands for Full Separation Logic. Revisiting Definition 2.2.1, we now have the following adapted clauses:

Definition 2.3.1 (Satisfaction relation). Given a structure $A = (A; I)$, a valuation of A, a heap h of A, and a separation logic formula . The satisfaction relation $A; h; \; j=^{\mathbf{FSL}}$ is defined inductively on the structure of :

- $: : :$
- $A; h; \; j=^{\mathbf{FSL}}$ i $A; h_1; \; j=^{\mathbf{FSL}}$ and $A; h_2; \; j=^{\mathbf{FSL}}$ for some $h_1; h_2$ such that $h \quad h_1 \;] \; h_2$,
- $A; h; \; j=^{\mathbf{FSL}}$ i $A; h^0; \; j=^{\mathbf{FSL}}$ implies $A; h^{00}; \; j=^{\mathbf{FSL}}$ for every $h^0; h^{00}$ such that $h^{00} \quad h \;] \; h^0$.

where the heaps $h; h_1; h_2; h^0; h^{00}$ range over (finite or infinite) heaps, not only finite heaps as in the standard semantics.

With this satisfaction relation, we can also introduce the usual no(ta)tions of validity, semantic consequence, and theories, but we have to make sure that we consider all, finite or infinite, heaps universally. For example, we write $A \; j=^{\mathbf{FSL}}$ to mean $A; h \; j=^{\mathbf{FSL}}$ for all (finite or infinite) heaps h.

Comparing the standard semantics and the full semantics with respect to the satisfaction relation, we can see some obvious connections. For structures, the full semantics also has the first-order theory included in its separation theory: we have $Th_1^{\mathbf{CL}}(A) \quad Th^{\mathbf{FSL}}(A)$. If the domain of our structure is finite, every heap is also a finite heap: so there is no distinction between the two semantics. However, there are structures with an infinite domain in which the full semantics and standard semantics di er in the sentences they satisfy.

Consider the sentence $= 9x:(x \; , \; ! \quad) \quad ?$, and take a structure N with the naturals N as domain. The sentence is considered false with respect to the standard semantics, but the sentence is satisfiable with respect to the full semantics.

- (SSL) Let h be an arbitrary finite heap. Let m be the maximum location of h, or 0 if h is empty. Then surely $[m + 1 := 0]$ is disjoint from h. However, $N; h[m + 1 := 0] \not\models^{\mathbf{SSL}} ?$.
- (FSL) To show satisfiability, we give a heap h such that $N; h \; j=^{\mathbf{FSL}}$. Take any function for h such that dom(h) = N, i.e. all locations are allocated. Now there is no disjoint h^0 that is not empty. Hence, for any choice of value for x, the formula $(x \; , \; ! \quad) \quad ?$ is vacuously satisfied.

To further explore the semantics of separation logic, we introduce the following abbreviation (also called the *box modality*) that we characterize below:

abbreviates > (emp ^ (>))

Formulas placed directly within the context of the box modality are interpreted with respect to an arbitrary heap. Thus, this modality expresses a limited form of second-order universal quantification. One may think of the box modality acting as a binder of the points-to construct.

Proposition 2.3.2. *The following holds:*

- $\mathsf{A}; h; \; \models^{\mathbf{SSL}}$ *if and only if* $\mathsf{A}; h'; \; \models^{\mathbf{SSL}}$ *for every finite heap* h',
- $\mathsf{A}; h; \; \models^{\mathbf{FSL}}$ *if and only if* $\mathsf{A}; h'; \; \models^{\mathbf{FSL}}$ *for every heap* h'.

Proof. We show it for the standard semantics first.
$\mathsf{A}; h; \; \models^{\mathbf{SSL}} > \quad (\mathsf{emp} \wedge (> \quad))$
if and only if
$\mathsf{A}; \; ; \; \models^{\mathbf{SSL}} >$
if and only if
$\mathsf{A}; h'; \; \models^{\mathbf{SSL}}$ for every finite heap h'.
The proof for the full semantics is similar, but quantifying over all heaps h'. □

Dually, we introduce the *diamond modality*

abbreviates $: \quad :$

which expresses a limited form of second-order existential quantification.

Corollary 2.3.3. *The following holds:*

- $\mathsf{A}; h; \; \models^{\mathbf{SSL}}$ *if and only if* $\mathsf{A}; h'; \; \models^{\mathbf{SSL}}$ *for some finite heap* h',
- $\mathsf{A}; h; \; \models^{\mathbf{FSL}}$ *if and only if* $\mathsf{A}; h'; \; \models^{\mathbf{FSL}}$ *for some heap* h'.

The above box modality can be used to characterize finiteness of the domain of the structure in the case of full separation logic. Let n abbreviate

$$(\mathit{tot}(, \; ! \;) \wedge \mathit{inj}(, \; ! \;) \; ! \; \mathit{surj}(, \; ! \;))$$

where the abbreviations tot, inj, surj of Proposition A.2.13 can be reused, but applied to points-to by considering the primitive separation logic formula $(x \; , ! \; y)$ to be obtained as if $, ! \;$ were a 2-ary relation (cf. the abbreviations below).

Proposition 2.3.4. $\mathsf{A} \models^{\mathbf{FSL}} n$ *if and only if the domain of* A *is finite.*

Proof. Recall Proposition A.2.12, that a set is finite if and only if every injective total function on that set is a surjection. In contrast to Proposition A.2.13, we do not need $\mathit{fun}(, \; ! \;)$ because in the semantics of separation logic we already have that heaps are partial functions from which this property holds for every heap. □

The above characterization fails for standard separation logic. In that semantics, the formula is valid: if the domain of the structure is finite then every injective total heap is already a surjection, and if the domain of the structure is infinite then there is no finite heap that satisfies $\mathit{tot}(, \; ! \;)$.

Going further, we can give a sufficient condition for the finiteness of the domain of the heap. Surely, for the standard semantics, this condition is useless since every heap already is finite. However, in the full semantics, the condition can be useful in certain contexts to restrict attention to heaps with a finite domain. We extend our signature with an additional 1-ary predicate symbol, P. We then have the following sentence:

$$\Box\forall x.\, P(x) \;\dot{\leftrightarrow}\; (x, !\;\;) \tag{2.1}$$

which expresses that the extension of the predicate P coincides with the domain of the current interpretation of the heap. This sentence can not be valid if the structure has at least one element in its domain, since each structure gives an *a priori* interpretation of P independently of the heap, which may satisfy the formula for one heap (say, the empty heap) but not for another (say, the non-empty heap).

However, we are still able to use the formula to capture the domain of the heap at top-level. How this can be used will become clear later. To express that the extension of P is finite we adapt our previous formula in the following way:

$$(tot^P(, !\,) \wedge inj^P(, !\,) \;\Rightarrow\; surj^P(, !\,)) \tag{2.2}$$

where we introduce the following relativized abbreviations:

- $inj^P(, !\,)$ abbreviates $\forall x,y,z.\, P(x) \wedge P(y) \wedge P(z) \wedge (x, !\; z) \wedge (y, !\; z) \Rightarrow x \doteq y$,
- $tot^P(, !\,)$ abbreviates $\forall x.\, P(x) \Rightarrow \exists y.\, P(y) \wedge (x, !\; y)$,
- $surj^P(, !\,)$ abbreviates $\forall y.\, P(y) \Rightarrow \exists x.\, P(x) \wedge (x, !\; y)$.

Even in case the domain of our structure is infinite and in the full semantics the box modality universally quantifies over all heaps, the above formula expresses that all total injective functions on P must be surjective on P, and thus that P is finite (for the same reason as in Proposition A.2.12).

Given that we can capture the domain of the heap at the top-level by a predicate P, and that we can express that the extension of P is finite, we have also non-compactness for the full semantics.

Lemma 2.3.5 (Non-compactness full semantics). *It is not the case that $\Box$ is finitely satisfiable implies that $\Box$ is satisfiable.*

Proof. The set of sentences $\Box$ is finitely satisfiable but not satisfiable:

$$\Box = \{\, \Box_n \mid n \geq 2 \,\} \cup \{\, \Box \,\}$$

where $\Box_n$ expresses that there are at least $n+1$ allocated and distinct elements as in Lemma 2.2.8, and $\Box$ is the conjunction of the sentences of Equations (2.1) and (2.2). Each finite subset of $\Box$ is satisfiable: we can construct structures in the same way as before, but now also select as interpretation for P the domain of the finite heap used. However, $\Box$ is not satisfiable: due to $\Box$ we need to choose a finite heap h and then not all $\Box_n$ can be satisfied. □

Corollary 2.3.6. *There is no sound, complete, finitary proof system for* FSL.

The above argument of non-compactness relies on the presence of separating implication (in the box modality), whereas in Lemma 2.2.8 we need not rely on any separating connective. Can we also show non-compactness without relying on separating implication?

We introduce the following abbreviation (also called the *sub-heap modality*):

$$\text{abbreviates } :(> \quad : \ )$$

that expresses that holds for every sub-heap of the current heap. Bannister and others have also introduced a related connective, called *separating coimplication*, in [15], of which this sub-heap modality is an instance. Formally, a heap h^0 is a sub-heap of heap h, denoted $h^0 \quad h$, if $\mathrm{dom}(h^0) \quad \mathrm{dom}(h)$ and $h^0(a) = h(a)$ for all a 2 $\mathrm{dom}(h^0)$. We have that $h \quad h_1 \ ] \ h_2$ implies $h_1 \quad h$ and $h_2 \quad h$. Conversely, if $h_2 \quad h$ then there exists a heap h_1 such that $h \quad h_1 \ ] \ h_2$.

Proposition 2.3.7. *The following holds:*

- $A; h; \ j=^{\mathbf{SSL}}$ *if and only if* $A; h^0; \ j=^{\mathbf{SSL}}$ *for every finite heap* $h^0 \quad h$,
- $A; h; \ j=^{\mathbf{FSL}}$ *if and only if* $A; h^0; \ j=^{\mathbf{FSL}}$ *for every heap* $h^0 \quad h$.

Proof. We show it for the standard semantics first.
$A; h; \ j=^{\mathbf{SSL}} :(> \quad : \ )$
if and only if
$A; h_1; \ j=^{\mathbf{SSL}} ?$ or $A; h_2; \ j=^{\mathbf{SSL}}$ for all $h_1; h_2$ such that $h \quad h_1 \ ] \ h_2$
if and only if
$A; h^0; \ j=^{\mathbf{SSL}}$ for every $h^0 \quad h$.
The proof for the full semantics is similar. □

Since a heap is a partial function on the domain of a structure, the values that are assigned to locations can themselves be used as locations. Intuitively, if $(x \ , ! \ y)$ and $(y \ , ! \ z)$ are satisfied, we may think of traversing the pointer at location x to obtain the value y, which is used as a location and can be traversed to obtain the value z. Such traversal is denoted by $(x \ , ! \ y \ , ! \ z)$. In general, we can form a *chain* of traversals

$$(x_0 \ , ! \ x_1 \ , ! \ ::: \ , ! \ x_n)$$

which means $(x_i \ , ! \ x_{i+1})$ for every $0 \quad i < n$. We traverse n locations to end up with the value x_n. We say one can *reach* value x_n by traversing n locations. A heap with cycles allows the same location to be revisited in a traversal:

$$x_0 \ , ! \ ::: \ , ! \ x_n \ , ! \ x_0$$

where we end up back at the starting location x_0 and we can infinitely continue traversing along the same locations of the chain. Although the cycle comprises finitely many locations, the chain can be extended infinitely long.

A *dead-end* is a location that is not allocated: we are unable to continue the traversal through that location. Formally, x is a dead-end if $8y(x \ ,6! \ y)$ (or, equivalently, $(x \ ,6! \quad)$). If a location is not a dead-end, we may progress our

traversal from that location. Conversely, an *unreachable* value (in the sense that it is unreachable from the heap) is a value which is not pointed to by any location allocated on the heap. Formally, x is unreachable if $\forall y(y \not\mapsto x)$. A value is reachable (from the heap) if it is not unreachable, so there exists a location which points to that value. One can think of the set of all values that are reachable from the heap as comprising the contents of the heap. Thus, a value unreachable from some heap is a value that is not contained in that heap.

In both the standard semantics and the full semantics, there is a heap in which every location is a dead-end: the empty heap. The empty heap has no contents. In the full semantics, we can also have heaps in which all values of the domain of the structure are reachable. Then the contents of the heap is the same as the domain of the structure. This is not possible in the standard semantics for structures with an infinite domain.

A heap h is *well-founded* if for every non-empty sub-heap $h' \subseteq h$ there is a value not contained in h'. Alternatively, a heap h is well-founded if there exists no infinite sequence of locations $a_0, a_1, \ldots$ such that $h(a_{n+1}) = a_n$. There are non-well-founded heaps, for example, a heap which has a cycle.

With the sub modality we can express the heap is *well-founded*, by the sentence

$$(\mathsf{emp} \mathbin{\_} \exists x((x \hookrightarrow) \wedge \forall y((y \hookrightarrow) \rightarrow (y \not\mapsto x)))). \tag{2.3}$$

Lemma 2.3.8 (Non-compactness full semantics). *It is not the case that is finitely satisfiable implies is satisfiable for theories in which the separating implication connective does not occur.*

Proof. Let $c_0, c_1, \ldots$ be countably many individual constant symbols (these can be encoded using unary predicate symbols with the appropriate property of existence and uniqueness). The set of sentences is finitely satisfiable but not satisfiable:

$$= \{c_{n+1} \hookrightarrow c_n \mid n \quad 0\} \cup \{ \}$$

where is Equation (2.3). Every finite subset of is satisfiable: take as domain of our structure N, interpret the individual constants $c_0, \ldots$ by unique naturals, and construct a finite heap that satisfies the (finitely many) points-to constraints. For every sub-heap of the constructed finite heap there exists a value not contained in it, so is also satisfied. However, is not satisfiable, as that would imply there is an infinite sequence of locations $a_0, a_1, \ldots$ such that $h(a_{n+1}) = a_n$. □

Note that we did not need to add to any sentence expressing that $c_i \not\approx c_j$ for $i \not\approx j$, since this possibility is already ruled out by : if $c_i = c_j$ for some $i \not\approx j$ then there is a cycle in the heap and thus is the heap non-well-founded.

Corollary 2.3.9. *There is no sound, complete, finitary proof system restricted to formulas without separating implication for* FSL.

The diamond and sub modality together can be used to express that the domain of the structure is countably infinite. The main idea is that we can express that there is a smallest heap that has the same structure as the natural numbers,

and that every element of the domain is reachable in that heap. For notational convenience, we introduce the following abbreviations: let *zero* abbreviate the sentence

$$9x((x\ ,!\ \ )\ \wedge\ 8y(y\ ,\!\circledast\ x))$$

which expresses that there is some unreachable value in the domain of the heap, and let *nat* abbreviate the sentence

$$tot(,!\ )\ \wedge\ inj(,!\ )\ \wedge\ zero:$$

If $A;h \models^{\mathbf{FSL}} nat$ then h encodes some copy of the natural numbers. We can then define $f : \mathbb{N} \to A$ inductively by $f(0) = a_0$, where

$$A;h;\ [x := a_0] \models^{\mathbf{FSL}} 8y((x\ ,!\ \ )\ \wedge\ y\ ,\!\circledast\ x))$$

for some valuation (i.e. a_0 is some 'minimal' element of h), and $f(n+1) = a'$, where $h(f(n)) = a'$. The latter is defined on $f(n)$ because of $tot(,!\ )$, since $A;h \models^{\mathbf{FSL}} 8x9y(x\ ,!\ y)$ means that the domain of h equals D. Note that the encoding is not necessarily unique: there could be multiple 'minimal' elements.

Next, let *ind* we denote the formula

$$(zero\ \wedge\ (8x;y:\ (x\ ,!\ y)\ \to\ (y\ ,!\ \ )))\ \to\ 8x(x\ ,!\ \ )$$

which expresses that the domain of the heap and the structure coincide, if there is some unreachable value in the domain of the heap and every reachable value is also in the domain of the heap. The formula $(nat\ \wedge\ ind)$ then characterizes the class of countably infinite structures.

Proposition 2.3.10. *$A \models^{\mathbf{FSL}} (nat\ \wedge\ ind)$ if and only if the domain of A is countably infinite.*

Proof. Let $A \models^{\mathbf{FSL}} (nat\ \wedge\ ind)$. From Corollary 2.3.3, there exists a heap h' such that $A;h' \models^{\mathbf{FSL}} nat\ \wedge\ ind$. From the definition of *nat*, as explained above, it follows that h' contains a copy of the natural numbers. From Proposition 2.3.7 and that *ind* is satisfied, it follows that the sub-heap h'' of h' satisfies *ind*. Now choose a particular sub-heap h''_0 of h' which contains precisely the copy of the natural numbers, and nothing else. Thus h''_0 satisfies $zero\ \wedge\ 8x;y((x\ ,!\ y)\ \to\ (y\ ,!\ \ ))$, and so we obtain that $A;h''_0 \models^{\mathbf{FSL}} 8x(x\ ,!\ \ )$, that is, the domain of h''_0 (which is $\mathbb{N}$) equals the domain of A.

Conversely, let A be countably infinite. For any enumeration of the domain of A, we can construct a corresponding heap h which encodes this enumeration and thus satisfies $A;h \models^{\mathbf{FSL}} nat\ \wedge\ ind$. □

Corollary 2.3.11. *$A \models^{\mathbf{FSL}} :\ n\ \wedge\ (nat\ !\ :\ ind)$ if and only if the domain of A is uncountably infinite.*

Thus, we are able to distinguish whether a structure is countably infinite or not. Consequently, full separation logic cannot satisfy the Löwenheim-Skolem theorem.

2.4 Embeddings

In the previous sections we defined the standard semantics and full semantics for separation logic. But what is the precise relation between the validity $\models^{\mathbf{FSL}}$ and $\models^{\mathbf{SSL}}$ for formulas of separation logic? It seems possible to embed the valid sentences of standard separation logic in full separation logic, if there is a formula nh that expresses the finiteness of the domain of the (current) heap in full separation logic. Consider the following translation $T(\ )$ defined by induction on the separation logic formula :

- $T(?) = ?$,
- $T(x \doteq y) = (x \doteq y)$,
- $T(x \mapsto y) = (x \mapsto y)$,
- $T(C(x_1, \ldots, x_n)) = C(x_1, \ldots, x_n)$,
- $T(\ \ -\!\!* \ \ ) = T(\ ) -\!\!* T(\ )$,
- $T(\forall x \ ) = \forall x(T(\ ))$,
- $T(\ \ \ ) = T(\ ) \ \ T(\ )$,
- $T(\ \ \ ) = (\ nh \wedge T(\ )) \ \ T(\ )$.

Proposition 2.4.1. $A, h, \models^{\mathbf{SSL}}$ *if and only if* $A, h, \models^{\mathbf{FSL}} nh \wedge T(\ )$.

Proof. Note that h necessarily is a finite heap. A finite heap can only be split into finite subheaps. In the translation of separating implication we ensure that in the full semantics we only quantify over finite heaps. □

Recall that we have already seen a *sufficient* condition for finiteness of the domain of the heap, the conjunction of Equations (2.1) and (2.2). However, that condition only works at the top-level and makes use of an additional predicate symbol added to the signature that can only be given an interpretation *a priori*, and not depending on the current interpretation of the heap as modified by the separating connectives. However, we still lack a *necessary* condition and thus have the following open problem:

Problem 2.1. *Is there a sentence in separation logic* nh *such that* $A, h \models^{\mathbf{FSL}} nh$ *if and only if* $\mathrm{dom}(h)$ *is finite?*

Next, we then turn our attention to the relation between separation logic and classical logic. Firstly, we embed a subset of separation logic, called *separation logic light*, into first-order logic. We argue that the resulting embedding preserves semantics, but does not give us a compact semantic consequence relation. Secondly, we show that it is possible to translate a formula of separation logic into a formula of second-order logic that preserves its semantics.

In the previous section, we have seen there is no sound, complete, finitary proof system for full separation logic, even when we restrict to formulas without

separating implication. Was the problem of non-compactness caused by the fact that, in the modality , that is defined as $\neg(\top \; \neg)$, we use negation outside of separating conjunction? We now study a restricted set of separation logic formulas, in which we restrict the occurrences of the separating conjunction. Are we able to show compactness whenever we disallow using separating conjunction under negation?

Recall that in a pure formula of separation logic we do not have any occurrences of points-to, separating conjunction, or separating implication. We now consider *semi-pure* formulas: in a semi-pure formulas there are no occurrences of separating conjunction or separating implication, but we are allowed to use points-to. We then define a restricted set of formulas of separation logic, called the formulas of separating logic light (SLL), as follows:

- a semi-pure formula of separation logic is a formula of separation logic light,
- ( ) is a formula of separation logic light given that and are formulas of separation logic light.

We thus restrict the use of separating conjunction to the 'top level' of the formula. For the purposes of our exposition, we use only the classical connectives of conjunction, disjunction and negation: in this case, implication ( ) is interpreted as material implication ($\neg \vee$). Note that the use of negation in a semi-pure formula can be pushed to the leaves of the formula, by the classical equivalences:

- $\neg(\forall x \;)$ reduces to $\exists x(\neg \;)$,
- $\neg(\exists x \;)$ reduces to $\forall x(\neg \;)$,
- $\neg(\; \wedge \;)$ reduces to $(\neg \;) \vee (\neg \;)$,
- $\neg(\; \vee \;)$ reduces to $(\neg \;) \wedge (\neg \;)$.

In fact, any semi-pure formula can first be normalized into prenex normal form, and then the negation in the quantifier-free part can be pushed to the leaves. Such formulas of separation logic light are said to be in *normal form*.

We now consider whether satisfiability of separation logic light formulas is compact, that is, whether a theory of separation logic light is satisfiable if and only if that theory is finitely satisfiable.

Proposition 2.4.2. *Given a set of separation logic light formulas. There exists a structure* A *and heap h such that* $A; h \models^{\mathbf{FSL}}$ *, if and only if, for every finite subset* $_0$ *there exists a structure* A *and heap h such that* $A; h \models^{\mathbf{FSL}} {}_0$.

Proof. We introduce the following first-order translation $@R$ of separation logic light formulas in normal form, where R is a binary relation symbol of the signature (so necessarily different from ,):

- $(x \doteq y)@R = (x \doteq y)$,
- $(x \mapsto y)@R = R(x; y)$,

- $C(x_1,\ldots,x_n)@R = C(x_1,\ldots,x_n)$,
- $(\ :\ )@R = \ :(\ @R)$,
- $(\ \wedge\ )@R = \ @R \wedge \ @R$ and $(\ \_\ )@R = \ @R \_ \ @R$,
- $(8x\ )@R = 8x(\ @R)$ and $(9x\ )@R = 9x(\ @R)$,
- $(\quad)@R = R \quad R_1\ ]\ R_2 \wedge \ @R_1 \wedge \ @R_2$,

where by $R \quad R_1\ ]\ R_2$ we denote the formula

$$8x;y:(R(x;y)\ \$\ R_1(x;y) \_ R_2(x;y)) \wedge :(R_1(x;y) \wedge R_2(x;y))$$

that expresses that R is the union of R_1 and R_1 and that R_1 and R_2 are disjoint. The binary relation symbols R, R_1 and R_2 are 'fresh'. It is sufficient to show that is satisfiable (in full separation logic) if and only if $fun(R) \wedge @R$ is satisfiable (in classical logic). More precisely, $A;h;\ \models^{\mathbf{FSL}}$ for some $A;h;$ if and only $B;\ ^0 \models^{\mathbf{CL}} fun(R) \wedge @R$ for some $B;\ ^0$. Note that the interpretation of R is the graph of the heap h.

Consequently, compactness of first-order logic implies compactness of separation logic light. Let be an infinite set of formulas of SLL and construct a corresponding set $^0 = \{fun(R) \wedge @R \mid \ 2\ \ g$ for some fixed binary relation symbol R. Note that ^{0} may require the introduction of a countably infinite number of fresh binary relation symbols. This is however no problem because first-order logic allows for the addition of a countably infinite set of relation symbols to the signature without affecting satisfiability. Now, given that every finite subset of is satisfiable, so is every finite subset of ^{0}. By the compactness of first-order logic, we then have that ^{0} is also satisfiable. From the structure witnessing satisfiability of ^{0} we can show the satisfiability of in full separation logic. Conversely, assume that is satisfiable in full separation logic, then we can use the same structure as witness for the satisfiability of every finite subset of . □

A similar result can be obtained for standard separation logic.

Although we are able to show compactness of the satisfiability relation, we have not yet reached the conclusion that the semantics can be useful for an adequate proof theory. This is because compactness of the satisfiability relation does not imply that the semantic consequence relation is compact. Non-compactness of the consequence relation for separation logic light follows directly from the above argument involving well-founded relations.

Lemma 2.4.3. *Given a set of separation logic light formulas, and a formula of separation logic light. There is a counter-example to the claim:* $\models^{\mathbf{FSL}}$ *implies there exists a finite subset* $_0$ *such that* $_0 \models^{\mathbf{FSL}}$.

Proof. Let denote the set of separation logic light formulas

$$\{(c_{n+1},!\ c_n) \mid n\ \ 0\}$$

where $c_0; c_1; \ldots$ are individual constant symbols (these can be again encoded). It follows that

$$\models^{\mathbf{FSL}} \quad (\neg \mathsf{emp} \wedge \forall x((x \mapsto \quad) \rightarrow \exists y(y \mapsto x)));$$

where also the latter is a separation logic light formula. The formula expresses that there exists some non-empty sub-heap, in which every location in the domain is reachable. Clearly, this is the case if we have a cycle in the heap. However, this is also the case when we have an infinite chain in the heap. But, importantly, there does not exist a finite subset $_0$ of such that

$$_0 \models^{\mathbf{FSL}} \quad (\neg \mathsf{emp} \wedge \forall x((x \mapsto \quad) \rightarrow \exists y(y \mapsto x))).$$

As soon as we restrict ourselves to a finite subset $_0$ of , we know we only have to interpret finitely many values for c_i. In that case, we no longer necessarily have a non-empty subheap in which every location in the domain is reachable. □

This failure of compactness of the semantic consequence relation is important for the design of a finitary proof system (see also Theorem A.4.3): it is not possible to have a sound and complete, finitary proof system for FSL even when we restrict to the formulas of separation logic light.

We now turn to the relation between separation logic and second-order classical logic. In particular, we focus on dyadic second-order logic, where we restrict second-order quantification to variables of arity 2. It is possible to embed separation logic in dyadic second-order logic, meaning that second-order logic is at least as expressive as separation logic.

Given a first-order signature . We define the following translation of formulas of separation logic into formulas of second-order logic. The translation is parameterized by a higher-order variable of arity 2, which intuitively takes the place of the points-to construct. This translation makes obvious how separating conjunction and separating implication can be read as particular second-order quantifications.

Definition 2.4.4. Let $R; R_1; R_2; R'; R''$ be variables of arity 2. The function $[\ ](R)$ translates formulas of separation logic to formulas of dyadic second-order logic, and is defined inductively as follows:

- $[?](R) = ?$,
- $[x \doteq y](R) = (x \doteq y)$,
- $[x \mapsto y](R) = R(x; y)$,
- $[C(x_1; \ldots; x_n)](R) = C(x_1; \ldots; x_n)$,
- $[\quad \rightarrow \quad](R) = [\ ](R) \rightarrow [\ ](R)$,
- $[\forall x\ ](R) = \forall x([\ ](R))$,
- $[\qquad](R) = \exists R_1 \exists R_2 (R \quad R_1 \rangle R_2 \wedge [\ ](R_1) \wedge [\ ](R_2))$,

- $[\quad](R) = \forall R'' \forall R'(\mathit{fun}(R'') \wedge R'' \quad R' \rangle R \rightarrow [\ ](R') \rightarrow [\ ](R''))$,

where $R \quad R_1 \rangle R_2$ denotes the formula

$$\forall x, y.\ (R(x,y) \Leftrightarrow R_1(x,y) \vee R_2(x,y)) \wedge \neg(R_1(x,y) \wedge R_2(x,y))$$

where R_1 and R_2 are variables of arity 2.

Note that, in second-order logic, we have the following properties:

- $\mathit{fun}(R) \wedge R \quad R_1 \rangle R_2 \rightarrow \mathit{fun}(R_1) \wedge \mathit{fun}(R_2)$,
- $\mathit{fun}(R) \wedge \mathit{fun}(R'') \wedge R'' \quad R' \rangle R \rightarrow \mathit{fun}(R')$,

hence we need not explicitly mention that all quantified variables are functional.

Proposition 2.4.5. *Given structure* $\mathsf{A} = (A, I)$ *and formula* *of separation logic. We have that* $\mathsf{A} \models^{\mathbf{FSL}}$ *if and only if* $\mathsf{A} \models^{\mathbf{CL}} \forall R(\mathit{fun}(R) \rightarrow [\ ](R))$.

Proof. By unfolding the semantics of separation logic, and induction on the structure of the formula , where the parameter R represents the current heap in the semantics of separation logic. □

2.5 Relational separation logic

We now ask ourselves the question: is separation logic also at least as expressive as dyadic second-order logic? Although this question is still open for full separation logic at the point of this writing, we can formulate a sufficient condition for having that separation logic is at least as expressive as dyadic second-order logic. That condition is the expressivity of the *binding operator* which captures the current interpretation of the heap in a second-order variable. However, before we are able to introduce the binding operator, we need to discuss two differences between separation logic and dyadic second-order logic.

The first difference between separation logic and dyadic second-order logic is that in our presentation, separation logic speaks about heaps as being *partial functions* whereas in dyadic second-order logic the second-order variables denote *binary relations*. We could thus generalize the semantics of separation logic, and remove the restriction to partial functions at the semantic level; instead consider a separation logic of binary relations called *relational separation logic*. In relational separation logic, we treat the primitive points-to $(x \hookrightarrow y)$ as a binary relation that is *not necessarily* functional, in contrast to the semantics of separation logic as presented in the previous chapter.

Let A be a set (say, the domain of a structure). A relation R is a subset of the Cartesian product $A \quad A$, that is, $R \quad A \quad A$. This is in contrast to heaps, which are partial functions from A to A. Every heap can be seen as a relation, where we take the graph of the partial function. We introduce the following notions on relations. The domain of a relation $\mathrm{dom}(R)$ is the set $\{a \mid (a, a') \in R \text{ for some } a'\}$. By $R \# R'$ we denote that the domains of the relations R and R' are disjoint. A

relation R is functional if for every element $a \in \mathrm{dom}(R)$ there is a unique a' such that $(a; a') \in R$. If R and R' are functional, $R \setminus R' = \emptyset$ if and only if $R \perp R'$.

Similar to our previous discussion regarding the semantics of *standard* and *full* separation logic, we can introduce the following relational separation logic semantics:

- WRSL is *weak relational separation logic* where second-order quantification of the separating connectives ranges over *all finite* binary relations,
- FRSL is *full relational separation logic* where second-order quantification of the separating connectives ranges over *all* binary relations.

We can now give the definitions of weak relational separation logic, WRSL, and full relational separation logic, FRSL. Only the definition for FRSL is shown in full below, that of WRSL is easily obtained by restricting to finite relations.

Definition 2.5.1 (Satisfaction relation). Given a structure $\mathcal{A} = (A; I)$, a valuation ν of $\mathcal{A}$, a finite binary relation $R \subseteq A \times A$, and a separation logic formula φ. The satisfaction relation $\mathcal{A}, R, \nu \models^{\mathbf{WRSL}} \varphi$ is defined inductively on φ:

- $\varphi ::= \ldots$
- $\mathcal{A}, R, \nu \models^{\mathbf{WRSL}} \varphi$ iff $\mathcal{A}, R_1, \nu \models^{\mathbf{WRSL}} \varphi$ and $\mathcal{A}, R_2, \nu \models^{\mathbf{WRSL}} \varphi$ for some finite $R_1, R_2 \subseteq A \times A$ such that $R = R_1 \cup R_2$ and $R_1 \perp R_2$,
- $\mathcal{A}, R, \nu \models^{\mathbf{WRSL}} \varphi$ iff $\mathcal{A}, R', \nu \models^{\mathbf{WRSL}} \varphi$ implies $\mathcal{A}, R \cup R', \nu \models^{\mathbf{WRSL}} \varphi$ for every finite $R' \subseteq A \times A$ such that $R \perp R'$.

Definition 2.5.2 (Satisfaction relation). Given a structure $\mathcal{A} = (A; I)$, a valuation ν of $\mathcal{A}$, a binary relation $R \subseteq A \times A$, and a separation logic formula φ. The satisfaction relation $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \varphi$ is defined inductively on the structure of φ:

- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \varphi$ never holds,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} (x \doteq y)$ iff $\nu(x) = \nu(y)$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} (x \mapsto y)$ iff $(\nu(x), \nu(y)) \in R$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} C(x_1, \ldots, x_n)$ iff $(\nu(x_1), \ldots, \nu(x_n)) \in C^I$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \neg\varphi$ iff $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \varphi$ implies $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \bot$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \forall x$ iff $\mathcal{A}, R, [x := a]\nu \models^{\mathbf{FRSL}} \varphi$ for every $a \in A$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \varphi * \psi$ iff $\mathcal{A}, R_1, \nu \models^{\mathbf{FRSL}} \varphi$ and $\mathcal{A}, R_2, \nu \models^{\mathbf{FRSL}} \psi$ for some $R_1, R_2 \subseteq A \times A$ such that $R = R_1 \cup R_2$ and $R_1 \perp R_2$,
- $\mathcal{A}, R, \nu \models^{\mathbf{FRSL}} \varphi -\!\!* \psi$ iff $\mathcal{A}, R', \nu \models^{\mathbf{FRSL}} \varphi$ implies $\mathcal{A}, R \cup R', \nu \models^{\mathbf{FRSL}} \psi$ for every $R' \subseteq A \times A$ such that $R \perp R'$.

Note that in the semantics of the separating connectives we require disjointedness of the *domains* of the relations, not the disjointedness of the relations themselves. This design choice is discussed in more detail later. In the definition of WRSL, it is not strictly necessary to require that R_1 and R_2 are finite, since this follows from the fact that their union must be finite too.

We can embed (functional) separation logic in relational separation logic as follows (we here only sketch the main ideas, and leave out the technical details). Let be a formula of separation logic. We then define the translation $T(\ )$ by induction on the separation logic formula :

- $T(?) = ?$,
- $T(x \doteq y) = (x \doteq y)$,
- $T(x \mapsto y) = (x \mapsto y)$,
- $T(C(x_1; \ldots; x_n)) = C(x_1; \ldots; x_n)$,
- $T(\ \ \to\ \ ) = T(\ ) \to T(\ )$,
- $T(\forall x\ ) = \forall x(T(\ ))$,
- $T(\qquad) = T(\ ) \quad T(\ )$,
- $T(\qquad) = (\mathit{fun}(,\mapsto) \wedge T(\ )) \quad T(\ )$.

We have that this translation preserves the semantics of full separation logic:

$$A; h; \ \models^{\mathbf{FSL}} \quad \text{if and only if } A; h; \ \models^{\mathbf{FSL}} T(\ )$$

because $\mathit{fun}(,\mapsto)$ already holds with respect to an interpretation based on heaps.

Proposition 2.5.3.

$$A; h; \ \models^{\mathbf{FSL}} \quad \textit{if and only if } A; \mathit{Graph}(h); \ \models^{\mathbf{FRSL}} T(\ ):$$

Conversely, if we also want to embed relational separation logic in (functional) separation logic, we need to encode binary relations in heaps by the introduction of multiple sorts and a binary relation representing *elementhood*: one sort corresponds to the elements, and another sort corresponds to non-empty sets of elements, and the binary relation is interpreted to represent the elements of non-empty sets of elements. In relational separation logic a location can be related to zero or more values. In our two-sorted separation logic we restrict locations to the sort of elements and values to the sort of non-empty sets of elements: then a heap either leaves a location not allocated or it is allocated and is assigned to a set consisting of one or more elements. The translation T^0 maps every connective in an identical manner, but only needs to reinterpret the points-to primitive:

$$T^0(x \mapsto y) = \exists S((x \mapsto S) \wedge (y \in S))$$

where S is of the non-empty set sort, and $(y \in S)$ indicates that value y is a member of the set S.

To make it possible to embed relational separation logic in separation logic in this straightforward manner forms the technical motivation for the disjointedness of the domains of relations in the semantics of the separating connectives of relational separation logic. Note that it is also possible to embed one-sorted separation logic into this two-sorted separation logic, where a heap that maps a location to a single value is represented by mapping that same location to the singleton set consisting of the sole value.

The second difference between separation logic and second-order logic is, intuitively, the *local perspective* of separation logic, which is determined by the 'current' heap. Separation logic has a restricted form of quantification over heaps by the modalities introduced earlier:

- holds in a heap h if holds in *every* heap regardless of h,
- holds in a heap h if holds in *some* heap regardless of h.

In dyadic second-order logic one can also quantify over binary variables (variables of arity 2), but multiple different such variables can be in scope. In contrast, in the semantics of separation logic we consider formulas with respect to a single heap.

To illustrate how subtle this difference is, we extend the syntax of separation logic with the binding operator $(\#R\ )$ which binds the binary second-order variable R in the evaluation of to the current interpretation of the points-to relation. The binding operator acts in a similar way as a quantifier: R is bound in in the formula $(\#R\ )$. For this extension, we need to extend the language of separation logic, in the sense that it is allowed to apply binary variables to individual variables to form primitive formulas, as in second-order logic. Note that this extension of the language still lacks quantification over second-order variables. Further, we need to adapt the semantics of relational separation logic, which we call FRSL#, since now valuations include assignments of relations to binary second-order variables.

We then give the following semantics to the binding operator:

- $A; R; \ j=^{\mathbf{FRSL}^{\#}} (\#R\ )$ if and only if $A; R; \ [R := R]\, j=^{\mathbf{FRSL}^{\#}}$.

Consider, for example, that for formulas of the form $(\#R(\quad))$ in the place of the subformulas and we can still refer to the (outer) heap. Specifically, in the place of we can express the locations that are in the extended part of the heap: $(x\ ,!\ ) \wedge \neg \exists y R(x;y)$ holds for those locations x in the domain of the heap with which was evaluated, but were not allocated in the outer heap.

Alternatively, if we would translate this extended language to dyadic second order logic (as in Definition 2.4.4), we would let the binding operator correspond to bounded (second-order) quantification

$$[\#R\ ](R') = \exists R((R \quad ,!\ ) \wedge [\ ](R'));$$

where R and R' are different variables, and $(R \quad ,!\ )$ abbreviates the first-order formula $\forall x;y(R(x;y) \leftrightarrow (x\ ,!\ y))$ that ensures the valuation of R coincides with

the current interpretation of the relation. Here we see why introducing relational separation logic is useful, since in dyadic second-order logic binary variables range over *arbitrary* relations. Note that, just like quantifiers, it is possible to perform variable renaming of the bound variable R in case it clashes with the chosen variable R^0 used for translation, but we leave these technical details to the reader.

The expressive power of this binding operator lies in that it allows to 'break the spell' of the local perspective since the bound second-order variable allows, in the local context of the current interpretation of the points-to relation, to refer to the 'outer' interpretations that have generated it (by the separating connectives).

This extension of separation logic consequently allows for a simple, compositional translation of dyadic second-order logic. For notational convenience, let $(\forall R\ )$ denote the (extended) separation logic formula $\neg(\#R\ )$. Now we have that

$$\mathsf{A}; R; \quad \models^{\mathbf{FRSL}^{\#}} \forall R\ \ \text{if and only if}\ \ \mathsf{A}; R; \ [R := R^0] \models^{\mathbf{FRSL}^{\#}} \ ;$$

for every relation R^0. To translate every dyadic second-order formula into a corresponding formula of separation logic, we first translate it into separation logic (extended with a binding operator). Let be a dyadic second-order formula (which is assumed not to contain occurrences of the points-to relation of separation logic). We then define $T(\ )$ by induction on the structure of the dyadic second-order formula :

- $T(?) = ?$,
- $T(R(x_1; x_2)) = R(x_1; x_2)$,
- $T(C(x_1; \ldots; x_n)) = C(x_1; \ldots; x_n)$,
- $T(\ \ !\ \ ) = T(\ )\ !\ T(\ )$,
- $T(\forall x(\ )) = \forall x T(\ )$,
- $T(\forall R(\ )) = \forall R(T(\ ))$.

Note that in the last clause we use our abbreviation introduced above. It now follows that $\mathsf{A}; \models^{\mathbf{CL}}$ if and only if $\mathsf{A}; R; \models^{\mathbf{FRSL}^{\#}} T(\ )$ for an arbitrary binary relation R. Note that $T(\ )$ does not depend on the interpretation of the points-to relation. The resulting formula $T(\ )$ is interpreted with respect to relational separation logic, but that can again be embedded in full (functional) separation logic by the introduction of the two sorts as mentioned above.

We end our analysis by stating two open problems:

Problem 2.2. *Is the binding operator* $(\#R\ )$ *definable by a standard formula of separation logic in the semantics* FRSL#*?*

Problem 2.3. *Are* FRSL *and* FRSL# *equally expressive?*

If the answers to both questions are affirmative, then it is possible to express the binding operator in FRSL too. We conjecture that this is not possible: in the first problem we have in the extended language and semantics FRSL# additional second-order variables, whereas in FRSL we only have first-order variables available. In the second problem, the binder breaks the 'local perspective' of separation logic. It may be possible, however, to express the binding operator relative to a sufficiently rich structure that allows encoding heaps as objects in the domain of the underlying structure, but the details remain to be worked out.

Bibliographic note

Remarkably, [35] presents a rather intricate encoding of (dyadic) weak second-order logic into standard separation logic. Apparently this restriction to finite heaps allows to break the local perspective. Our conjecture, however, is that full separation logic is strictly less expressive than (dyadic) second-order logic.

Chapter 3

Proof theory of separation logic

Our goal is to obtain a sound and complete, finitary proof system for reasoning about the valid formulas of separation logic. To that end we introduce two calculi consisting of separation logic formulas and *rooted assertions*, which are separation logic formulas that are annotated with a representation of the heap with respect to which the separation logic formula is evaluated. We shall argue that the obtained finitary proof systems are sound and complete with respect to different interpretations. However, before jumping to the conclusion, we first need to explain the development of this result.

As seen in the previous chapter, the semantics WSL and FSL can not be adequately used as interpretations, due to their failure of compactness (of the satisfaction relation or the semantic consequence relation). The main model-theoretic results are that WSL is already non-compact even for the pure formulas, and in the setting of FSL we can express: (1) finiteness of structures, (2) well-foundedness of the points-to relation, and (3) existence of countably infinite and uncountable structures. As a consequence we have that FSL satisfies neither compactness nor the downward and upward Löwenheim-Skolem theorems. In fact, we have seen that the well-foundedness of the points-to relation can already be expressed in FSL using only separating conjunction. Consequently, FSL without separating implication is already non-compact. For FSL without separating implication but in which separating conjunction only occurs positively, the fragment which we called separation logic light (SLL), we do have compactness, but its semantic consequence relation is not compact. Non-compactness (of the satisfaction relation or the semantic consequence relation) implies that there does not exist a finitary, sound and complete proof system with respect to these interpretations.

Recall that in Section 2.4 we have seen that it is possible to embed full separation logic in dyadic second-order classical logic, and in Section 2.5 we have seen an investigation of the converse: can dyadic second-order logic be embedded in full separation logic too? This is a good starting point in light of the above goal, since if separation logic is equally expressive as second-order logic we could simply use Henkin's semantics of second-order logic directly. However, this question is

still open, and we conjecture that the binding operator cannot be expressed in separation logic due to its inherent local perspective.

The question thus arises whether there exists an *alternative* interpretation of separation logic that does allow for a finitary, sound and complete proof system. Clearly, the main complexity of separation logic stems from the (second-order) quantification over heaps (or sub-heaps, in the case of the separating conjunction). For second-order logic a sound and complete axiomatization can be obtained by generalizing its semantics by means of so-called *general structures*. Such structures extend first-order structures with a set of possible interpretations of the second-order variables. For example, instead of interpreting a second-order variable of arity 1 as ranging over *all* possible subsets of the given first-order domain, a general structure restricts its interpretation to a given set of such subsets. The standard structures of second-order logic are thus a particular instance of general structure. This generalization of the semantics of second-order logic allows for a sound and complete axiomatization by restricting to Henkin structures [108]. A Henkin structure is a general structure for second-order logic which additionally satisfies the comprehension axiom scheme

$$\exists R^n \forall x_1, \ldots, x_n (R^n(x_1, \ldots, x_n) \Leftrightarrow \varphi(x_1, \ldots, x_n))$$

for any second-order formula $\varphi(x_1, \ldots, x_n)$ which does not contain the n-ary variable R. In the *arithmetic* comprehension axiom $\varphi(x_1, \ldots, x_n)$ is first-order.

Generalizing the semantics of separation logic accordingly in terms of a given set of possible heaps, which does not necessarily contain *all* heaps, we can formulate in separation logic the following version of the arithmetic comprehension axiom scheme

$$(\forall x, y((x \mapsto y) \Leftrightarrow \varphi(x, y)))$$

which expresses the existence of a heap such that its *graph*, as denoted by the points-to relation $\mapsto$, satisfies the pure first-order formula $\varphi(x, y)$. The formula φ is pure in the sense that it does not involve the separation connectives or the points-to relation. The $\Diamond$-modality expresses the existence of a heap which satisfies the associated formula. Such an instance of the arithmetic comprehension axiom holds if there exists a heap which is characterized by the formula $\varphi(x, y)$. Therefore, we introduce a new interpretation of separation logic that restricts the (second-order) quantification over heaps to *first-order definable* heaps.

For this new interpretation we introduce a *sequent calculus* which is sound and complete. In this sequent calculus we introduce so-called *rooted formulas* $\varphi @ \psi$ where $\varphi(x, y)$ are pure first-order formulas. In the interpretation of rooted formulas, $\varphi(x, y)$ determines the interpretation of the heap with respect to which ψ is evaluated. The completeness proof is based on the construction of a model for a *deductively consistent* theory (a theory from which false is not derivable), following Henkin's approach. From the completeness proof we further derive that this new interpretation satisfies both compactness and the downward Löwenheim-Skolem theorem. By the seminal theorem of Lindström [217, 210] we then infer that this new interpretation is as expressive as first-order logic.

However, we cannot generalize arithmetic comprehension to arbitrary separation logic formulas because that leads to obvious contradictions, such as

$$(\exists x;y((x \mapsto y) \ast \neg(x \mapsto y))).$$

Simply requiring that the points-to relation does not occur in $(x;y)$ does not give more than what the arithmetic comprehension axiom above gives, because compositions of pure first-order formulas with separating connectives are equivalent to some pure first-order formula (this easily follows from the semantics of separation logic). To overcome this issue, we extend our rooted formulas $@$ without any restrictions on , where @ can now be understood as a special let binding connective. We need a new interpretation of separation logic, which no longer can be captured by a syntactic comprehension axiom scheme, and instead we consider a class of general structures which satisfy a closure condition called *semantic comprehension*.

For this new, second interpretation of separation logic we introduce a *natural deduction calculus* which is also sound and complete. We show completeness by constructing models for deductively consistent theories, in a similar way as for our sequent calculus.

3.1 Sequent calculus

In full relational separation logic we have that the following formulas are valid:

$$(\exists x;y((x \mapsto y) \ast \quad (x;y)))$$

where is a pure, first-order formula. The above class of formulas, called the *arithmetic comprehension axiom scheme*, expresses, for each pure first-order formula $(x;y)$, the existence of a relation such that its *graph*, as denoted by the points-to relation $\mapsto$, satisfies $(x;y)$. In this section, we shall consider a restriction of full relational separation logic in which we consider *only* those relations which have a corresponding first-order description. These relations are called first-order definable. This means that we restrict our attention to the interpretation of the separating connectives to such first-order definable binary relations.

Let denote a first-order formula which does not contain occurrences of the points-to relation $\mapsto$ of separation logic. We have the standard inductive truth definition $A; \models^{\mathbf{CL}}$ for first-order formulas . By $(x_1;:::;x_n)$ we denote that the free (first-order) variables of are among the distinct variables $x_1;:::;x_n$. A formula $(x;y)$ is called a *binary* formula. For notational convenience we assume that the variables x and y of any binary formula are fixed and do not occur in any separation logic formula. A binary formula is also simply denoted by , omitting its free variables x and y. Given a structure $\mathfrak{A} = (A; I)$ and a first-order formula $(x;y)$, we denote by $Rel_{\mathfrak{A}}( )$ the relation $\{(a;b) \mid A; [x \mapsto a; y \mapsto b] \models^{\mathbf{CL}} \quad g \quad A \quad A\}$. Note that the evaluation of $(x;y)$ only depends on the values of its free variables x and y, that is, $A; \models^{\mathbf{CL}}$ if and only if $A; {}^{0} \models^{\mathbf{CL}}$, where $(x) = {}^{0}(x)$ and $(y) = {}^{0}(y)$. By $(t;t^{0})$ we denote the result of replacing in $(x;y)$ the variables x and y by terms t and t^{0}, respectively (if necessary renaming bound variables to ensure that the variables of t and t^{0} do not become bound).

Definition 3.1.1 (First-order definability). For a given structure $\mathsf{A} = (A; I)$, the relation $R \subseteq A \times A$ is *first-order definable* if $R = Rel_{\mathsf{A}}(\alpha)$, for some binary formula $\alpha(x; y)$.

Note that, given a structure $\mathsf{A} = (A; I)$, we have $I(R) = Rel_{\mathsf{A}}(R)$, that is, for any binary relation symbol R its interpretation $I(R)$ is trivially a first-order definable relation. We introduce the abbreviation $\alpha = \alpha_1 \rfloor \alpha_2$ that denotes that the binary formulas $\alpha_1(x; y)$ and $\alpha_2(x; y)$ represent a partition of the binary formula $\alpha(x; y)$ which is expressed by the conjunction of the three formulas

$$\begin{array}{l} \forall x; y(\alpha(x; y) \Leftrightarrow (\alpha_1(x; y) \vee \alpha_2(x; y))); \\ \forall x; y; z(\alpha_1(x; y) \rightarrow \neg \alpha_2(x; z)); \\ \forall x; y; z(\alpha_2(x; y) \rightarrow \neg \alpha_1(x; z)): \end{array}$$

The latter two formulas, which state that the domains of the binary relations represented by $\alpha_1(x; y)$ and $\alpha_1(x; y)$ are disjoint, we abbreviate by $\alpha_1 \perp \alpha_2$. A similar abbreviation can be given for binary relation symbols $R = R_1 \rfloor R_2$. By usual abuse of notation, we mean that the equality holds for the *extension* of R (so we need to universally quantify two variables $x; y$ and apply them to $R; R_1; R_2$).

In this section, to avoid confusion between formulas of separation logic and formulas of first-order logic, we shall denote the former by $p; q$ and the later by $\varphi; \psi$. We introduce the semantics $\mathsf{A}; R; s \models^{\mathbf{FORSL}} p$ which is a *restriction* of the general relational semantics of separation logic (see also Definition 3.4.2) such that instead of quantifying over arbitrary binary relations, the separating connectives involve quantification over first-order definable binary relations. It is worthwhile to observe here that, as for Henkin models of second-order logic, the implicit second-order quantification depends on the underlying signature of function and relation symbols. Extending or restricting the signature affects the semantics of formulas of the 'old' signature.

To reason about the implicit quantification over definable (binary) relations, we introduce *rooted* assertions of the form $p@\alpha$, where α denotes a binary formula and p is a formula of separation logic. We define $\mathsf{A}; \models^{\mathbf{FORSL}} p@\alpha$ if and only if $\mathsf{A}; R; \models^{\mathbf{FORSL}} p$, where $R = Rel_{\mathsf{A}}(\alpha)$. The variables x and y of the binary formula $\alpha(x; y)$ are thus implicitly bound by the @-connective, that is, $\mathsf{A}; \models^{\mathbf{FORSL}} p@\alpha$ if and only if $\mathsf{A}; {}^{0} \models^{\mathbf{FORSL}} p@\alpha$, for any α and ^{0} such that $\alpha(z) = {}^{0}(z)$, for any free variable occurring in p.

We further assume that our signature includes a (countably) infinite set of binary relation symbols R (needed for the selection of fresh 'witnesses'). However, definability of binary relation by a first-order formula should not depend on these additional binary relation symbols. That is, these binary relation symbols are added as 'bookkeeping devices'. Alternatively, we could have introduced these as (second-order) *variables* and extend evaluations so that $(R) \subseteq D \times D$, for any such (second-order) variable. However, for both technical and notational convenience we prefer to define their semantics as part of a structure.

Note that the separating connectives are interpreted in terms of relations which are definable by first-order formulas which do not involve the points-to

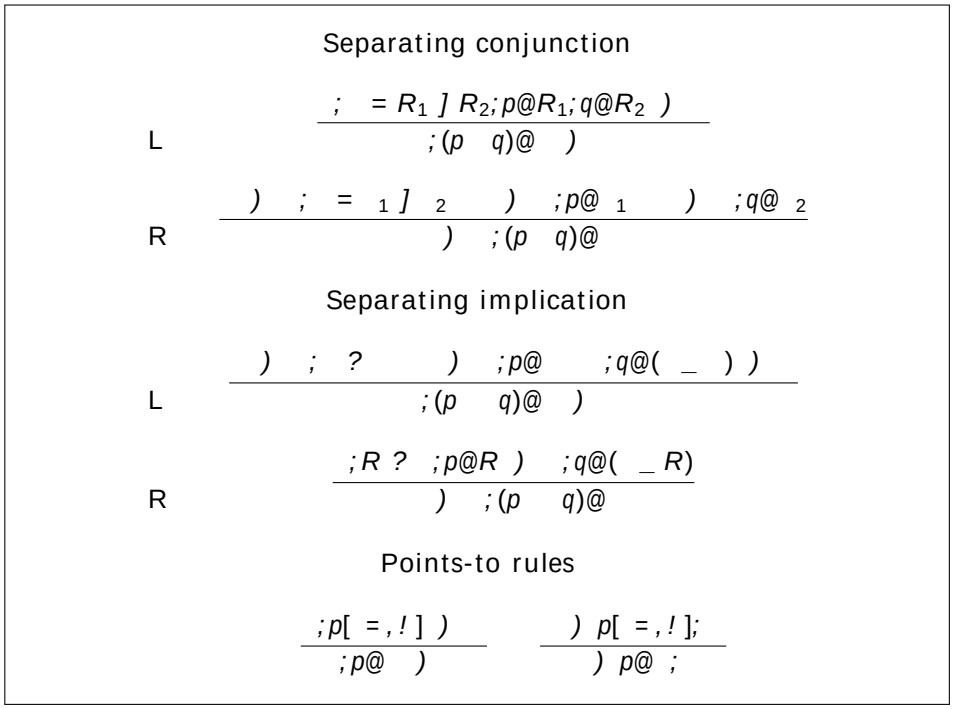


Figure 3.1: Sequent calculus for FORSL. The binary relation symbols $R_1; R_2$ and R introduced in the rules L and R are 'fresh'. In the points-to rules p denotes a semi-pure formula (which does not contain occurrences of the separating connectives).

relation $,!$. This allows for the following alternative *predicative*[1] characterization of the semantics of the separating connectives in rooted assertions (used in both the soundness and completeness proofs).

Lemma 3.1.2. *We have*

- $A; j \models^{\mathbf{FORSL}} (p \quad q)@$ *if and only if there exist binary formulas* $_1$ *and* $_2$ *such that:*
  $A; j \models^{\mathbf{FORSL}} = _1] _2$,
  $A; j \models^{\mathbf{FORSL}} p@ _1$, *and*
  $A; j \models^{\mathbf{FORSL}} q@ _2$.
- $A; j \models^{\mathbf{FORSL}} (p \quad q)@$ *if and only if*
  $A; j \models^{\mathbf{FORSL}} ?$ *and* $A; j \models^{\mathbf{FORSL}} p@$ *implies*
  $A; j \models^{\mathbf{FORSL}} q@(\_)$, *for all binary formulas* .

We now develop a calculus for sequents $A_1; : : : ; A_n \ ) \ B_1; : : : ; B_m$, where each A_i (given $i = 1; : : : ; n$), and B_j (given $j = 1; : : : ; m$), is constructed from first-order

[1] For a foundational discussion concerning predicativity, see [57].

formulas and rooted assertions, which can be further composed using propositional connectives and quantification of first-order variables. In particular, we have the following abstract grammar:

$$\begin{array}{rcl} \Gamma\,;\,\Delta & ::= & \cdot \mid (x \doteq y) \mid C(x_1,\ldots,x_n) \mid (\quad \multimap \quad) \mid (\forall x\ ) \\ p;q & ::= & \cdot \mid (x \doteq y) \mid (x \not\doteq y) \mid C(x_1,\ldots,x_n) \mid (p \ast q) \mid (p \mathrel{-\!\!*} q) \mid (p \rightarrow q) \mid (\forall x p) \\ A;B & ::= & \cdot \mid p@\ \mid (A \rightarrow B) \mid (\forall x A) \end{array}$$

where in rooted formulas $p@$ the first-order formula has at most free variables x, y. Note that the free variables of $p@$ are only the free variables of p, since the @-connective binds the free variables x and y.

This calculus is an extension of standard first-order sequent calculus, where the standard rules are applicable with respect to top-level propositional connectives and quantifiers. Figure 3.1 shows the left and right rules for separating conjunction and implication. These rules closely follow the translation of relational separation logic into second-order logic, eliminating the explicit second-order quantification by applying the standard proof rules for second-order quantification (which themselves are straightforward generalizations of the rules for first-order quantification, instantiating the second-order variables by formulas). The binary relation symbols R_1, R_2 and R introduced in the rules $\ast$L and $\mathrel{-\!\!*}$R are 'fresh' binary relation symbols, that is, they must not appear in the formulas of the conclusion of the rules.

We also have rules which allow classical reasoning under rooted assertions: $(p \ast q)@ \dashv\vdash (p@\ ) \ast (q@\ )$, where $\ast$ denotes binary propositional connectives, e.g., conjunction, disjunction, and implication, $(\neg p)@ \dashv\vdash \neg(p@\ )$, and $(\forall x p)@ \dashv\vdash \forall x(p@\ )$, and similarly $(\exists x p)@ \dashv\vdash \exists x(p@\ )$. Further, we have $(\forall x, y(\quad \dashv\vdash\ )) \rightarrow (p@\quad \dashv\vdash p@\ )$. It is straightforward to validate these rules, but we omit the details of the semantics $A, \models_{\mathbf{FORSL}} A$, which follows the standard Tarski-style classical semantics, given the semantics of rooted assertions which may appear in the place of atomic formulas.

In the so-called 'points-to' rules of Figure 3.1 the formula p does not involve occurrences of the separating connectives. Such a formula of separation logic we call *semi-pure*. Note that it differs from pure first-order formulas in that semi-pure formulas additionally may involve the points-to relation. For such formulas we denote by $p[=, \not\doteq]$, for any binary formula (x, y), the result of replacing every atomic assertion $(t \not\doteq t')$ in p by (t, t'), which is a pure first-order formula. It follows that $A, \models_{\mathbf{FORSL}} p[=, \not\doteq]$ if and only if $A, Rel_A(\ ), \models_{\mathbf{FORSL}} p$, for any semi-pure formula p.

We now see a number of example proofs, in which we use the sequent calculus defined above.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(q@R, R_1 \mathrel{-\!\!*} R_2) \vdash q@R \quad \Gamma(q@R, p@R_1) \vdash q@(R_1 \ast R_2)}{R = R_1 \circ R_2, p@R_1, (p \ast q)@R_2 \vdash q@R}\,\ast\text{L}}{(p \ast (p \rightarrow q))@R \vdash q@R}\,\ast\text{L}}{\Gamma \vdash (p \ast (p \rightarrow q))@R \rightarrow q@R}}{\Gamma \vdash ((p \ast (p \rightarrow q)) \rightarrow q)@R}$$

As a first example of the use of the sequent calculus, above we have a derivation of the sequent $\vdash ((p \quad (p \quad q)) \; ! \; q)@R$ which represents the validity of $(p$ $(p \quad q)) \; ! \; q$. This derivation essentially consists of an application of the rule L followed by an application of the rule L . In this derivation denotes the formulas $R = R_1 \;] \; R_2; p@R_1$ generated by the application of rule L . The second premise of the application of the rule L is derivable from an instance of the axiom $; A \vdash A;$. Note that (in the L rule) is instantiated with R_1. The first and third premise follows from the fact that $R = R_1 \;] \; R_2$ reduces to $R_1 \; ? \; R_2$ and $R = R_1 \; [\; R_2$ (that part of the proof is not shown above).

Next we show how to use the calculus in reasoning about the equivalence of weakest preconditions that arise in the practice of verifying the correctness of heap manipulating programs. Let p denote the weakest precondition

$$(u \; , ! \quad) \wedge (z = 0 \; / \; u = v \; . \; v \; , ! \; z)$$

of the heap update $[u] := 0$ which ensures the postcondition $v \; , ! \; z$ after assigning the value 0 to the location denoted by the variable u, where $/ \; b \; .$ abbreviates $(b \wedge \quad) \_ (: b \wedge \quad)$ (in Section 4.4 a dynamic logic extension of separation logic is introduced which generates this weakest precondition). The standard rule for backwards reasoning in [188] gives the weakest precondition $(u \; 7! \quad) \quad (u \; 7! \; 0$ $v \; , ! \; z)$, which we denote by p^0. These preconditions are equivalent because both are the weakest.

In fact, the equivalence between the above two formulas can be expressed in quantifier-free separation logic, for which a complete axiomatization of all valid formulas has been given in [70]. In the sequent calculus we can express the equivalence of p and p^0 in terms of the sequent $fun(R) \vdash (p \; \$ \; p^0)@R$. Here R is an arbitrary binary relation symbol used to represent the current interpretation of the points-to relation. We abbreviate $8x; y; z((R(x; y) \wedge R(x; z)) \; ! \; y = z)$ by $fun(R)$. A proof of the above sequent amounts to proving the sequents $fun(R); p^0@R \vdash p@R$ and $fun(R); p@R \vdash p^0@R$.

Proposition 3.1.3. $\vdash fun(R); p@R \vdash p^0@R.$

Proof. This direction is easy to prove, by a case analysis whether $u = v$ holds or not. If $u = v$, then $z = 0$ and so we can easily prove $v \; , ! \; z$ in a heap where $u \; , ! \; 0$. Otherwise, if $u \neq v$, then $v \; , ! \; z$ follows immediately. □

Lemma 3.1.4. $\vdash fun(R); p^0@R \vdash p@R.$

Proof. Below we present a high-level proof of the first sequent, abstracting from some basic first-order reasoning in the calculus. By an application of L to derive the sequent $fun(R); p^0@R \vdash p@R$ it su□ces to derive

$$fun(R); R = R_1 \;] \; R_2; (u \; 7! \quad)@R_1; (u \; 7! \; 0 \quad v \; , ! \; z)@R_2 \vdash p@R$$

for some fresh R_1 and R_2. Let $(x; y)$ denote the binary formula $x = u \wedge y = 0$. Further, let denote the set of formulas $fun(R); R = R_1 \;] \; R_2; (u \; 7! \quad)@R_1$. By an application of the rule L it then su□ces to prove the following sequents (from

) we can derive) A; by right-weakening). First we prove) $R_2 \setminus = ;$: By the points-to rules the rooted assertion $(u \mapsto)@R_1$ (appearing in) reduces to $9z(R_1(u;z) \wedge 8x;y(R_1(x;y) \; ! \; x = u \wedge y = z))$ (the forall-part of the formula is due to the 'strict' points-to which states that the domain contains u as its only location). Further, $R_2 \setminus = ;$ logically boils down to $:9x;y(R_2(x;y) \wedge (x = u \wedge y = 0))$, that is, $:R_2(u;0)$, which in basic first-order logic follows from $9zR_1(u;z)$ and the assumptions $R = R_1 \;] \; R_2$ and $fun(R)$.

Second, we prove) $(u \mapsto 0)@$: By the points-to rules $(u \mapsto 0)@$ (using the expanded definition of $u \mapsto 0$ and the definition of the substitution $[= , !]$) reduces to $(u = u) \wedge (0 = 0) \wedge 8x;y((x = u \wedge y = 0) \; ! \; (x = u \wedge y = 0))$ which is equivalent to true.

And, finally, we prove $;(v , ! \; z)@(R_2 \_) \;) \; p@R$: First note that (again, by the points-to rules)

$$((u , ! \;) \wedge (z = 0 / u = v \; . \; v , ! \; z))@R$$

reduces to

$$(9zR(u;z)) \wedge (z = 0 / u = v \; . \; R(v;z))):$$

The assertion $9zR(u;z)$ clearly follows from the assumptions $R = R_1 \;] \; R_2$ and $(u \mapsto)@R_1$ in . To prove $z = 0 / u = v \; . \; R(v;z)$, we first reduce the assumption $(v , ! \; z)@(R_2 \_)$ to $R_2(v;z) \_ (v = u \wedge z = 0)$. Now, if $v = u$ then $:R_2(v;z)$, because of the assumptions $fun(R)$, $R = R_1 \;] \; R_2$ and $(u \mapsto)@R_1$. So we have that $z = 0$. Otherwise, we have $R_2(v;z)$, and thus $R(v;z)$, because $R = R_1 \;] \; R_2$. □

3.2 Soundness and completeness

We denote by `) that there exists a proof of the sequent) . To define $j=$) , let denote a substitution which assigns to every binary relation symbol R of the sequent) a binary formula . Such a substitution simply replaces occurrences of $R(t;t^0)$ by $(t;t^0)$, where $(R) = (x;y)$. By $j=$) we then denote that A; $j=$ (that is, A; $j= A$, for every $A \; 2$) implies A; $j=$ $\bigvee$ (that is, A; $j= B$, for some $B \; 2$), for every A; and every substitution .

In the soundness proof below we use these substitutions to instantiate the fresh binary relation symbols introduced in the rules L and R . Note that updating the interpretation of these symbols (as provided by A) would affect the semantics of the separating connectives if binary formulas would refer to these fresh binary relation symbols (note that they are only supposed not to appear in formulas of the conclusion of the rules L and R). See also the previous discussion about 'bookkeeping devices'.

We generalize the above notions of derivability and validity to possibly infinite : ` indicates that ` ^{0}) , for some finite ^{0} , and $j=$ indicates that for every substitution we have that A; $j=$ (that is, A; $j= A$, for every $A \; 2$) implies A; $j= B$, for some $B \; 2$.

For the soundness proof we need the following substitution lemma.

Lemma 3.2.1 (Substitution lemma). *A; Rel*$_{\mathsf{A}}(\ )$*;* $\models p$ *if and only if* A; $\models p[\ =, I\ ]$*, for any semi-pure formula p.*

Theorem 3.2.2 (Soundness). *We have that* $\vdash$) *implies* $\models$) .

Proof. We prove that the rules for the separating connectives preserve validity. The points-to rules are sound because A; *Rel*$_{\mathsf{A}}(\ )$; $\models p$ if and only if A; $\models p[\ =, I\ ]$, for any semi-pure formula p (note that $p[\ =, I\ ]$ is a pure first-order formula which does not depend on the heap).

$\bigvee$ L : Let A; $\models$ and A; $\models (p \quad q)@$. We have to show that A; $\models$. By Lemma 3.1.2, there exist $_1$ and $_2$ such that A; $\models (\ ) = {}_1 \rceil {}_2$, A; $\models p \ @ \ {}_1$, and A; $\models q \ @ \ {}_2$. Let $^0 = [R_1; R_2 := {}_1; {}_2]$. Since R_1 and R_2 are fresh and as such do not appear in ; $(p \quad q)@$, it follows that A; $\models {}^0 \ {}^0$, where $^0 =$; $= R_1 \rceil R_2; p@R_1; q@R_2$. By the validity of the premise we thus obtain that A; $\models {}^0$. Since R_1 and R_2 also do not appear in , we conclude that A; $\models$.

R : Let A; $\models$ and suppose that A; $\not\models \bigvee$. From the validity of the premises it then follows that A; $\models = ({}_1 \rceil {}_2)$, A; $\models p \ @ \ {}_1$, and A; $\models q \ @ \ {}_2$, By Lemma 3.1.2 we conclude A; $\models (p \quad q)@$.

$\bigvee$ L : Let A; $\models$ and A; $\models (p \quad q)@$, and suppose that A; $\not\models \bigvee$. From the validity of the first two premises it then follows that A; $\models ?$ and A; $\models p \ @$. By Lemma 3.1.2 again, it follows that A; $\models q \ @(\ \_ \ )$. By the validity of the third premise we thus derive that A; $\not\models$, which contradicts our assumption.

R : Let A; $\models$ and suppose that A; $\not\models \bigvee$. We have to show that A; $\models (p \quad q)@$. Let be such that A; $\models ? (\ )$ and A; $\models p \ @$. Further, let R be a fresh variable and $^0 = [R :=]$. It follows that A; $\models {}^0 \ {}^0$, where $^0 =$; $R \ ?$; $p@R$ and A; $\not\models {}^0$. And so we derive from the validity of the premise of the rule that A; $\models q \ @(\quad [\ ])$. Since was arbitrarily chosen, by Lemma 3.1.2 again we conclude that A; $\models (p \quad q)@$. □

As a corollary we obtain that $\vdash$ implies $\models$.

Following the completeness proof of first-order logic as described in [108], it suffices to show that every consistent set of formulas is satisfiable (the so-called 'model existence theorem'). A set of formulas is consistent if $\not\vdash$;. We first show that every consistent set of formulas can be extended to a maximal consistent set. To this end we assume an infinite set of 'fresh' binary relation symbols R that do not appear in . We construct for any consistent set a maximal consistent extension ^{1}, assuming an enumeration of all formulas A (which also covers all first-order formulas). We define $_0 =$ and $_{n+1}$ satisfies the general rule: if $_n; A_n \not\vdash$; then $_n \ [\ f A_n g \quad _{n+1}$, otherwise $_{n+1} = {}_n$. Additionally, in case A_n is added and A_n is of the form $9xA$ or a rooted assertion $(p \quad q)@$ or $:(p \quad q)@$, we also include corresponding *witnesses* in $_{n+1}$:

- If A_n is of the form $9xA$ we additionally add $A(y)$, where $A(y)$ results from replacing all free occurrences of x in A by the fresh variable y which does not appear in $_n$.

Note that $A(y)$ can indeed be added consistently because from $\Gamma_n; A(y) \vdash \Delta$; we would derive $\Gamma_n; \exists x A \vdash \Delta$, which contradicts the assumption that $\Gamma_n; \exists x A \not\vdash \Delta$.

- If A_n is of the form $(p \mathrel{-\!\!*} q)@$ we additionally add the formulas $\Gamma = R_1 \mapsto R_2; R_1 \# R_2; p@R_1$, and $q@R_2$, where R_1 and R_2 are fresh (e.g., not appearing in Γ_n).

  Note that these formulas can indeed be added consistently because from $\Gamma_n; \Gamma = R_1 \mapsto R_2; R_1 \# R_2; p@R_1; q@R_2 \vdash \Delta$; we would derive $\Gamma_n; (p \mathrel{-\!\!*} q)@ \vdash \Delta$; (by rule $\mathrm{L}\mathrel{-\!\!*}$).

- If A_n is of the form $\neg(p \mathrel{-\!\!*} q)@$ (which is equivalent to $\neg((p \mathrel{-\!\!*} q)@)$) we additionally add the formulas $R \# \Gamma; p@R(x,y)$, and $\neg q@(\_ \mathrel{-\!\!*} R)$, where R is fresh (e.g., not appearing in Γ_n).

  Note that these formulas can indeed be added consistently because from $\Gamma_n; R \# \Gamma; p@R(x,y); \neg q@(\_ \mathrel{-\!\!*} R) \vdash \Delta$; we would derive $\Gamma_n \vdash (p \mathrel{-\!\!*} q)@$ (by rule $\mathrm{R}\mathrel{-\!\!*}$), which contradicts the assumption that $\Gamma_n; \neg(p \mathrel{-\!\!*} q)@ \not\vdash \Delta$;.

We define $\Gamma^1 = \bigcup_n \Gamma_n$. By construction Γ^1 is maximal consistent. Given a maximal consistent set of formulas Γ, let $\mathcal{A} = (D, I)$, where D is the set of equivalences classes $[x] = \{y \mid x \approx y \in \Gamma\}$. For any relation symbol R (excluding the points-to relation $\hookrightarrow$) we define

$$I(R)([x_1], \ldots, [x_n]) = \mathsf{true} \text{ if and only if } R(x_1, \ldots, x_n) \in \Gamma$$

Given a maximal consistent set of formulas Γ and the structure $\mathcal{A} = (D, I)$, a corresponding valuation ρ assigns to every variable x an equivalence class $[x]$. However, in the sequel we will represent such a valuation by a *substitution* s which simply assigns to each variable a variable. The value $I_s(x)$ of a variable x then is given by the equivalence class $[s(x)]$ of the variable $s(x)$.

Given a substitution s and formula A (of the sequent calculus) we denote by ts and As the result of replacing every free occurrence of a (first-order) variable x in t and A by $s(x)$, respectively. Note that $(p@\rho)s = ps@\rho$, because the meaning of $p@\rho$ does not depend on the free variables x and y of the binary formula $\rho(x,y)$.

Given a maximal consistent set of formulas Γ and the structure $\mathcal{A} = (D, I)$, it follows that $I_s(x) = [xs]$, for every variable x and substitution s.

Lemma 3.2.3. *Given a maximal consistent set of formulas Γ and the structure $\mathcal{A} = (D, I)$, we have $\mathcal{A}, s \models A$ if and only if $As \in \Gamma$, for every formula A and substitution s.*

Proof. The proof proceeds by induction on the following well-founded ordering $A < B$ on formulas of the sequent calculus: Let $\#A = (n, m)$, where n denotes the number of occurrences of the separating connectives and the @-connective of A and m denotes the number of occurrences of the (standard) first-order logical operations of A. Then $A < B$ if $\#A < \#B$, where the latter denotes the lexicographical ordering on $\mathbb{N} \times \mathbb{N}$ (w.r.t. the standard 'smaller than' ordering on the natural numbers). We treat the following main cases (for notational convenience $\mathcal{A}$ denotes the structure $\mathcal{A}, s$).

- For any semi-pure formula p (that is, which does not involve occurrences of the separating connectives) we have:
  $A; s \models p@\ $ if and only if (by definition)
  $A; Rel_A(\ ); s \models p$ if and only if (substitution lemma 3.2.1)
  $A; s \models p[\ =, !\ ]$ if and only if (induction hypothesis)
  $(p[\ =, !\ ])s \in$ if and only if
  $(p@\ )s \in$.

  Note that by an application of the points-to rules $(p[\ =, !\ ])s \in$ implies $\vdash (p@\ )s$, and so $(p@\ )s \in$, by the maximal consistency of . On the other hand, let $(p@\ )s \in$ and assume $(p[\ =, !\ ])s \notin$, that is, $(\neg p[\ =, !\ ])s \in$, by the maximal consistency of . By the points-to rules it then follows that $\vdash (\neg p@\ )s$, which contradicts the consistency of .

- Let $A; s \models A$, where A denotes the formula $(p \ast q)@$. By Lemma 3.1.2 there exist $_1$ and $_2$ such that $A; s \models = _1 \mathrel{]} _2$, $A; s \models p@\ _1$ and $A; s \models q@\ _2$. From the induction hypothesis it follows that $ps@\ _1; qs@\ _2; = _1 \mathrel{]} _2 \in$ (note that the first-order formula $= _1 \mathrel{]} _2$ does not contain free variables, and thus is not affected by the substitution s). So we derive by rule R that $\vdash (ps \ast qs)@$. By maximal consistency of , we then conclude that $(ps \ast qs)@\ \in$, that is, $As \in$.

  On the other hand, let $As \in$. That is, $(ps \ast qs)@\ \in$. By the construction of we have $= R_1 \mathrel{]} R_2; ps@R_1; qs@R_2 \in$, for some witnesses R_1 and R_2. By the induction hypothesis it then follows that $A; s \models p@R_1$ and $A; s \models p@R_2$. Further, the induction hypothesis gives $A; s \models = R_1 \mathrel{]} R_2$ (again, note that the formula $= R_1 \mathrel{]} R_2$ has no free variables, and thus is not affected by the substitution s). We conclude by Lemma 3.1.2 that $A; s \models (p \ast q)@$.

- Let $A; s \models A$, where A denotes the formula $(p \ast q)@$. Suppose $As \notin$. By the maximal consistency of , we then have $\neg(ps \ast qs)@\ \in$. By construction $R\ ?\ ; ps@R; \neg qs@(\ \_\ R) \in$, for some witness R, which contradicts $A; s \models (p \ast q)@$ (after application of the induction hypothesis and using Lemma 3.1.2 again).

  On the other hand, let $As \in$. To show that $A; s \models (p \ast q)@$, let $A; s \models\ ?$ and $A; s \models p@$, for some binary formula . By the induction hypothesis we have that $?\ ; ps@\ \in$. Suppose that $qs@(\ \_\ ) \notin$, that is $\neg qs@(\ \_\ ) \in$ (is maximal consistent), and thus $\ ; qs@(\ \_\ ) \vdash\ ;$. Applying rule L we then derive $\ ; (ps \ast qs)@ \vdash\ ;$, which contradicts the consistency of $((ps \ast qs)@\ \in)$. So we have that $qs@(\ \_\ ) \in$, that is, $A; s \models q@(\ \_\ )$, by the induction hypothesis. Since is chosen arbitrarily, it follows by Lemma 3.1.2 that $A; s \models (p \ast q)@$.

- Let A be a formula $p@$, where p denotes a semi-pure formula. Let $R = Rel_A(\ )$. We then have:
  $A; s \models p@\ $ iff (by definition)
  $A; R; s \models p$ iff (straightforward induction on p)

$A; s \models p[\ =, !\ ]$ i□ (induction hypothesis for $p[\ =, !\ ]$)
$ps[\ =, !\ ]\ 2$ i□ (by the points-to rules)
$ps@\ \ 2$.

Note that applying the substitution s to $p@$ and $p[\ =, !\ ]$ results in $ps@$ and $ps[\ =, !\ ]$. □

The downward Löwenheim–Skolem property follows. It should be noted that we cannot remove from the constructed model the binary relation symbols which are introduced as witnesses, as these determine the notion of first-order definability.

Theorem 3.2.4 (Completeness). *We have that* $\models$ *implies* $\vdash$.

Compactness follows. We thus derive (by Lindström's theorem [210]) that this interpretation of separation logic is as expressive as first-order logic.

3.3 Natural deduction

The sequent calculus introduced and proven sound and complete in the previous sections was defined in terms of three syntactic categories: the pure first-order formulas, the separation logic formulas, and the rooted formulas closed under propositional connectives and quantification. In this section, we investigate what happens when we consider only a single syntactic category of formulas: those of separation logic closed under the @-connective. We thus introduce the *extended separation logic* formulas by the following abstract grammar:

$$; \ ::= \ ? \mid (x \doteq y) \mid C(x_1, \ldots, x_n) \mid (\ \ !\ \ ) \mid (8x\ ) \mid (\ \ \ \ ) \mid (\ \ \ \ ) \mid (\ @\ )$$

The new @-connective can be understood as a binder of $, !$, in the sense that it lets the interpretation of determine the denotation of $, !$ with respect to which the formula is interpreted. Revisiting Definition A.1.6 (Free and bound variables), we need to add the following clauses:

- $FV(\ @\ ) = FV(\ ) \cup (FV(\ ) \cap \{x, y\})$, and
- $BV(\ @\ ) = BV(\ ) \cup BV(\ ) \cup \{x, y\}$.

By abuse of notation, we may think of @ as a *let binding* in the following sense:

$$(\ @\ ) = \text{let } , ! := (\ x\ y{:}\ \ ) \text{ in}$$

since the interpretation of $, !$ becomes 'bound' in by the let binding, and the free variables x and y in are 'captured' by the abstraction. Further, for technical convenience, we also have second-order binary variables $R; R_1; \ldots$, but it is not possible to quantify over such second-order variables.

We now give an extension of the natural deduction calculus for classical logic in the following way. The objects of this proof system, called RSL, are the above formulas of extended separation logic. Derivability in this proof system is denoted by $\vdash$. We have the usual axioms and proof rules of natural deduction, and add the axioms and proof rules of Figure 3.2.

We have the following example proofs using the above proof system.

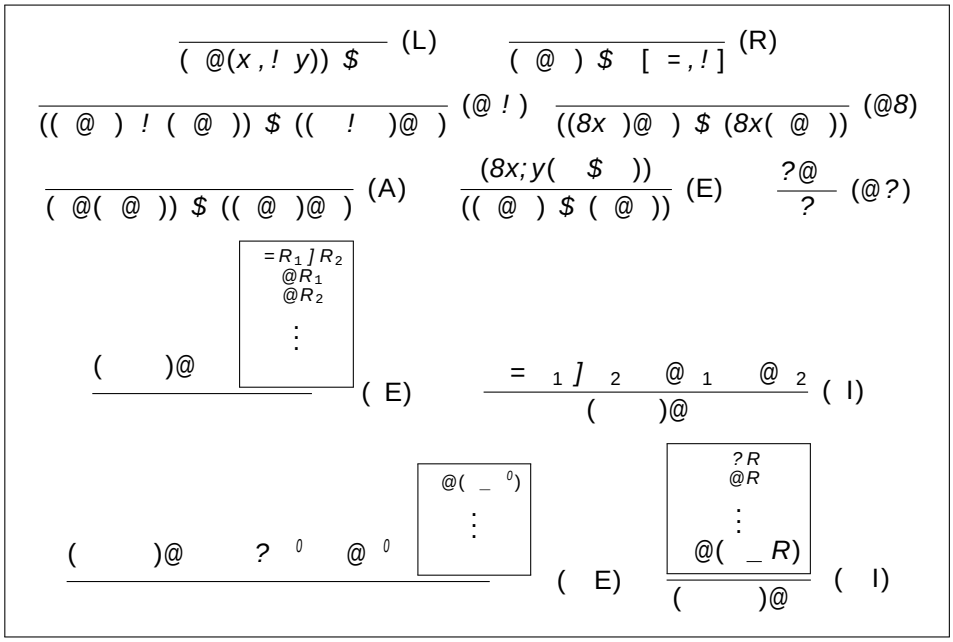


Figure 3.2: Natural deduction system for extended separation logic. In the rule (R) the formula is semi-pure. In the rule (E), $R_1; R_2$ do not occur in . In the rule (I), R does not occur in ; ; .

Proposition 3.3.1. $\vdash \mathbf{emp}@?$.

Proof. Recall that emp abbreviates $\forall x,y.\, \neg(x \circ y)$. We apply ($@\forall$) and ($\forall$I) twice, so now it suffices to show $\neg(x \circ y)@?$. The logical negation abbreviates $(x \circ y) \to ?$, so we apply ($@\to$) and by ($\to$I) we may assume $(x \circ y)@?$. From (R) and the new premise we infer $?$, and hence by ($?$E) we have $?@?$. □

Proposition 3.3.2. $\vdash \quad \to \quad$.

Proof. By (L), it suffices to show $(\quad \to \quad)@(x \circ y)$, and by ($@\to$) we may assume $(\quad)@(x \circ y)$ and show $(\quad)@(x \circ y)$. We do this by ($\circ$E), where we assume $(x \circ y) = R_1 \bullet R_2$ for fresh R_1, R_2, and $\quad@R_1$ and $\quad@R_2$. It is easy to see we also have $(x \circ y) = R_2 \bullet R_1$, and hence by ($\circ$I) we have $(\quad)@(x \circ y)$, completing the proof. □

Proposition 3.3.3. $\vdash \quad(\quad) \wedge (\quad)\quad$.

Proof. We have two directions (classical conjunction). We show $\quad(\quad) \to (\quad)\quad$ first. By (L) we wrap it under the trivial root, and by ($@\to$) we thus assume $(\quad(\quad))@(x \circ y)$. We use ($\circ$E) twice, to obtain $(x \circ y) = R_1 \bullet R_2$ and $R_2 = R_3 \bullet R_4$, so that $\quad@R_1$, $\quad@R_3$, $\quad@R_4$. We have $R_1 \# R_3$, so $R_1 \bullet R_3$ is defined. Further, we have $(R_1 \bullet R_3) \# R_4$ so $(R_1 \bullet R_3) \bullet R_4$ is also defined. The latter is equivalent to $(x \circ y)$. Now by ($\circ$I) twice, we obtain $(\quad)@(R_1 \bullet R_3)$ and $((\quad)\quad)@(x \circ y)$. The other direction goes in a similar way. □

Proposition 3.3.4. $\vdash (\mathbf{emp}@\quad) \wedge (\forall x,y.\quad \to ?)$.

Proof. Two classical directions:

- Assume $\mathbf{emp}@\quad$, and take arbitrary x_0, y_0 and assume $\quad(x_0 \circ y_0)$. We need to show $?$. Unfold the abbreviation emp and we have $(\forall x,y.\, \neg(x \circ y))@\quad$. Specializing this assumption with x_0 and y_0, we obtain $\neg(x_0 \circ y_0)@\quad$. To show $?$ it is sufficient to show $?@\quad$. We apply our assumption, so it suffices to show $(x_0 \circ y_0)@\quad$. But that follows from our assumption $\quad(x_0 \circ y_0)$.

- Assume $(\forall x,y.\quad \to ?)$. Unfold the abbreviation emp, and take arbitrary x_0, y_0, and assume $(x_0 \circ y_0)@\quad$. We need to show $?@\quad$, but it suffices to show $?$. From our assumption, we know $\quad(x_0 \circ y_0)$ holds. But that contradicts our earliest assumption. □

Proposition 3.3.5. $\vdash \quad \mathbf{emp} \wedge \quad$.

Proof. There are two directions (classically).

- We assume $(\quad \mathbf{emp})@(x \circ y)$ and need to show $\quad@(x \circ y)$. From our assumption we have $\quad@R_1$ and $\mathbf{emp}@R_2$ and $(x \circ y) = R_1 \bullet R_2$. Since $\mathbf{emp}@R_2$ we know $R_2 = ?$ (by previous proposition), and hence $(x \circ y) = R_1$. So by $\quad@R_1$ we then have $\quad@(x \circ y)$.

- We assume @(*x* ,! *y*) and need to show (emp)@(*x* ,! *y*). To show the latter it su□ces to show (*x* ,! *y*) = (*x* ,! *y*)] where = *?*. Clearly the disjoint union of those two relations is defined, and we already have (*x* ,! *y*). Also we have emp@ (by our previous proposition). □

Proposition 3.3.6. *The following holds:*

- ` (_) $ _ ,
- ` (^) ! ^ ,
- ` (9*x* (*x*)) $ 9*x*((*x*)),
- ` (8*x* (*x*)) ! 8*x*((*x*)),
- ` ( ) ! .

Proof. Left as exercises for the reader, as their proofs are not long. The proofs are also formalized, see Appendix D. □

Note that distributivity of conjunction (universal quantification) and separating conjunction only works in one direction.

Proposition 3.3.7. *The following holds:*

- ` ! ,
- ` ! (@),
- ` (!) ! @ ! @ ,
- ` (! ^{0}) ! (! ^{0}) ! ! ^{0} ^{0},
- ` (*x* ,! *y*) $ (*x* 7! *y*) >,
- ` :(*x* ,!) ! (((*x* 7! *y*) (*x* 7! *y*)) $).

Proof. Left as exercises for the reader, as their proofs are not long. The proofs are also formalized, see Appendix D. □

Proposition 3.3.8. *If* @ *is deducible for every* , *then* ` .

Proof. If @ is deducible for every heap description, then cannot depend on the heap and as such it holds in every heap. The proof is formalized, see Appendix D. □

Proposition 3.3.9. *If* $ *is deducible, and we have a deduction of* *from premises* *then we may replace any occurrence of* *by* *in any of the premises in* *and the conclusion* .

We again investigate the weakest precondition of the postcondition (*v* ,! *z*) and the program [*u*] := 0. As before, let *p* denote the weakest precondition (*u* ,!) ^ (*z* = 0 /*u* = *v* . *v* ,! *z*), where again /*b*. abbreviates (*b* ^) _ (:*b* ^). Let p^{0} denote the weakest precondition (*u* 7!) (*u* 7! 0 *v* ,! *z*).

Proposition 3.3.10. $\vdash p \Rightarrow p^0$.

Proof. The proof can be formalized, see Appendix D. □

Lemma 3.3.11. $\vdash p^0 \Rightarrow p$.

Proof. The proof can be formalized, see Appendix D. □

Note that it is not needed to assume functionality of the heap (since the separating implication speaks of all disjoint relational heaps, including those that satisfy functionality).

3.4 Soundness and completeness

We shall give a general relational semantics to these extended separation logic formulas, but to do so we need to construct the satisfaction relation in two stages.

Definition 3.4.1. A *general relational structure* $\mathsf{H} = (\mathsf{A}; H)$ consists of a structure $\mathsf{A} = (A; I)$ with domain A and a set of binary relations $H \subseteq \mathcal{P}(A \times A)$.

In the first stage, we give a general relational semantics GRSL which is suitable for interpreting the let binding. We shall use valuations that assign both the first-order and binary second-order variables. The binary second-order variables are not constrained and ranges over *arbitrary* binary relations between elements of the domain of the underlying structure. Also the relation R is not constrained and ranges over arbitrary relations, whereas in the interpretation of the separating connectives quantification *is* restricted to the set of relations of the general relational structure.

Definition 3.4.2 (Satisfaction relation). Given a general relational structure $\mathsf{H} = (\mathsf{A}; H)$ with domain A and interpretation I, a valuation of A, a binary relation $R \subseteq A \times A$, and an extended separation logic formula . The satisfaction relation $\mathsf{H}; R; \models^{\mathbf{GRSL}}$ is defined inductively on the structure of :

- $\mathsf{H}; R; \models^{\mathbf{GRSL}} \bot$ never holds,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} (x \doteq y)$ iff $(x) = (y)$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} (x, y)$ iff $((x); (y)) \in R$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} R(x_1; x_2)$ iff $((x_1); (x_2)) \in (R)$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} C(x_1; \ldots; x_n)$ iff $((x_1); \ldots; (x_n)) \in C^I$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} \Rightarrow$ iff $\mathsf{H}; R; \models^{\mathbf{GRSL}}$ implies $\mathsf{H}; R; \models^{\mathbf{GRSL}}$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}} \forall x$ iff $\mathsf{H}; R; [x := a] \models^{\mathbf{GRSL}}$ for every $a \in A$,
- $\mathsf{H}; R; \models^{\mathbf{GRSL}}$ iff $\mathsf{H}; R_1; \models^{\mathbf{GRSL}}$ and $\mathsf{H}; R_2; \models^{\mathbf{GRSL}}$ for some $R_1; R_2 \in H$ such that $R = R_1 \cup R_2$ and $R_1 \perp R_2$,

- H; R; j=GRSL i H; R'; j=GRSL implies H; R [R'; j=GRSL for every R' 2 H such that R ? R',
- H; R; j=GRSL @ i H; R'; j=GRSL for R' = Rel H; R; ( ).

where Rel H; R; ( ) denotes fhd_x; d_y i j H; R; [x; y := d_x; d_y] j=GRSL g A A.

Note that if one takes H to be the set of all finite relations and restrict to the (non-extended) formulas of separation logic, we obtain weak relational separation logic, and similarly if one takes H to be the set of all relations, we obtain full relational separation logic.

For the second stage, we define the following class of general relational structures. This class captures *semantic comprehension* by means of a closure condition on the set of relations, that constraints the range of second-order quantifiers implicitly used for giving semantics to the separating connectives, in the sense that every binary relation that can be expressed by an extended formula of separation logic must be in the set of binary relations of the general structure too.

Definition 3.4.3. A *comprehensive relational structure* H = (A; H) is a general relational structure such that for every relation R 2 H, valuation of A where (R) 2 H for every second-order variable R, and extended formula of separation logic , we have Rel H; R; ( ) 2 H.

We then define our intended semantics as follows.

Definition 3.4.4 (Satisfaction relation). Given a comprehensive relational structure H = (A; H), a relation R 2 H, and valuation of A where (R) 2 H for every second-order variable R, we define the satisfaction relation H; R; j=RSL with the same conditions as given before in Definition 3.4.2.

Notice how in this satisfaction relation, compared to the previous stage, the relation R and the value of second-order variables are constrained to be in H. Since the semantic comprehension condition imposed on comprehensive relational structures is expressed using the first stage semantics, there is no circularity in the condition that R (and the value of any second-order variable) needs to be in H.

Again, note that if one takes H to be the set of all finite relations, to obtain weak relational separation logic, we may fail to make a comprehensive relational structure out of it: there is a formula, such as >, that express that infinitely many locations are related to a value, but that contradicts the requirement that we restrict to finite relations. There is no problem for structures with finite domain, since there weak relational separation logic and full relational separation logic coincide. If one takes H to be the set of all relations on a structure with infinite domain, we obtain full relational separation logic, which is also trivially a comprehensive relational structure. It does seem possible to construct a comprehensive relational structure out of a set H consisting of all finite and cofinite relations, but we leave that structure for the reader to investigate further.

From the definition above, we can see that the formula in the let binding (@) is a type with free variables among x and y. In particular, we have the properties (L)eft-root, (R)ight-root, (A)ssociative-root, and (E)quivalent-root:

Lemma 3.4.5 (Soundness I).

(L) $H; R; \; \models^{\mathbf{RSL}} (\; @(x, !\; y)) \; \$ \;\; ,$

(R) $H; R; \; \models^{\mathbf{RSL}} (\; @\;\;) \; \$ \;\; [\;=, !\;]$ *where* $\;\;$ *is semi-pure,*

(A) $H; R; \; \models^{\mathbf{RSL}} (\; @(\;\; @\;\;)) \; \$ \; ((\;\; @\;\;)@\;\;),$

(E) $H; R; \; \models^{\mathbf{RSL}} (8x; y(\;\; \$ \;\;)) \; ! \; ((\;\; @\;\;) \; \$ \; (\;\; @\;\;)).$

Proof. (L) Suppose $H; R; \; \models^{\mathbf{RSL}} (\; @(x, !\; y))$ holds, then we know that also $H; R; \; \models^{\mathbf{RSL}} \;\;$ holds, since $Rel_{H;R;} \; (x, !\; y) = R$. The converse is similar.

(R) Suppose $H; R; \; \models^{\mathbf{RSL}} (\; @\;\;)$ holds, then by definition we know that also $H; R^0; \; \models^{\mathbf{RSL}} \;\;$ holds for $Rel_{H;R;} \; (\;\;) = R^0$. Since $\;\;$ is semi-pure, in the evaluation of $\;\;$ we never change R^0. Hence we can replace $(z, !\; w)$ by $(z; w)$ in $\;\;$ and we have that $H; R; \; \models^{\mathbf{RSL}} \;\; [\;=, !\;]$ holds. Note how the free variables of $\;\;$ (other than $x; y$ which are replaced by the variables $z; w$) are still evaluated with respect to $\;\;$. The converse is similar.

(A) Suppose $H; R; \; \models^{\mathbf{RSL}} (\; @(\;\; @\;\;))$ holds, then we know that $H; R^0; \; \models^{\mathbf{RSL}} \;\;$ holds for $Rel_{H;R;} \; (\;\; @\;\;) = R^0$. We then also know that for every pair $\hbar d_x; d_y i \; 2 \; R^0$ we have that $H; R^{00}; \; [x := d_x; y := d_y] \; \models^{\mathbf{RSL}} \;\;$ where we take $Rel_{H;R; \; [x:=d_x; y:=d_y]}(\;\;) = R^{00}$. Note that we have $R^{00} = Rel_{H;R;} \; (\;\;)$, since x and y are bound, and thus we have $H; R^{00}; \; \models^{\mathbf{RSL}} \; @\;\;$ since we have that $H; R^{000}; \; \models^{\mathbf{RSL}} \;\;$ where $Rel_{H;R^{00};} \; (\;\;) = R^{000}$, from $H; R^0; \; \models^{\mathbf{RSL}} \;\;$ and knowing that $R^0 = R^{000}$.

(E) Similar to the cases before. □

The following properties describe the interactions between connectives:

Lemma 3.4.6 (Soundness II).

(@?) $H; R; \; \models^{\mathbf{RSL}} (?@\;\;) \; ! \; ?,$

(@ !) $H; R; \; \models^{\mathbf{RSL}} ((\;\; @\;\;) \; ! \; (\;\; @\;\;)) \; \$ \; ((\;\; ! \;\;)@\;\;),$

(@8) $H; R; \; \models^{\mathbf{RSL}} ((8x\;\;)@\;\;) \; \$ \; (8x(\;\; @\;\;))$ *where* x *is not free in* $\;\;,$

(E) $H; R; \; \models^{\mathbf{RSL}} ((\;\;\;\;)@\;\;) \wedge (\;\; = R_1 \;] \; R_2 \wedge (\;\; @R_1) \wedge (\;\; @R_2) \; ! \;\;) \; !$ *where* $R_1; R_2$ *do not occur in* $\;\;,$

(I) $H; R; \; \models^{\mathbf{RSL}} \;\; = \;\; _1 \;] \;\; _2 \wedge (\;\; @\;\; _1) \wedge (\;\; @\;\; _2) \; ! \; ((\;\;\;\;)@\;\;),$

(E) $H; R; \; \models^{\mathbf{RSL}} ((\;\;\;\;)@\;\;) \wedge \;\; ? \;\; ^0 \wedge (\;\; @\;\; ^0) \wedge ((\;\; @(\;\; \_ \;\; ^0) \; ! \;\;) \; ! \;\;) \; ! \;\;,$

(I) $H; R; \; \models^{\mathbf{RSL}} (\;\; ? \; R \wedge (\;\; @R) \; ! \; (\;\; @(\;\; \_ R))) \; ! \; ((\;\;\;\;)@\;\;)$ *where* R *does not occur in* $\;\; ; \;\; ; \;\; .$

We also have the following derived properties:

Corollary 3.4.7.

- $A; R;\ j=^{\mathbf{RSL}} (\ @\ ) \$\ $ *where* $\ $ *is a pure formula,*
- $A; R;\ j=^{\mathbf{RSL}} ((\ @\ ) \wedge (\ @\ )) \$ ((\ \wedge\ )@\ )$,
- $A; R;\ j=^{\mathbf{RSL}} ((\ @\ ) \_ (\ @\ )) \$ ((\ \_\ )@\ )$,
- $A; R;\ j=^{\mathbf{RSL}} ((9x\ )@\ ) \$ (9x(\ @\ ))$ *where* x *is not free in* $\ ,$
- $A; R;\ j=^{\mathbf{RSL}}\ \ !\ (\ @\ )$.

The proof system RSL is sound with respect to the semantics RSL.

Lemma 3.4.8 (Soundness). $\ \vdash^{\mathbf{RSL}}\ $ *implies* $\ j=^{\mathbf{RSL}}\ .$

Proof. By induction on the structure of a deduction. Note that the semantics of RSL follows that of classical logic for all logical connectives, hence the proof rules involving classical connectives are sound via their usual argument. For the additional axioms and proof rules, see Lemma 3.4.5 and Lemma 3.4.6. □

We now investigate a proof reduction technique. Every deduction in the natural deduction proof system can be reduced to a deduction with only rooted formulas of a particular shape, by introducing additional fresh binary variables. The shape of rooted formulas we wish to obtain are precisely those that can be worked with in our previous sequent calculus, i.e. rooted assertions with a pure right-side. The purpose of the procedure is as follows. Suppose we are given a set of premises $\ $ and a conclusion $\ $. Our goal is to obtain an equisatisfiable set of premises $\ ^{0}$ and conclusion $\ ^{0}$ in which every occurrence of a rooted formula does not have any roots occurring the left, has a first-order formula on the right, and is not nested under separating connectives. Such equi-satisfiable set of premises then allows us to obtain a proof using our previous sequent calculus, and that proof is straightforwardly mapped to a proof in natural deduction.

We sketch out the following provability-preserving and semantics-preserving operations on the premises and conclusion:

1. For all formula occurrences $\ @\ $ that are nested on the left under a top-level root $(:::(\ @\ ):::)@\ $, we 'push down' the outer root until it reaches the nested root, and we perform an associative root swap so that from $(:::(\ @\ ):::)@\ $ we obtain $(:::(\ @(\ @\ )):::)$. For the classical connectives this 'pushing down' is straightforward. For $(x\ ,!\ y)$, we can simply substitute using the right root rule. For an occurrence that is a separating conjunction $(\ _1\ \ _2)@\ $ we introduce fresh binary variables $R_1; R_2$, replace the occurrence with $(\ _1@R_1) \wedge (\ _2@R_2)$, add the premise $\ = R_1\ ]\ R_2$, and proceed with pushing down in the occurrences $\ _1@R_1$ and $\ _2@R_2$. A similar construction happens for separating implication, but we leave one fresh variable open for interpretation. We repeat this step until no longer we have roots nested under the left.

2. For each formula occurrence @ that does not occur on the left under a root and where is not first-order, in some formula of or in the conclusion , we introduce a fresh binary variable R. We add a premise on the top level $(\forall x, y : R(x, y) \mathbin{\$})$, and replace in the occurrence @ by $R(x, y)$.

3. For any rooted formula @ that occurs under a separating connective, we dissolve the separating connective in a similar matter as described above.

Now we can show completeness of the natural deduction proof system by reduction to a completeness theorem of the sequent calculus (that is similar to the proof in Section 3.2, but slightly different due to its different semantics).

Lemma 3.4.9 (Completeness). $\models^{\mathbf{RSL}}$ *implies* $\vdash^{\mathbf{RSL}}$.

Proof. The proof goes along the following lines, and mostly uses standard techniques from interpretational proof theory [212]. We adapt the premises and conclusion in an equisatisfiable way, as sketched out above. We then obtain a sequent $'$) $'$ for which, by the completeness result of the sequent calculus established previously (but adapted to the new semantics), we can obtain a deduction. Every deduction in sequent calculus can be mapped to a deduction in natural deduction. The operations to obtain the adapted premises and conclusion can be reversed to obtain a proof of the original conclusion with the original premises in the natural deduction proof system. □

We gloss over the details comparing the semantics RSL and FORSL. However, these details are not essential to the completeness result above: it is also possible to prove the completeness of the proof system RSL directly, by replicating much of the work done previously to show completeness of the sequent calculus (the model existence theorem): again by constructing a maximal consistent set of formulas of (extended) separation logic out of a given set of formulas, and constructing a model out of it to show the satisfiability of the given set of formulas. After doing such a direct proof of completeness, one also establishes a relation between the two semantics.

3.5 Discussion

One may think of relational separation logic to be an abstraction of (functional) separation logic in the following sense: suppose, in an object-oriented setting, we would have a functional 'points to' relation for each field of an object. In the abstract view of (one-step) reachability, it does not matter by which field an object points to another object, what only matters is that another object is reachable through *some* field. Reachability is thus modeled as a points-to relation that is not necessarily functional, and interpreting the separating conjunction thus involves a partition of objects. In particular, we have that the formula $(x \mathbin{,} ! \ ) \ (x \mathbin{,} ! \ )$ should be equivalent to false, because an object x cannot be in both separate parts at the same time. With the condition on the disjointedness of the domains of R_1 and R_2 this equivalence indeed holds.

However, and contrary to our intuition of separation, it is possible to satisfy $(x \mapsto \quad) * (x \mapsto \quad)$ if we merely require the relations R_1 and R_2 to be disjoint (since one part can assign the location x to a different value than the other part). But then what does *separate* mean if an object x can be in both separate parts at the same time?

We discuss the consequence of the fact that in relational separation logic the points-to relation is no longer functional. We previously have seen the following two concepts:

- We have that the primitive formula $(x \mapsto y)$ expresses 'x points to y' or 'location x has value y'. In the relational setting, we no longer have that if $(x \mapsto y)$ holds that y is the *only* value that x points to, since it is possible that there are other values that x points to as well.
- We have that $(x \hookrightarrow y)$ abbreviates $((x \mapsto y) \wedge \forall z, w((z \mapsto w) \rightarrow x = z))$, which expresses that '$x$ strictly points to y' or 'x points to y and only x is allocated'. Similarly, in the relational setting, we also no longer have that if $(x \hookrightarrow y)$ holds that y is the *only* value that x points to, since it is also possible that there are other values that the location x points to. However, we do have that x is the only allocated location.

In the relational setting, that a location points to a value does not necessarily mean that this location points to only one value. Thus it is warranted that we introduce the following abbreviations:

$$(x \mapsto^* y) \text{ abbreviates } ((x \mapsto y) \wedge \forall z((x \mapsto z) \rightarrow y = z))$$
$$(x \hookrightarrow^* y) \text{ abbreviates } ((x \mapsto y) \wedge \forall z, w((z \mapsto w) \rightarrow x = z \wedge y = w))$$

where z is a fresh variable. We speak about these formulas in the following way:

- for $(x \mapsto y)$ we say 'x points to y' or 'location x has value y',
- for $(x \mapsto^* y)$ we say 'x points to y alone' or 'location x has (and only has) value y',
- for $(x \hookrightarrow y)$ we say 'strictly x points to y' or 'x points to y and only x is allocated',
- for $(x \hookrightarrow^* y)$ we say 'strictly x points to y alone' or 'x points to y alone and only x is allocated' or 'the one and only location-value pair is (x, y)'.

Strictness (resp. looseness) indicates exactly one (resp. at least one) location on the heap, and narrowness (resp. wideness) indicates precisely one (resp. at least one) value is associated to that location. The four points-to relations can be systematized as in Table 3.1. The following sentences are valid:

$$\forall x, y.\ (x \hookrightarrow^* y) \dashv\vdash (x \hookrightarrow y) \wedge (x \mapsto^* y);$$
$$\forall x, y.\ (x \hookrightarrow y) \sqcup (x \mapsto^* y) \rightarrow (x \mapsto y);$$

	Narrow	Wide
Strict	$(x \mapsto^* y)$	$(x \mapsto^! y)$
Loose	$(x ,^* y)$	$(x ,^! y)$

Table 3.1: The four points-to relations.

and $(x \mapsto^! y)$ and $(x ,^* y)$ are themselves incomparable. If we have

$$\forall x, y.\ (x ,^! y) \mathrel{-\!\!*} (x ,^* y)$$

then the points-to relation must be functional. This is the case in (functional) separation logic as introduced in the previous chapter, but no longer for relational separation logic. The latter formula is equivalent to $\mathit{fun}(,^!)$.

In this chapter we have investigated relational separation logic, but how much work does it take to adapt the semantics and the proof system to (functional) separation logic?[1] Both the semantics and the proof system of relational separation logic rely on the fact that we can express relations using arbitrary binary formulas. We can not simply use the proof system but restrict to functional interpretations of the binary variables: the problem lies in that rooted assertions $p@$ allow any binary formula , which may denote non-functional relations as well. And the same problem happens when considering comprehensive relational structures. However, without a lot of effort we can overcome this problem, by introducing additional notational conventions, obligations, and assumptions.

Similar to how terms can be added to a first-order logic that only has constant symbols such as predicate and relation symbols, by *declaring* constant symbols as individual symbols and function symbols, we can also keep track of a subclass of binary formulas for which we declare the property of functionality holds. By writing such binary formulas $^\wedge$, to mean that must be functional, then we can keep track for which formulas we have additional obligations to show functionality, or assumptions that witness their functionality.

We can then adapt the proof system RSL to obtain the proof system SL: additional proof obligations are required for the introduction rule of separating conjunction (because the disjoint union of two functional relations is not necessarily functional) and the elimination rule of separating implication (where also the disjoint union is not necessarily functional). In the case of the elimination rule of separating conjunction, we already know that splitting a functional relation always results in two functional relations, leading to additional assumptions. In the case of the introduction rule of separating implication we can add functionality (of the relation representing the extension, and of the disjoint union of the outer heap and the extension) as an additional assumptions.

[1] Many respectable colleagues have told the author that 'nobody reads Ph.D. theses'—in the interest of their reputation it is best to leave them anonymous. The full description of the proof system **SL** and its soundness and completeness proof remain to be published in a forthcoming journal article. However, from the sketch provided here, it is not difficult for a reader to come up with it themselves.

Adapting GRSL to GSL involves restricting to partial functions h instead of relations R, to obtain general heap structures. We then have the set $Fun_{H;h;}$ (^) that denotes a partial function based on the formula for which we know it has the property of functionality. Consequently, we can consider heap structures closed under semantic comprehension (for a restricted class of formulas, namely those that define a functional relation) in a similar way as before, to obtain SL.

Finally, notice how in the rooted assertion @ the @-connective is related to the binding operator (#R) of the previous chapter, by comparing their semantics:

- A$;R;$ $\models^{\mathbf{FRSL}^{\#}}$ (#R) if and only if A$;R;$ [R := R] $\models^{\mathbf{FRSL}^{\#}}$,
- H$;R;$ $\models^{\mathbf{GRSL}}$ @ i H$;R^{0};$ $\models^{\mathbf{GRSL}}$ for $R^{0} = Rel_{H;R;}$ ( ).

In some sense, the binding operator 'captures' the current interpretation of R, whereas in the interpretation of the @-connective we replace the current interpretation of R. The connection is interesting from the perspective of Henkin models of second-order logic, which satisfy a comprehension axiom, by which we know that every formula also denotes a relation over which one can quantify. If we would add second-order variables to GRSL, the connection may become more obvious:

- A$;R;$ $\models^{\mathbf{FRSL}^{\#}}$ (#R) if and only if A$;R;$ [R := R] $\models^{\mathbf{FRSL}^{\#}}$,
- H$;R;$ $\models^{\mathbf{GRSL}}$ @R if and only if H$;$ (R)$;$ $\models^{\mathbf{GRSL}}$.

Chapter 4

Reynolds' logic

In Chapter B, we recall Hoare's logic for reasoning about while-programs and recursive programs. In this chapter, we investigate Reynolds' logic, an extension of Hoare's logic for reasoning about pointer programs. We introduce a novel semantics and several original proof systems for Reynolds' logic, all interpreted with respect to the same semantics: it is possible to reinterpret the original proof systems in the novel semantics. We then introduce an extension to dynamic logic, called dynamic separation logic, and show how to simplify program modalities. This yields the discovery of an alternative proof system of Reynolds' logic.

What is remarkable is that all the proof systems in this chapter are *equivalent* in the following sense: the original proof system proposed by Reynolds and all other proof systems in this chapter have *exactly the same set of provable objects*! This justifies us to call all these proof systems which have a single common semantics 'Reynolds' logic', in honor of J.C. Reynolds. But, why an alternative proof system for Reynolds' logic? It would be fair to say, that the alternative proof systems sheds light on the same matter from a di□erent angle. In fact, what distinguishes the new proof systems (both the weakest precondition calculus and the strongest postcondition calculus) from the original ones is the property of gracefulness. Gracefulness means that the weakest precondition of any statement and a first-order formula remains a first-order formula, and similar for the strongest postcondition. Furthermore, by the techniques developed in this chapter, we are able to fill in a missing gap of proving that the global axioms can be derived from the local axioms and the frame rule without using the *magic wand*, the connective of separating implication.

Reynolds' logic can be seen as an extension of Hoare's logic in two ways:

- In Hoare's logic the programming language is based on a first-order program signature, but in Reynolds' logic the programming language is based on a pointer program signature. Moreover, the proof objects of the two proof systems are specifications $f\ g\ S\ f\ g$. In Hoare's logic the formulas and are first-order formulas, whereas in Reynolds' logic these formulas are further extended to the formulas of separation logic.

- Since every formula of first-order logic is *also* a formula of separation logic, all the instances of the axioms and rules of Hoare's logic can also be considered part of Reynolds' logic.

However, the second point raises the question: how much rules of Hoare's logic remain sound also when extending the instances to formulas of separation logic?

If we consider the invariance rule

$$\frac{f \; g \; S \; f \; g}{f \;\wedge\; g \; S \; f \;\wedge\; g}$$

of Hoare's logic, where the free variables of are not changed by the statement S, then clearly we cannot extend this rule to arbitrary formulas of separation logic: the rule would become unsound. The problem here lies in the fact that the program S may change a location on the heap, but it is not easily recognized from the syntax of the program what location has changed. This is in contrast to the invariance rule, where an approximation of the effects of a statement in Hoare's logic is captured by $access(S)$ and $change(S)$.

An important proof rule that is novel in Reynolds' logic, that is not part of Hoare's logic, is the so-called *frame rule*

$$\frac{f \; g \; S \; f \; g}{f \quad g \; S \; f \quad g}$$

where the free variables of are not changed by the statement S. Note that in this rule we use separating conjunction instead of logical conjunction. However, it turns out that the soundness of the frame rule is quite delicate [232]. In this chapter we shall revisit the soundness proof of the frame rule, and give an alternative proof that is more proof theoretic in nature.

As one can recall in Chapter B, due to the compositional nature of the semantics of programs, it suffices here to restrict our attention to the base case, the pointer manipulating operations, and not to the control structures of sequential composition, if and while-statements, or recursive procedures, since these latter constructs of the programming language are orthogonal to our current concerns. In fact, already in [125], Ishtiaq and O'Hearn recognize that the novelty of Reynolds' logic lies in the treatment of the basic operations:

> "We will not give a full syntax of [statements], as the treatment of conditionals and looping statements is standard. Instead, we will concentrate on assignment statements, which is where the main novelty of the approach lies."

It is the case that our previous discussion concerning the compositional nature of program semantics naturally transfers to the context of pointer programs. However it is important to note, in light of the previous discussion about the invariance rule, that Hoare's proof rules involving complex statements remain sound even when the instances of formulas range over the formulas of separation logic.

To see why we first need to introduce the proper semantic basis in which we can interpret the proof objects of Reynolds' logic. Although the semantics is a generalization of the standard semantics of separation logic, it is quite remarkable to observe that the original axiomatization due to Reynolds is still sound with respect to this general semantics.

Secondly, we present Reynolds' logic in its usual way. There are four ways in which the proof system of Reynolds' logic can be presented: a *local* axiomatization, and a *global* axiomatization, a *backwards* weakest precondition axiomatization, and a *forwards* strongest postcondition axiomatization. In the first proof system with the *local* axiomatization, however, the frame rule plays a crucial role. One can recover the *global* axiomatization from the *local* axiomatization by using the frame rule. Our only contribution here, is that we shall argue there is an alternative way of proving the soundness of the frame rule, from a proof theoretical point of view, rather than the 'surprisingly delicate' semantic point of view as done by Yang and O'Hearn [232].

Third, we show the soundness and (relative) completeness of a novel weakest precondition axiomatization, by introducing an extension to separation logic, called *dynamic separation logic*, in which a logical modality is added that captures the weakest precondition. We introduce pseudo-operations corresponding to *heap update* and *heap clear*. It is crucial that these pseudo-operations are *not* part of the programming language, and for both of them we show they satisfy a useful meta-property that we capture by proving corresponding *substitution lemmas*. That these pseudo-operations are not part of the programming language is crucial, since the frame rule is not sound with respect to them. However, by introducing the pseudo-operations on the logical level, and thereby not allowing the application of the frame rule over them, we obtain a way of expressing the weakest precondition of all other basic instructions in terms of these pseudo-instructions. In fact, the pseudo-instructions satisfy a property called *gracefulness*, in the sense that computing the weakest precondition of a pseudo-instruction with respect to a first-order postcondition results in a first-order weakest precondition. Even stronger, computing the weakest precondition of a pseudo-instruction with respect to the fragment of separation logic without magic wand also results in a formula without magic wand: thus eliminating magic wand from the generated weakest precondition. It turns out that all the previous axiomatizations of Reynolds' logic lack this property of gracefulness.

Fourth, we recall two strongest postcondition axiomatizations of Reynolds' logic, again a *local* strongest postcondition axiomatization and a *global* strongest post-condition axiomatization. We then also introduce a novel strongest postcondition axiomatization by again using our previously introduced pseudo-instructions, and we compare our alternative axiomatization with the two existing calculi.

In this chapter, we focus on classical separation logic as assertion language. In Appendix C, our approach is also extended to intuitionistic separation logic, also resulting in novel weakest preconditions and strongest postconditions. This shows that our approach, of introducing pseudo-instructions, is robust under di□erent interpretations.

4.1 General semantics and memory models

In the Sections 2.2 and 2.3 we have established that both the standard semantics and the full semantics of separation logic are not compact, leading to the impossibility to have a complete finitary proof system. In this section we investigate two more general semantics for separation logic. We introduce *general heap structures*, which extends structures to include a set of heaps over which quantification in the semantics of the separating connectives ranges. This approach is similar to the general structures of Henkin semantics [108]. Although this semantics is sufficient for interpreting separation logic formula, the set of heaps does not necessarily contain all heaps needed to reason about memory modification effects of pointer programs. Therefore, we also introduce the *heap semantics*, in which we extend structures with a particular set of heaps called a *memory model*. We further show the relation between the semantics based on memory models and the standard semantics, full semantics, and novel semantics of separation logic for which we obtained a sound and complete, finitary proof system in Section 3.2 and Section 3.4.

Given a structure $\mathsf{A} = (A; I)$ with domain A, and let H be a set of heaps (partial functions from A to A) with $h; h_1; h_2 \in H$. Recall that we can express the partitioning of heaps by $h = h_1 \bullet h_2$ which satisfies the following properties:

$$h = h_1 \bullet h_2 \Rightarrow \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset; \tag{4.1}$$

$$h = h_1 \bullet h_2 \Rightarrow h(a) = \begin{cases} h_1(a) & \text{if } a \in \mathrm{dom}(h_1); \\ h_2(a) & \text{if } a \in \mathrm{dom}(h_2); \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{4.2}$$

A *general heap structure* (for separation logic) can be used to give semantics to separating connectives, where only the heaps in a given set of heaps are considered. Our approach here is similar to Henkin's general heap structures for higher-order logic, where quantification over values of higher-order arities is restricted to range over a given set of values.

Definition 4.1.1 (General heap structures). A *general heap structure* $\mathsf{H} = (\mathsf{A}; H)$ consists of a structure $\mathsf{A} = (A; I)$ with domain A and interpretation I and a set of heaps H of partial functions from A to A.

A general heap structure thus includes a set of possible heaps, and in the general semantics one limits quantification in the semantics of the separating conjunction and separating implication to the given set of heaps. The satisfaction relation over general heap structures can now be given. It is similar to Definition 2.2.1, except for the following clauses that are relative to the given set of heaps.

Definition 4.1.2 (Satisfaction relation). Given a general heap structure $\mathsf{H} = (\mathsf{A}; H)$, a valuation $\ldots$ of A, a heap h of H, and a separation logic formula $\ldots$. The satisfaction relation $\mathsf{H}; h; \ldots \models \ldots$ is defined inductively on the structure of $\ldots$:

- $\ldots$

- H; h; j=^{GSL} i H; h_1; j= and H; h_2; j= for some $h_1; h_2 \in H$ such that $h = h_1 \bullet h_2$,
- H; h; j=^{GSL} i A; h'; j= implies A; h''; j= for every $h'; h'' \in H$ such that $h'' = h \bullet h'$.

The superscript GSL on j= stands for General Separation Logic. Note that in general heap structures H = (A; H) we have that H can be empty, similar to the situation of an empty domain in the classical semantics. However, in the context of the satisfaction relation, we may assume the set of heaps is non-empty since h is a heap in H. This situation is similar to the situation in classical logic where the domain must be non-empty, since a valuation assigns values to variables.

Similar to before, we have the coincidence condition and invariance under renaming. Both propositions are with respect to a fixed heap in our memory model.

Proposition 4.1.3 (Coincidence condition). *Given that* [FV( )] = ^{0}[FV( )]*, it follows that* H; h; j=^{GSL} *if and only if* H; h; ^{0} j=^{GSL} .

Proposition 4.1.4 (Invariance under renaming). *Given a renaming such that all free variables of stay the same, i.e.* (v) = v *for all* v 2 FV( )*. It follows that* H; h; j=^{GSL} *if and only if* H; h; j=^{GSL} ( )*.*

Definition 4.1.5 (Denotation). The *denotation* of a formula H K^{GSL} is defined:

$$H\ K^{GSL} = \{(h;) \mid H; h; \models^{GSL} \}$$

We write H; h j=^{GSL} to mean H; h; j=^{GSL} for all valuations of A, and we write H j=^{GSL} to mean A; h j=^{GSL} for all heaps h 2 H. Given a sentence that is satisfied, using the coincidence condition we can obtain that it is also satisfied by the same general heap structure but with any other valuation: the valuation has no influence on whether a sentence is satisfied by the structure, but the heap does have such influence (as it does with standard and full semantics). So if is a sentence, H; h j=^{GSL} if and only if H; h; j=^{GSL} for some valuation .

Given a sentence , we write j=^{GSL} to mean that H j=^{GSL} for all gneral structures H, and we then say that is *valid*. Valid sentences in general separation logic thus are properties that hold for all general heap structures.

An interesting instance is the general heap structure (A; f g) where is the partial function that is never defined anywhere. This general heap structure clearly satisfies the formula emp. Further, since we can split the empty heap only in two empty heaps, we have that separating conjunction is equivalent to logical conjunction. For a similar reason we have that separating implication is equivalent to logical implication. Thus, any valid classical formula in which we replace (some) logical conjunction and implication connectives by separating conjunction and implication, respectively, is a valid separation logic formula with respect to the general semantics. Note that this works since classical formulas do not have any occurrence of a points-to relation, since the points-to relation is only available to separation logic and not present in the signature.

A well-known example of a valid formula in the general semantics is

$$\models^{\mathbf{GSL}} (\quad (\quad)) \quad !$$

where and are arbitrary separation logic formulas. It is easily verified that this formula is indeed valid: *if* a split is possible, then we can always recombine the two separate parts of the heap by the separating implication to obtain the succedent of the separating implication.

However, in the general semantics, there are formulas which are not valid, but which are valid in both the standard and the full semantics of separation logic. Take the formulas $(x \mapsto y)$ and $(x \mapsto y)$ true. These formulas are equivalent in both WSL and FSL. While it is the case that

$$\models^{\mathbf{GSL}} \forall x, y. ((x \mapsto y) \quad \mathbf{true}) \quad ! \quad (x \mapsto y);$$

the converse does not hold. Take the general heap structure H = (A; H) with the domain A of A having at least two elements, and H being the set $\{\emptyset, h\}$ where $h(a) = a$ for some $a \in A$, and $h(b) = b$ for some other $b \in A$, and undefined on all other elements. Now we have H; h; $\models^{\mathbf{GSL}} (x \mapsto y)$ where assigns x and y to a. However, it is not the case that H; h; $\models^{\mathbf{GSL}} (x \mapsto y)$ true, since H contains the heap h but not the two subheaps making the split possible.

Another counter-example is the following implication (for any formula):

$$(x \quad) \wedge ((x \mapsto) \quad ((x \mapsto) \quad)) \quad !$$

which is valid in both WSL and FSL (in WSL we do not even need $(x \quad)$). However, this fails for GSL. We now take $H = \{\emptyset, g\}$. Now, clearly, both a and b are not allocated in every heap. Take a valuation in which x has value a, and y has the value b. We could for example let express that y is allocated: $(y \mapsto)$. The antecedent of the implication is satisfied in the empty heap: x is not allocated, and for every heap in which x is allocated the succedent of the separating implication holds for the joined heap (vacuously). However, is not satisfied, since y is not allocated.

The counter-examples above demonstrates two issues with the general semantics. We expect certain closure conditions to hold for the set H to be able to naturally reason about the semantics of separation logic (stated informally):

- we expect that emp and are equivalent, so $\in H$;
- we expect that $(x \mapsto y)$ true and $(x \mapsto y)$ are equivalent, but for that to work we need that splitting off finite parts from any heap in H is also in H;
- under the assumption that there is some free space, that is $(x \quad)$, we expect that $(x \mapsto)$ $((x \mapsto)$) and are equivalent, but for that to work every finite extension of any heap in H must be in H too.

We thus consider sets of heaps which satisfy certain closure conditions, called *memory models*. A memory model is a set of heaps that is closed under the operations of heap update and heap clear, and closed under heap existence conditions.

The combination of heap update and heap clear operations allows us to express finite splits and finite extension as required. For example, for any heap h where $a \in \mathrm{dom}(h)$ we can apply the heap update $h[a := a^0]$ to obtain a larger heap, which can be split into the disjoint heaps h and $[a := a^0]$, since $h[a := a^0] \quad h \;] \; [a := a^0]$. Similarly, if a heap h can be split so that $h \quad h_1 \;] \; [a := a^0]$ then we know $a \in \mathrm{dom}(h_1)$ and consequently that $h_1 = h[a := ?]$.

Definition 4.1.6 (Memory model). A *memory model* over A is a set H of heaps (partial functions from A to A) such that all the following conditions hold:

$$\quad 2 H; \tag{4.3}$$

$$h[a := a^0] \; 2 \; H; \tag{4.4}$$

$$h[a := ?] \; 2 \; H; \tag{4.5}$$

$$\mathrm{dom}(h_1) \setminus \mathrm{dom}(h_2) = \; ; =) \; h^0 \quad h_1 \;] \; h_2 \text{ for some heap } h^0 \; 2 \; H; \tag{4.6}$$

$$h_1 \quad h =) \; h \quad h_1 \;] \; h^0 \text{ for some heap } h^0 \; 2 \; H; \tag{4.7}$$

for every $h; h_1; h_2 \; 2 \; H$ and $a; a^0 \; 2 \; A$.

A memory model which satisfies all conditions except Equation (4.7) is called a *classical memory model*. In fact, in the remainder of this chapter, all results also hold for classical memory models: only in Chapter C, where we introduce intuitionistic separation logic, it is necessary to consider memory models with the additional condition of Equation (4.7). However, for uniformity in presentation, we shall include the condition, even when it is strictly speaking not necessary.

It is not a problem to confuse a memory model and the set H of heaps, as long as we have that H is a memory model: this convention is not different from taking H as the carrier set that is closed under the given operations and conditions. There are many memory models, and we have a look at various ways of constructing them later in this section.

The above conditions require compatibility conditions on the heap partitioning relation. In particular, Equation (4.6) imposes the condition on the set H of heaps that any two disjoint heaps can be merged, and Equation (4.7) imposes the condition on H that if we know that heap h is an extension of heap h_1 then there must exists a heap h_2 which consists of the remaining location-value mappings. In fact, the last condition imposes that the empty heap must be in H by taking $h_1 = h$, thus Equation (4.3) is redundant.

Note that, in the case we work with the set of finite heaps, Equations (4.6) and (4.7) do not add much. For any two disjoint finite heaps $h_1; h_2$, we can always construct the heap which is their union $h_1 \;] \; h_2$. This is easy to see, since any finite heap can simply be regarded as a finite construction from the empty heap and a finite association list of locations and values. The resulting finite heap simply zips the two association lists together. Similarly, for any finite heap h, and hence also a finite heap $h_1 \quad h$, we clearly can find a heap h^0 that is the remainder: we just look at the association list of locations and values that comprise h and filter out any of the locations that are in h_1. These constructions are rather obvious.

However, and this is crucial, in the case of infinite heaps these constructions are no longer straightforward. Equation (4.6) thus expresses that it must always be possible to merge heaps, even if one of the two heaps is infinite. And, furthermore, Equation (4.7) expresses that in case we have an infinite heap h and a (finite or infinite) heap h_1 we can always find the remainder of 'subtracting' h_1 from h, that is potentially infinite too.

Proposition 4.1.7. *If h $h_1 \rceil h_2$ and g $h_1 \rceil h_2$ then $h = g$.*

Proof. We have heap extensionality, $h = g$ if $h(a) = g(a)$ for all $a \in A$. But $h(a)$ is fixed by Equation (4.2), and so is $g(a)$. Our property follows from a case analysis of a: for either $a \in \mathrm{dom}(h_1)$, or $a \in \mathrm{dom}(h_2)$, or $a \notin \mathrm{dom}(h_1)$ and $a \notin \mathrm{dom}(h_2)$, $h(a) = g(a)$. The case $a \in \mathrm{dom}(h_1)$ and $a \in \mathrm{dom}(h_2)$ does not occur by Equation (4.1). □

Now we can use memory models to give semantics to the separating connectives, where we consider memory models instead of arbitrary non-empty sets of heaps.

Definition 4.1.8 (Memory structures). A *memory structure* $\mathsf{H} = (\mathsf{A}; H)$ is a general heap structure consisting of a structure $\mathsf{A} = (A; I)$ with domain A and interpretation I, and as set of heaps a memory model H over A.

Since MSL can be seen as GSL but restricted to a certain class of structures, we also have the coincidence condition, invariance under renaming, and the denotation of a formula $[\![\ ]\!]^{\mathbf{MSL}}$:

$$[\![\ ]\!]^{\mathbf{MSL}} = f(h; \ ) \mid \mathsf{H}; h; \quad \models^{\mathbf{MSL}} \quad g.$$

We shall use the superscript MSL to mean that the notions of satisfaction, validity, and denotation, previously defined for general heap structures, are restricted to memory structures.

Given a theory, i.e. a set of sentences , we write $\mathsf{H}; h \models^{\mathbf{MSL}}$ to mean that all sentences in are satisfied by all memory structures $\mathsf{H} = (\mathsf{A}; H)$ and heaps $h \in H$, that is, $\mathsf{H}; h \models^{\mathbf{MSL}}$ for all $\in$. We may also speak of ' is satisfied by H and h'. A theory is *satisfiable* if there exists a memory structure H and heap h such that $\mathsf{H}; h \models$. A theory is *finitely satisfiable* if every finite subset of is satisfiable.

Given a sentence , we write $\models^{\mathbf{MSL}}$ to mean $\mathsf{H}; h \models^{\mathbf{MSL}}$ for all memory structures $\mathsf{H} = (\mathsf{A}; H)$ and heaps $h \in H$ such that $\mathsf{H}; h \models^{\mathbf{MSL}}$, and say that is a *semantic consequence* of . In case is a context and a formula, then by $\models^{\mathbf{MSL}}$ we mean $\mathsf{H}; h; \quad \models^{\mathbf{MSL}}$ for all memory structures $\mathsf{H} = (\mathsf{A}; H)$, heaps $h \in H$, and valuations of A such that $\mathsf{H}; h; \quad \models^{\mathbf{MSL}}$ for all $\in$. Both readings coincide if and have no free variables. (The superscript MSL may be dropped if clear from context.)

To investigate the relation between the semantics defined above, and the standard and full semantics of separation logic defined earlier, we relate their notions of validity. Our main objective now is to show that $\models^{\mathbf{MSL}}$ implies $\models^{\mathbf{WSL}}$ and $\models^{\mathbf{FSL}}$. To do so, we construct a number of different memory

models given a fixed structure A with domain A. Note that there is a natural order between memory models by measuring their cardinality. We will show that the smallest memory model corresponds with the standard semantics, and the largest memory model corresponds with the full semantics.

To consider the smallest memory model H, suppose we start out with the empty set and add in only the necessary heaps to satisfy the requirements for H to be a memory model. The empty heap must be in H. For every heap and pair of elements of A, we can perform a heap update operation. Every finite sequence of heap updates results in a heap. Performing a heap clear operation only removes a location from the heap, which can be undone by performing a heap update again. The set of finitely-based partial functions over A is the smallest memory model.

The largest memory model is simply the set of all partial functions over A.

Definition 4.1.9. A *finite memory structure* is a memory structure $H = (A; H)$ where H is the smallest memory model over A.

Definition 4.1.10. A *full memory structure* is a memory structure $H = (A; H)$ where H is the largest memory model over A.

Every structure induces a full memory structure, since the largest memory model is unique. Similarly, every structure also induces a finite memory structure, since the smallest memory model is also unique. It is easy to verify that the general semantics restricted to finite memory structures coincides with the standard semantics, and the general semantics restricted to full memory structures coincides with the full semantics of separation logic. If the underlying structure has a finite domain, then the smallest and largest memory model coincide (and this confirms that the standard and full semantics coincide in this case too). If the structure has an infinite domain, the smallest and largest memory model are separated.

Proposition 4.1.11. $\models^{\mathbf{GSL}}$ *implies* $\models^{\mathbf{MSL}}$, *and* $\models^{\mathbf{MSL}}$ *implies* $\models^{\mathbf{WSL}}$ *and* $\models^{\mathbf{FSL}}$.

Proof. If $\models^{\mathbf{GSL}}$ then $H \models$ for all general heap structures H. That includes all memory structures H. Since each structure A induces a finite and full memory structure, we also have $\models^{\mathbf{WSL}}$ and $\models^{\mathbf{FSL}}$. □

From the above proposition, there is an easy heuristic for finding invalid sentences with respect to the general semantics: sentences that are invalid in either the standard semantics or the full semantics are also invalid in the general semantics. The counter-examples for either of them can be used as counter-example in the general semantics, by taking one of the induced memory structures corresponding to the specific counter-example.

Finally, before turning to the semantics of pointer programs, we consider the relation between MSL and the semantics underlying the proof system SL (or, RSL under the assumption that every heap is functional). In the completeness proof of Chapter 3, we have constructed a structure out of a maximally consistent set of separation logic formulas. It turns out that this constructed structure *also* is a memory structure. Consider that in the constructed structure we have as set H of

heaps the set of first-order definable heaps. Now it is easy to see that the empty heap is first-order definable, and given any heap h then the operations of heap update and heap clear are also first-order definable since every element of the domain is expressible, since we have explicitly taken as domain the equivalence classes of terms and hence every element of the underlying structure has a denotation as a term. Going further, for any two first-order definable heaps that are disjoint, we can also form the disjoint union by simply taking the disjunction of the corresponding formulas. And, finally, for any first-order definable heap and any first-order definable subheap, we can also express the 'subtraction' of one heap from the other as a first-order formula based on the given descriptions.

In the remainder we shall speak of *comprehensive memory structures* to mean structures which lie in the intersection of the two classes of general structures, namely those general structures that satisfies both the properties of a comprehensive structure (i.e. closed under semantic comprehension) and the properties of a memory structure (i.e. the set of heaps must be a memory model).

4.2 Semantics of pointer programs

In this section, we introduce the semantics of pointer programs following along the lines of Chapter B on Hoare's logic.

Given a first-order signature . A *pointer program signature* $PPS(\ )$ includes the following operations:

- the *assignment* operation $x := y$
  (where y is an accessible and x is a changed program variable),
- the *lookup* operation $x := [y]$
  (where y is an accessible and x is a changed program variable),
- the *mutation* operation $[x] := y$
  (where x and y are accessible program variables),
- the *allocation* operation $x :=$ new(y)
  (where y is an accessible and x is a changed program variable),
- the *deallocation* operation delete(x)
  (where x is an accessible program variable),

and every test is a quantifier-free pure formula $(x_1;::::;x_n)$, with as accessible program variables $x_1;::::;x_n$ corresponding to the free variables of . Every pointer program signature is also a first-order program signature (see Definition B.1.2). A *pointer program* is a statement formed from statements based on a pointer program signature (and, similarly, a *recursive pointer program* consists of a set of declarations and a main statement). Note that the tests of (statements of) pointer programs are pure quantifier-free formulas, as before. To give semantics to pointer programs, we introduce *spatial machine models*, analogous to *logical machine models* of Chapter B. A spatial machine model is based on a memory

structure H. The spatial machine model is a failure-sensitive machine model in the sense that its states are pairs of heaps (in the set of heaps of H) and valuations (of the underlying structure of H).

Before introducing the semantics, we introduce auxiliary concepts needed in the formulation of spatial machine models. Given two heaps $h; h^0$ (partial functions over some set), then we say that h^0 has a *finite distance* to h whenever it is the case that $h^0 = h[a_1 := a_1^0] ::: [a_n := a_n^0][a_{n+1} := ?] ::: [a_{n+k} := ?]$. Informally, one obtains h^0 by finitely many applications of heap update and heap clear to h. This notion is symmetric: h^0 has a finite distance to h if and only if h has a finite distance to h^0. Note that it is also possible that h and h^0 have a finite distance even in the case that both h and h^0 are partial functions without a finite basis (i.e. both have an infinite domain). Further, when we compare sets of states in the notations $X \quad Y \bmod Z$ and $X \quad f(X) \bmod Z$, we speak only about the valuations and leave the heaps untouched.

Definition 4.2.1. A *spatial machine model* M is a pair of a memory structure H and an operationalization consisting of:

- for each operation O, a transition function which is a partial function O^M of valuations to a set of valuations of H,
- for every operation $x := y$, the transition function $(x := y)^M$ is defined by mapping $(h; \ )$ to $(h; \ [x := \ (y)])$,
- for every operation $x := [y]$, the transition function $(x := [y])^M$ is defined by mapping $(h; \ )$ to $(h; s[x := h(s(y))])$ if $s(y)$ *2* $dom(h)$ and to **fail** otherwise,
- for every operation $[x] := y$, the transition function $([x] := y)^M$ is defined by mapping $(h; \ )$ to $(h[s(x) := s(y)]; s)$ if $s(x)$ *2* $dom(h)$ and to **fail** otherwise,
- for every operation $x := \mathbf{new}(y)$, the transition function $(x := \mathbf{new}(y))^M$ is defined by mapping $(h; \ )$ to the set $f(h[n := s(y)]; s[x := n]) \ j \ n \ 2 \ dom(h)g$,
- for every operation $\mathbf{delete}(x)$, the transition function $(\mathbf{delete}(x))^M$ is defined by mapping $(h; \ )$ to $(h[s(x) := ?]; s)$ if $s(x)$ *2* $dom(h)$ and to **fail** otherwise,
- for the transition function O^M we have the *change condition* that either $O^M(h; \ ) = \mathbf{fail}$ or $^0[V_1 \, n \, change(O)] = [V_1 \, n \, change(O)]$ for every $(h^0; \ ^0)$ *2* $O^M(h; \ )$,
- for the transition function O^M we have the *access condition* that states that $O^M(h; \ ) \quad O^M(h^0; \ ^0) \bmod var(O)$ for every $; \ ^0$ for which $[access(O)] = \ ^0[access(O)]$ holds,
- for the transition function O^M we have the *e ect condition* that for every $(h^0; \ ^0)$ *2* $O^M(h; \ )$ we have that h^0 has a finite distance to h,
- for the transition function O^M we have the *frame condition* that for every $O^M(h_0; \ ) \ 6= \mathbf{fail}$ and $(h^0; \ ^0)$ *2* $O^M(h_0 \ ] \ h_1; \ )$, there is a h_0^0 such that $(h_0^0; \ _0^0)$ *2* $O^M(h_0; \ )$ and $h^0 = h_0^0 \ ] \ h_1$.

Since spatial machine models are failure-sensitive machine models, we get the operational and denotational semantics 'for free'. We also write $\langle M; \mathsf{H}\rangle$ for a spatial machine model to indicate its underlying memory structure H. A spatial machine model is a failure-sensitive machine model in the following sense: the state space of a spatial machine model is the set of pairs of heaps and valuations of H, and the given operationalization induces an operationalization for tests by associating every quantifier-free pure formula φ to the set $[\![\varphi]\!]^{\mathbf{MSL}}_{\mathsf{H}}$ that denotes the pairs of heaps and valuations that satisfy φ in structure H. Note that for these formulas φ we have $(h; \zeta) \in [\![\varphi]\!]^{\mathbf{MSL}}_{\mathsf{H}}$ if and only if $\zeta \in [\![\varphi]\!]^{\mathbf{CL}}_{\mathsf{A}}$ where A is the underlying structure of H, that is, tests do not depend on the heap.

In fact, in the operational semantics of pointer programs, we have as configurations triples consisting of a program S, a heap h, and a valuation ζ. The successful execution of any basic instruction S is denoted by $(S; h; \zeta) \to (\downarrow; h'; \zeta')$, whereas $(S; h; \zeta) \to \mathsf{fail}$ denotes a failing execution (e.g. due to access of a 'dangling pointer'). The small-step semantics (and similar for the big-step semantics) of the primitive operations are as follows, as follows from the definition of the operationalization of spatial machine models above:

- $(x := y; h; \zeta) \to (\downarrow; h; [x := \zeta(y)])$,
- $(x := [y]; h; \zeta) \to (\downarrow; h; [x := h(\zeta(y))])$ if $\zeta(y) \in \mathit{dom}(h)$,
- $(x := [y]; h; \zeta) \to \mathsf{fail}$ if $\zeta(y) \notin \mathit{dom}(h)$,
- $([x] := y; h; \zeta) \to (\downarrow; h[\zeta(x) := \zeta(y)]; \zeta)$ if $\zeta(x) \in \mathit{dom}(h)$,
- $([x] := y; h; \zeta) \to \mathsf{fail}$ if $\zeta(x) \notin \mathit{dom}(h)$,
- $(x := \mathsf{new}(y); h; \zeta) \to (\downarrow; h[n := \zeta(y)]; [x := n])$ where $n \notin \mathit{dom}(h)$,
- $(\mathsf{delete}(x); h; \zeta) \to (\downarrow; h[\zeta(x) := ?]; \zeta)$ if $\zeta(x) \in \mathit{dom}(h)$,
- $(\mathsf{delete}(x); h; \zeta) \to \mathsf{fail}$ if $\zeta(x) \notin \mathit{dom}(h)$.

Crucially, with this definition we abstract from out of memory errors, e.g. from the heap which has no free location it is not possible to take a step with the program $x := \mathsf{new}(y)$. Intuitively, this makes 'out of memory' behave similarly as a divergence: it is as if the underlying implementation would keep searching for a free spot on the heap, but diverge since it will never find one.

The other operations do have explicit failures: looking up the value of an unallocated location results in fail, so do mutation of an unallocated location, or the deallocation thereof.

Note that it is possible to instantiate this program semantics with different structures, and thus different conceptions of the heap. In the standard semantics one would use finite heaps, but it is also possible to base the operational semantics on top of the full set of heaps (finite or infinite heaps) of any underlying structure, or to base it on top of any other memory structure introduced in the previous section. However, it is now obvious from the definition above that general structures

are insufficient, because we want to define transitions in terms of heap updates and heap clear operations.

Since we also have an interpretation where there are potentially infinite heaps, the design choice to let allocation behave as a divergence in case of no free location becomes more clear. In the case of finite heaps (but an infinite domain of the underlying structure), allocations do not diverge, as is the case in the standard semantics.

Just like in Hoare's logic, we have that the access and change conditions can be lifted to statements S.

Lemma 4.2.2 (Change Lemma). *Given a set of proper states X,*

$$X \quad \langle M; \mathrm{H}\llbracket S \rrbracket(X) \bmod \mathit{change}(S).$$

Lemma 4.2.3 (Access Lemma). *Given two sets of proper states X, Y such that $X \quad Y \bmod (V \setminus \mathit{access}(S))$, then*

$$\langle M; \mathrm{H}\llbracket S \rrbracket(X) \quad \langle M; \mathrm{H}\llbracket S \rrbracket(Y) \bmod (V \setminus \mathit{var}(S)).$$

Similar to Hoare's logic, these express that a statement only modifies the variables $\mathit{change}(S)$, and that the outcome of a statement is only dependent on the variables $\mathit{access}(S)$. However, note that these notions require the same initial heap. These properties do not capture the so-called 'heap footprint' of a statement. For dealing with the heap, we have the other two conditions: the effect and frame conditions can also be lifted to statements.

Lemma 4.2.4 (Effect Lemma). *For any $(h', \_') \in \langle M; \mathrm{H}\llbracket S \rrbracket(h, \_)$ we have that h' has a finite distance to h.*

Proof. Intuitively we only have to check the small-steps, and check that the property is preserved compositionally. Since we already have that for every primitive operation this property holds due to the definition of spatial machine models, it suffices to observe that in a terminating execution we have only finitely many steps. □

A statement S which has no effect on the heap is called an *effect-free statement*, whereas a statement that does have an effect on the heap is called *effectful*.

Lemma 4.2.5 (Frame Lemma I). *If $\mathsf{fail} \in \langle M; \mathrm{H}\llbracket S \rrbracket(h_0, \_)$ then it is also the case that $\mathsf{fail} \in \langle M; \mathrm{H}\llbracket S \rrbracket(h_0 \rightharpoonup h_1, \_)$.*

Proof. We know an execution from the smaller heap h_0 does not lead to failure. Suppose we now add additional locations and this causes a failure to appear in the computation. This failure must happen in some execution at a primitive operation (the other statements do not cause failures). Due to the frame condition, and the fact that the operationalization maps a state to either fail or a set of states, we know that if a smaller heap does not lead to failure, a larger heap must also not. But that contradicts the assumption that a failure appears in the computation from a larger heap. □

Lemma 4.2.6 (Frame Lemma II). *Given* fail $\notin \langle M, \mathrm{H}i\mathrm{JS}\mathrm{K}(h_0, \_) \rangle$ *and* $(h', \_) \in \langle M, \mathrm{H}i\mathrm{JS}\mathrm{K}(h_0 \bullet h_1, \_) \rangle$, *there is a* h'_0 *such that* $(h'_0, \_) \in \langle M, \mathrm{H}i\mathrm{JS}\mathrm{K}(h_0, \_) \rangle$ *and* $h' = h'_0 \bullet h_1$.

Proof. The intuition is that from the first premise, we know that statement S accesses or changes the locations on heap h_0. Hence, adding additional locations to the heap does not affect the execution of the statement, and these additional locations remain unmodified during the execution (otherwise, if it would change these locations, then the statement S would lead to a failure when executing it on the smaller heap h_0 that lacks these additional locations). □

In fact, the first premise suggests an important concept of pointer programs: the 'footprint' of a program. Let the footprint of a statement S be the set of states $(h, \_)$ such that fail $\notin \langle M, \mathrm{H}i\mathrm{JS}\mathrm{K}(h, \_) \rangle$.

We again have program specifications $\{ \_ \}\ S\ \{ \_ \}$, being a triple that consists of a precondition $\_$, a program S, and a postcondition $\_$. Note that, in Reynolds' logic, the precondition and postcondition are formulas of separation logic, and statements are formed according to a pointer program signature. Every program specification of Hoare's logic is *also* a program specification in Reynolds' logic, since every statement of a pointer program signature is also a statement of a first-order program signature and every formula of classical logic is also a formula of separation logic.

Again we formally define whether a program specification is satisfied, but now in a spatial machine model. Recall that formulas never denote the improper state fail. Thus we have the interpretation of program specifications called *strong partial correctness*, defined as such:

$$\langle M, \mathrm{H}i \rangle \models^{\mathbf{RL}} \{ \_ \}\ S\ \{ \_ \} \text{ if and only if } \langle M, \mathrm{H}i\mathrm{JS}\mathrm{K}(\mathrm{HJ}\ \_\mathrm{K}^{\mathbf{MSL}}) \subseteq \mathrm{HJ}\ \_\mathrm{K}^{\mathbf{MSL}}.$$

Since fail is never in $\mathrm{HJ}\ \_\mathrm{K}^{\mathbf{MSL}}$, this interpretation explicitly states that the machine never fails when executing program S starting from any state in $\mathrm{HJ}\ \_\mathrm{K}^{\mathbf{MSL}}$. The superscript RL stands for Reynolds' Logic.

Before introducing the proof system for Reynolds' logic, note that in Hoare's logic we distinguish the background theory and program theory. Recall that a program theory is a set of program specifications (see Section B.4): it is possible to consider theories to consist of program specifications *or* formulas, since every formula in the background theory can be encoded as a program specification over the skip statement. As such, we directly introduce the following notion of semantic consequence in Reynolds' logic, without distinguishing the background theory from the program theory.

Let $\_$ be a theory (a set of program specifications or formulas). We write $\_ \models^{\mathbf{RL}} \{ \_ \}\ S\ \{ \_ \}$ to mean $\langle M, \mathrm{H}i \rangle \models^{\mathbf{RL}} \{ \_ \}\ S\ \{ \_ \}$ for every spatial machine model $\langle M, \mathrm{H}i \rangle$ such that $\langle M, \mathrm{H}i \rangle \models^{\mathbf{RL}} \{ \_' \}\ S'\ \{ \_' \}$ for each $\{ \_' \}\ S'\ \{ \_' \} \in \_$. We then say that the program specification $\{ \_ \}\ S\ \{ \_ \}$ is a *semantic consequence of* $\_$. For an empty theory, we simply write $\models^{\mathbf{RL}} \{ \_ \}\ S\ \{ \_ \}$ to mean that the program specification is *universally valid*.

4.3 Standard proof system

The standard proof system of Reynolds' logic **RL** is an extension of Hoare's logic **HL** (see Section B.4). Reynolds' logic is an extension of Hoare's logic, so we first revisit the proof rules of Hoare's logic.

If we would interpret the proof rules (axioms are proof rules without premises) of **HL** under the intended interpretation of **RL**, being spatial machine models, do they remain sound? Clearly, if we only consider the instances of the proof rules where the formulas are restricted to pure formulas in separation logic, i.e. those that are heap-independent, these proof rules remain sound also under our new interpretation with respect to spatial machine models. This follows from the fact that spatial machine models are extensions of logical machine models and as such have the same conditions as logical machine models: the interpretation of the basic assignment $x := y$ is the same, and the access and change conditions are also present.

But in Reynolds' logic, we want to extend these rules to all formulas of separation logic. The **skip** and **halt** axioms are easily shown sound, and so is the assignment axiom. The rules concerning complex statements remain sound, since these do not depend on the interpretation of formulas: their semantics is defined at the level of states, which abstracts away from the particular logical structure (being either valuations as in Hoare's logic, or heaps and valuations in Reynolds' logic).

What remains to be considered are the so-called adaptation rules.

- The consequence rule (conseq) is sound because

  hM; AiJSK(AJ 0K**MSL**) AJ 0K**MSL**

  follows from monotonicity of the semantics, and the fact that we have both AJ 0K**MSL** AJ K**MSL** and AJ K**MSL** AJ 0K**MSL**.

- In the substitution rule (subst) we make use of the access lemma to take any computation from *f g S f g* and change the initial state with respect to variable *x* that is not occurring in *S* to obtain another computation (the variable *x* can then not be overwritten by *S*). This still works as before. Further, the value to assign to *x* is the value of *y*, which must have the same value in the initial and final state again due to the change lemma. The specification then is satisfied by applying the substitution lemma on the initial and final state. Since the substitution lemma also holds for separation logic, this works out.

- The *9*-introduction rule (*9*-intro) still follows from the access lemma, since the value of *x* cannot have any eect on the computation of *S* nor influence the denotation of .

- However, the invariance rule (invar) no longer follows from the change lemma. The problem is that the formula can be heap-dependent, whereas *change*(*S*) only tracks variables and not locations on the heap. If *S* is eect-free, then it does not change the heap and thus the formula remains invariant. Otherwise,

S is effectful and could thus affect (dynamic) parts of the heap on which the formula depends. Concretely, we could take a program $[x] := y$ that modifies the location in x to become the value of y. However $change([x] := y)$ is empty, because none of the program variables change value after its execution. If we take the (valid) program specification $\{(x \mapsto \_) * (y \mapsto z)\}\ [x] := y\ \{\mathsf{true}\}$ as premise, and we would take as invariant formula $(x \mapsto z)$, then the resulting program specification $\{(x \mapsto \_) * (x \mapsto z)\}\ [x] := y\ \{\mathsf{true} * (x \mapsto z)\}$ no longer is valid! Namely, take y and z to be different values in the initial state. No variable is changed in the final state when compared to the initial state. However, it no longer is the case that $(x \mapsto z)$ holds in the final state, since the location was modified.

Summarizing, in Reynolds' logic we can have all proof rules of Hoare's logic, also extended to instances which have all formulas of separation logic, *except* for the invariance rule: the invariance rule only remains sound for pure formulas.

Furthermore, in the standard proof system for Reynolds' we have the following proof rules [188]. The *frame rule* is introduced to fill up the gap left by the invariance rule, allowing one to adapt local specifications to global specifications. We first introduce RL, and later give different sets of axioms to describe the primitive operations of every pointer program.

Definition 4.3.1. The proof system RL consists of:

- program specifications or formulas of separation logic as objects,
- the smallest deduction relation $\vdash_{\mathsf{RL}}$ satisfying the conditions:

  (skip) $\vdash_{\mathsf{RL}} \{p\}\ \mathsf{skip}\ \{p\}$,

  (halt) $\vdash_{\mathsf{RL}} \{p\}\ \mathsf{halt}\ \{\mathsf{false}\}$,

  (assign) $\vdash_{\mathsf{RL}} \{p[x := y]\}\ x := y\ \{p\}$,

  (block) $\{p[x := y]\}\ S\ \{q\} \vdash_{\mathsf{RL}} \{p\}\ \mathsf{begin\ local}\ x := y; S\ \mathsf{end}\ \{q\}$ if $FV(q) \setminus x = \emptyset$,

  (comp) $\{p\}\ S_1\ \{r\}; \{r\}\ S_2\ \{q\} \vdash_{\mathsf{RL}} \{p\}\ S_1; S_2\ \{q\}$,

  (if) $\{p \wedge g\}\ S_1\ \{q\}; \{p \wedge \neg g\}\ S_2\ \{q\} \vdash_{\mathsf{RL}} \{p\}\ \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \{q\}$,

  (while) $\{p \wedge g\}\ S\ \{p\} \vdash_{\mathsf{RL}} \{p\}\ \mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}\ \{p \wedge \neg g\}$,

  (conseq) $(p' \Rightarrow p); \{p\}\ S\ \{q\}; (q \Rightarrow q') \vdash_{\mathsf{RL}} \{p'\}\ S\ \{q'\}$,

  (subst) $\{p\}\ S\ \{q\} \vdash_{\mathsf{RL}} \{p[x := y]\}\ S\ \{q[x := y]\}$ for $x \notin var(S); y \notin change(S)$,

  (invar) $\{p\}\ S\ \{q\} \vdash_{\mathsf{RL}} \{p \wedge r\}\ S\ \{q \wedge r\}$ for either pure r or effect-free S, $FV(r) \setminus change(S) = \emptyset$,

  ($\exists$-intro) $\{p\}\ S\ \{q\} \vdash_{\mathsf{RL}} \{\exists x.\ p\}\ S\ \{q\}$ for $x \notin var(S) \cup FV(q)$,

  (frame) $\{p\}\ S\ \{q\} \vdash_{\mathsf{RL}} \{p * r\}\ S\ \{q * r\}$ if $FV(r) \setminus change(S) = \emptyset$.

Lemma 4.3.2 (Soundness).

$$\vdash^{\mathbf{RL}} \{P\}\, S\, \{Q\} \text{ implies } \models^{\mathbf{RL}} \{P\}\, S\, \{Q\}.$$

Proof. By induction on the structure of the deduction. Most cases are already discussed above, except the frame rule. The soundness of the frame rule goes along the following lines, see also [230]. From the premise we know that $\models^{\mathbf{RL}} \{P\}\, S\, \{Q\}$. We thus know that the execution of S does not fail in a state that satisfies P. Due to the first frame lemma we know that, if we extend the initial heap with an additional part, it does not lead to failure either. From the second frame lemma we also know that the final state can be split again in the unaffected part of the heap, in which the additional formula R still holds as was assumed as part of the semantics of the separating conjunction in the initial state. □

Now consider the set of local axioms [188].

Definition 4.3.3. The set of *local* axioms is:

(lookup) $\vdash \{(y \mapsto z)\}\, x := [y]\, \{(x \doteq z) \wedge (y \mapsto z)\}$ where $x \neq y$ and z is fresh,

(lookup') $\vdash \{(x \doteq w) \wedge (x \mapsto z)\}\, x := [x]\, \{(x \doteq z) \wedge (w \mapsto z)\}$ where z, w are fresh,

(mutation) $\vdash \{(x \mapsto \_)\}\, [x] := y\, \{(x \mapsto y)\}$,

(allocation) $\vdash \{\mathsf{emp}\}\, x := \mathsf{new}(y)\, \{(x \mapsto y)\}$ where $x \neq y$,

(allocation') $\vdash \{(x \doteq z) \wedge \mathsf{emp}\}\, x := \mathsf{new}(x)\, \{(x \mapsto z)\}$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto \_)\}\, \mathsf{delete}(x)\, \{\mathsf{emp}\}$.

Lemma 4.3.4. *The* local *axioms are sound with respect to* MSL.

By applying the frame rule it becomes possible to extend some of these program specifications to a global description of the heap.

Next, consider the set of *global* axioms [188].

Definition 4.3.5. The set of *global* axioms is:

(lookup) $\vdash \{\exists z.\, (y \mapsto z)\}\, [w := x]\, x := [y]\, \{\exists w.\, (y[x := w] \mapsto x)\}\, [z := x]$ where z, w, x are distinct, z, w, y are distinct and $x \notin FV(\_)$,

(mutation) $\vdash \{(x \mapsto \_)\}\, [x] := y\, \{(x \mapsto y)\}$,

(allocation) $\vdash \{\}\, x := \mathsf{new}(y)\, \{\exists z.\, (x \mapsto y[x := z])\}\, [x := z]$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto \_)\}\, \mathsf{delete}(x)\, \{\}$.

Lemma 4.3.6. *The* global *axioms are sound with respect to* MSL.

There is also the set of *backwards* axioms [188], in the sense that these axioms allow reasoning backwards from a given postcondition.

Definition 4.3.7. The set of *backwards* axioms is:

(assign) $\vdash \{P\}\ [x := y]\{x := y\}\ P\ \{Q\}$,

(lookup) $\vdash \{\forall z.(y \mapsto z) \wedge [x := z]\{x := [y]\}\ P\ \{Q\}$ where z is fresh,

(mutation) $\vdash \{(x \mapsto \_) \wedge ((x \mapsto y) \rightarrow P)\}\ [x] := y\ \{Q\}$,

(allocation) $\vdash \{\forall z.(z \mapsto y) \rightarrow [x := z]\{x := \mathsf{new}(y)\}\ P\ \{Q\}$ where z is fresh,

(deallocation) $\vdash \{(x \mapsto \_) \wedge \{\mathsf{delete}(x)\}\ P\ \{Q\}$.

These backwards axioms also express the weakest precondition:

Lemma 4.3.8. *The* backwards *axioms are sound with respect to* MSL*, and describe the weakest precondition with respect to the given primitive operations and postcondition.*

And finally the set of *forwards* axioms [15], in the sense that these axioms allow reasoning forwards from a given precondition.

Definition 4.3.9. The set of *forwards* axioms is:

(assign) $\vdash \{P\}\ x := y\ \{\exists z.\ [x := z] \wedge (y[x := z] \doteq x) \wedge Q\}$ where z is fresh,

(lookup) $\vdash \{P \wedge (y \mapsto \_)\}\ x := [y]\ \{\exists z.\ \ldots \wedge [x := z]Q\}$
where $\ldots = (y[x := z] \mapsto x)$ and z is fresh,

(mutation) $\vdash \{P \wedge (x \mapsto \_)\}\ [x] := y\ \{(x \mapsto y) \wedge ((x \mapsto \_) \rightarrow Q)\}$,

(allocation) $\vdash \{P\}\ x := \mathsf{new}(y)\ \{\exists z.\ (x \mapsto y[x := z]) \wedge [x := z]Q\}$,

(deallocation) $\vdash \{P \wedge (x \mapsto \_)\}\ \{\mathsf{delete}(x)\}\ \{(x \mapsto \_) \rightarrow Q\}$.

Lemma 4.3.10. *The* forwards *axioms are sound with respect to* MSL*, and describe the strongest postcondition with respect to the given primitive operations and precondition.*

These forwards axioms also express the strongest postcondition with respect to the given primitive operations and precondition, but note that we have additional assumptions in the precondition that ensures absence of failure.

Finally, we observe that we have admissibility of the frame rule for pure pointer programs in case we take the *backwards* axioms for the primitive operations. In pure pointer programs, there are no other operations than the operations of assignment, lookup, mutation, allocation and deallocation.

Lemma 4.3.11. *The frame rule is admissible in the proof system* RL *with the* backwards *axioms, given that the background theory is maximally consistent.*

Proof. Consider a deduction that makes use of the frame rule. The strategy is to 'push upward' the instance of the frame rule to the top of the deduction, i.e. where there is an axiom applied that is either (skip), (halt), (assign), or one of the pointer program operations (lookup), (mutation), (allocation), (deallocation). For these axioms, it can be verified that the conclusion of the frame rule is deducible (from the empty context). We then analyze the adaptation and structural rules.

For the adaptation rule, we illustrate the proof by showing how to push the frame rule up the consequence rule. Consider the following deduction in which the frame rule is applied directly after the consequence rule:

$$\dfrac{\dfrac{\overset{D}{\vdots}}{\Theta \Rightarrow \Theta' \quad \{P\}\, S\, \{Q\} \quad \Theta' \Rightarrow \Theta}}{\{P'\}\, S\, \{Q'\}}$$

It is our induction hypothesis that the frame rule is admissible for shorter deductions, so from deduction D with conclusion $\{P\}\, S\, \{Q\}$ we obtain deduction D' with conclusion $\{P'\}\, S\, \{Q'\}$. By then applying the consequence rule we obtain the following deduction:

$$\dfrac{\Theta \Rightarrow \Theta' \quad \dfrac{\overset{D'}{\vdots}}{\{P'\}\, S\, \{Q'\}} \quad \Theta' \Rightarrow \Theta}{\{P\}\, S\, \{Q\}}$$

since we know $\Theta \Rightarrow \Theta'$ and $\Theta' \Rightarrow \Theta$ are in the background theory and the background theory is maximally consistent, the two remaining premises must be in the background theory too. The remaining adaptation rules are similar.

For the structural rules, we illustrate how the proof goes by looking at sequential composition. Consider that we have a deduction in which the frame rule is applied directly following the sequential composition:

$$\dfrac{\dfrac{\overset{D_1}{\vdots}}{\{P\}\, S_1\, \{R\}} \quad \dfrac{\overset{D_2}{\vdots}}{\{R\}\, S_2\, \{Q\}}}{\{P\}\, S_1; S_2\, \{Q\}}$$

By induction hypothesis, we obtain from D_1 and D_2 two deductions D_1' and D_2' with respectively conclusions $\{P'\}\, S_1\, \{R'\}$ and $\{R'\}\, S_2\, \{Q'\}$. Note that the changed variables of the smaller programs are contained in the changed variables of the sequential composition, so the frame rule would be applicable. We then obtain the following deduction:

$$\dfrac{\dfrac{\overset{D_1'}{\vdots}}{\{P'\}\, S_1\, \{R'\}} \quad \dfrac{\overset{D_2'}{\vdots}}{\{R'\}\, S_2\, \{Q'\}}}{\{P'\}\, S_1; S_2\, \{Q'\}}$$

which finishes this case. The remaining structural cases are similar. For the **while** and **if** rules, one also needs the equivalence $(\Gamma \wedge \Pi) \Rightarrow (\Delta \wedge \Pi)$, which holds since Π is a pure quantifier-free formula. □

Note that it is an open problem whether the proof above may be adapted to the setting of recursive procedures, with or without parameters (see also the Ph.D. thesis of Al Ameen [6]). Although intuitively we can 'push' the frame rule through the assumptions about each procedure call, this may result in an infinite amount of assumptions obtained that way and it is not obvious whether this infinite set of assumptions is compact, in the sense that all consequences can also be derived from a finite subset of these infinite assumptions without using the frame rule.

4.4 Dynamic separation logic

We have now seen different axiomatizations of the primitive operations of all pointer programs. In particular, it can be observed that in the *backwards* and *forwards* sets of axioms, we do not systematically analyze the structure of the given postcondition or precondition. For example, in the mutation axiom of the *backwards* set, the given postcondition is simply *verbatim* part of the weakest precondition. Furthermore, the axioms for (mutation), (allocation) and (deallocation) all increase the complexity of the generated formula as measured by their number of nested separating connectives. Thus, even starting with a first-order formula, the resulting generated formula necessarily is a formula of separation logic. This is contrary to how, e.g., the (assign) axioms work in *backwards* and *forwards*, and the (lookup) axiom works in *backwards*, which perform a substitution to perform a structural analysis of the given postcondition, and do not introduce additional separating connectives.

This raises the questions: are there alternative ways to axiomatize these operations? In particular, is there a way to axiomatize (mutation), (allocation) and (deallocation) so that the structure of the given postcondition is analyzed, akin to a substitution operator? Can we give weakest preconditions and strongest postconditions without increasing the nesting depth of separating connectives?

To answer these questions, we introduce in our assertion language an additional *program modality* for each statement S, which has highest binding priority, denoted as $[S]p$. This extended language is called *dynamic separation logic* (DSL), and as such the syntax of dynamic separation logic becomes:

$$p; q ::= \ldots \mid [S]p$$

We shall use the Roman letters $p; q$ to stand for formulas of DSL, whereas we use the Greek letters $;$ to stand for formulas of separation logic. Note that in case of the assignment $x := y$ the program modality $[x := y]$ is different from the (capture-avoiding) substitution operator $[x := y]$, since the former is a formula of our extended language, whereas the latter is a meta-operation defined on the separation logic formula .

We also extend the semantics of separation logic to interpret the additional program modalities. The intended semantics of dynamic separation logic extends the semantics of separation logic by interpreting the modality $[S]p$ as expressing the weakest precondition of statement S with postcondition p:

- ...
- $\langle M, \mathrm{H}i, h, \ldots \rangle \models^{\mathbf{DSL}} [S]p$ iff $(S, h, \ldots) \not\to \mathsf{fail}$ and $\langle M, \mathrm{H}i, h', \ldots' \rangle \models p$ for all h', s' such that $(S, h, \ldots) \to (X, h', \ldots')$.

Note that to give semantics to formulas of dynamic separation logic, we need to interpret the formulas with respect to a fixed spatial machine model $\langle M, \mathrm{H}i$, which itself depends on a memory structure $\mathrm{H} = (\mathrm{A}, H)$, with an underlying structure A and memory model H, which is the same memory structure with which we evaluate the formula of dynamic separation logic. This also explains the need for the effect condition on spatial machine models, since we need the resulting heap h' in every final configuration to be in the set of heaps H. Since h' is a finite distance from h, and we know that memory models are closed under the operations of heap update and heap clear, we know that h' must be in the memory model too.

Extending Reynolds' logic to also include formulas of dynamic separation logic in its program specifications, it is not difficult to see that we have that

$$\models \{p\}S\{q\} \quad \text{iff} \quad \models p \Rightarrow [S]q$$

holds, and $\models \{p\}S\{q\}$ implies $\models^{\mathbf{DSL}} p \Rightarrow [S]q$, that is, $[S]q$ indeed expresses the weakest precondition of statement S and postcondition q.

In dynamic logic axioms are introduced to simplify formulas in which modalities occur. We have the following basic equivalences E1-3 for assignments.

Lemma 4.4.1 (Assignment). *Let the statement S be the assignment $x := y$.*

$$[S]\mathsf{false} \quad \mathsf{false} \tag{E1}$$
$$[S](p \circ q) \quad [S]p \circ [S]q \tag{E2}$$
$$[S](\forall z\, p) \quad \forall z\, ([S]p) \tag{E3}$$
$$[S]b \quad b[S] \tag{E4}$$

In E2 *we have that $\circ$ stands for the binary connectives $\Rightarrow$, $\wedge$, $\vee$.*
In E3 *we assume that z is not equal to x or y.*
In E4 *we have that b is either $(z_1 \doteq z_2)$, $C(z_1, \ldots, z_n)$, or $(z_1 \hookrightarrow z_2)$.*

The proofs of these equivalences for $[x := y]p$ proceed by a straightforward induction on the structure of p. The base cases of logical equality, predicates, and the weak 'points to' construct are handled by a straightforward extension of the *substitution lemma* of separation logic. In fact, for assignments, by E4 as base case and E1-3 for the inductive cases, we have for any formula of separation logic φ,

$$[x := y]\varphi \quad \text{iff} \quad [x := y]\varphi;$$

where the latter is the (capture-avoiding) substitution operator. The reason we present the axioms E1-4 in the way it is done above, is because we want to analyze what happens when we take different statements in the place of S: do these axioms then still hold?

The above equivalences E1-3 do not hold in general for the other primitive operations of pointer programs. Let $x; y$ be distinct variables. For example,

$$[x := [y]]\mathsf{false} \quad :(y \mapsto \_);$$

showing that lookup fails E1. For allocation, we do have

$$[x := \mathsf{new}(y)]\mathsf{false} \quad \mathsf{false};$$

but $[x := \mathsf{new}(y)](x \dot{\mapsto} y)$ is not equivalent to $:([x := \mathsf{new}(y)](x \doteq y))$, since

$$[x := \mathsf{new}(y)](x \dot{\mapsto} y) \quad (y \mapsto \_);$$

because to end up in a final state where $x \dot{\mapsto} y$ we need to have that y is already allocated in the initial state, whereas

$$[x := \mathsf{new}(y)](x \doteq y) \quad \forall z.((z \mapsto \_) \Rightarrow z \doteq y);$$

which forces the only free location to be y. But then it is also the case that $:([x := \mathsf{new}(y)](x \doteq y))$ expresses that $(z \mapsto \_)$ for some $z \dot{\mapsto} y$. So this shows allocation fails E2. Similar examples exists for the other primitive operations.

We now introduce new primitive operations, separate from pointer programs, called *pseudo-operations*. These pseudo-operations are not part of pointer programs, but we can give them semantics in the usual way through an operationalization. We have the pseudo-operation

$$\langle x \rangle := e$$

called *heap update*, and

$$\langle x \rangle := ?$$

called *heap clear*. These pseudo-operations could be described by the following small-step transitions:

$$(\langle x \rangle := y; h; \_) \Rightarrow (\mathsf{X}; h[\mathcal{E}(x) := \mathcal{E}(y)]; \_)$$
$$(\langle x \rangle := ?; h; \_) \Rightarrow (\mathsf{X}; h[\mathcal{E}(x) := ?]; \_)$$

In contrast to the mutation and deallocation operations, these pseudo-operations do not require that $\mathcal{E}(x) \in \mathit{dom}(h)$, e.g., if $\mathcal{E}(x) \notin \mathit{dom}(h)$ then the heap update $\langle x \rangle := y$ extends the domain of the heap, whereas mutation $[x] := y$ leads to failure in that case.

It is now crucial to observe that these are *pseudo*-operations, precisely because they fail the frame condition. As such, these operations can never occur in a spatial machine model, which requires the frame condition to hold for all operations. Before, we could establish the footprint of a pointer program simply by running a program on a small heap: if the program would lead to failure, then the location that was missing from the initial heap necessarily is in the footprint. However, these pseudo-operations never fail. Hence these cannot be used to determine the footprint of a program.

That the pseudo-operations do not satisfy the frame condition can best be seen by considering why the frame rule (in a hypothetical situation where we let the pseudo-operations in the place of a statement) would become unsound. Clearly, we have that

$$\models \{\mathsf{emp}\}\ \langle x\rangle := y\ \{(x \mapsto y)\}$$

holds. However, the conclusion of the frame rule fails:

$$\not\models \{\mathsf{emp} * (x \mapsto y)\}\ \langle x\rangle := y\ \{(x \mapsto y) * (x \mapsto y)\}$$

because the initial state is satisfiable (there surely is a heap in which only the location x is allocated and has value y), and the execution successfully terminates (the pseudo-operations never lead to failure). But, the final state does not satisfy $(x \mapsto y) * (x \mapsto y)$ because that formula is equivalent to false.

Strictly speaking, we thus consider not only the modality $[S]$ where S is given semantics by a spatial machine model, but also consider modalities over these two pseudo-operations: $[\langle x\rangle := y]$ and $[\langle x\rangle := ?]$.

The above equivalences E1-3, with E2 restricted to the (standard) logical connectives, *do* hold for the *pseudo*-operations.

In the following lemma we give an axiomatization in dynamic separation logic of the primitive operations in terms of simple assignments and these two pseudo-operations. For comparison we also give the standard *backwards* axiomatization [188, 81, 15].

Lemma 4.4.2 (Axioms basic instructions).

$$[x := [e]]p \quad \exists y((e \hookrightarrow y) \wedge [x := y]p); \tag{E5}$$

$$[[x] := e]p \quad \begin{array}{l}(x \hookrightarrow -) \wedge [\langle x\rangle := e]p \\ (x \mapsto -) \quad ((x \mapsto e) \quad p)\end{array} \tag{E6}$$

$$[x := \mathsf{new}(e)]p \quad \begin{array}{l}\forall x((x \not\mapsto -) \quad [\langle x\rangle := e]p) \\ \forall x((x \mapsto e) \quad p)\end{array} \tag{E7}$$

$$[\mathsf{delete}(x)]p \quad \begin{array}{l}(x \hookrightarrow -) \wedge [\langle x\rangle := ?]p \\ (x \mapsto -) \quad p\end{array} \tag{E8}$$

We require in the axiom for $x := \mathsf{new}(e)$ *that* x *does not appear in* e*, for technical convenience.*

In the sequel E5-8 refer to the corresponding dynamic separation logic equivalences. The proofs of these equivalences are straightforward (consist simply of expanding the semantics of the involved modalities) and therefore omitted.

We have the following separation logic axiomatization of the heap update and heap clear pseudo-operations.

$$[\langle x\rangle := e]p \quad ((x \mapsto -) \quad ((x \mapsto e) \quad p)) \mathbin{\_} ((x \not\mapsto -) \wedge ((x \mapsto e) \quad p))$$
$$[\langle x\rangle := ?]p \quad ((x \mapsto -) \quad p) \mathbin{\_} ((x \not\mapsto -) \wedge p)$$

This axiomatization thus requires a case distinction between whether or not x is allocated.

Note that, letting p be any formula in separation logic , we have that $[x := y]$ in E5 reduces to $[x := y]$ by E1-4. As such, it is possible to eliminate the modality, in the case of the assignment and lookup instructions.

We now want to eliminate the modalities for the heap update and heap clear instructions compositionally in terms of p, because such an elimination would also allow us to eliminate the modalities of the other instructions. What thus remains for a complete axiomatization is a characterization of $[S]b$, $[S](e\ ,!\ e')$, $[S](p\ \ q)$, and $[S](p\ \ q)$, where S denotes one of the two pseudo-instructions. Lemma 4.4.3 provides an axiomatization in DSL of a heap update.

Lemma 4.4.3 (Heap update). *We have the following equivalences for the heap update modality.*

$$[hxi := e]b \quad b; \tag{E9}$$

$$[hxi := e](e'\ ,!\ e'') \quad (x = e' \wedge e'' = e) \_ (x \not\doteq e' \wedge e'\ ,!\ e''); \tag{E10}$$

$$[hxi := e](p\ \ q) \quad ([hxi := e]p\ \ q') \_ (p'\ \ [hxi := e]q); \tag{E11}$$

$$[hxi := e](p\ \ q) \quad p'\ \ [hxi := e]q; \tag{E12}$$

where p' abbreviates $p \wedge (x \not\doteq\ )$ and, similarly, q' abbreviates $q \wedge (x \not\doteq\ )$.

These equivalences we can informally explain as follows. Since the heap update $hxi := e$ does not a□ect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $([hxi := e]b)\ \ b$.

Predicting whether $(e'\ ,!\ e'')$ holds after $hxi := e$, we only need to make a distinction between whether x and e' are aliases, that is, whether they denote the same location, which is simply expressed by $x = e'$. If $x = e'$ then $e'' = e$ should hold, otherwise $(e'\ ,!\ e'')$ (note again, that $hxi := e$ does not a□ect the values of the expressions $e; e'$ and e''). As a basic example, we compute

$[hxi := e](y\ ,!\ \ )$	(definition $y\ ,!\ \ $)
$[hxi := e]9z(y\ ,!\ z)$	(E3)
$9z[hxi := e](y\ ,!\ z)$	(E10)
$9z((y = x \wedge e = z) \_ (y \not\doteq x \wedge (y\ ,!\ z)))$	(semantics SL)
$y \not\doteq x\ \ !\ (y\ ,!\ \ )$	

We use this derived equivalence in the following example:

$[hxi := e](y \not\doteq\ \ )$	(definition $y \not\doteq\ \ $)
$[hxi := e]((y\ ,!\ \ ) \wedge 8z((z\ ,!\ \ )\ !\ z = y))$	(E2, E3, E9)
$[hxi := e](y\ ,!\ \ ) \wedge 8z([hxi := e](z\ ,!\ \ )\ !\ z = y)$	(see above)
$(y \not\doteq x\ !\ (y\ ,!\ \ )) \wedge 8z((z \not\doteq x\ !\ (z\ ,!\ \ ))\ !\ z = y)$	(semantics SL)
$y = x \wedge (\mathsf{emp} \_ (x \not\doteq\ \ ))$	

Predicting whether $(p\ \ q)$ holds after the heap update $hxi := e$, we need to distinguish between whether p or q holds for the sub-heap that contains the (updated) location x. Since we do not assume that x is already allocated, we

instead distinguish between whether p or q holds initially for the sub-heap that does *not* contain the updated location x. As a simple example, we compute

$$
\begin{array}{ll}
[\langle x \rangle := e](\mathsf{true} \mathrel{-\!\!*} (x \mapsto \_)) & \text{(E9,E11)} \\
(\mathsf{true} \mathrel{-\!\!*} ((x \mapsto \_) \wedge (x \not\mapsto \_))) \mathbin{\_} ((x \not\mapsto \_) \mathrel{-\!\!*} [\langle x \rangle := e](x \mapsto \_)) & \text{(see above)} \\
(\mathsf{true} \mathrel{-\!\!*} ((x \mapsto \_) \wedge (x \not\mapsto \_))) \mathbin{\_} ((x \not\mapsto \_) \mathrel{-\!\!*} (\mathsf{emp} \mathbin{\_} (x \mapsto \_))) & \text{(semantics SL)} \\
(\mathsf{true} \mathrel{-\!\!*} \mathsf{false}) \mathbin{\_} ((x \not\mapsto \_) \mathrel{-\!\!*} (\mathsf{emp} \mathbin{\_} (x \mapsto \_))) & \text{(semantics SL)} \\
\mathsf{true} &
\end{array}
$$

Note that this coincides with the above calculation of $[\langle x \rangle := e](y \hookrightarrow \_)$, which also reduces to true, instantiating y by x.

The semantics of $(p \mathrel{-\!\!*} q)$ after the heap update $\langle x \rangle := e$ involves universal quantification over all disjoint heaps that do not contain x (because after the heap update x is allocated). Therefore we simply add the condition that x is not allocated to p, and apply the heap update to q. As a very basic example, we compute

$$
\begin{array}{ll}
[\langle x \rangle := 0]((y \hookrightarrow 1) \mathrel{-\!\!*} (y \hookrightarrow 1)) & \text{(E12)} \\
((y \mapsto 1) \wedge (x \not\mapsto \_)) \mathrel{-\!\!*} [\langle x \rangle := 0](y \hookrightarrow 1)) & \text{(E10)} \\
((y \mapsto 1) \wedge (x \not\mapsto \_)) \mathrel{-\!\!*} ((y = x \wedge 0 = 1) \mathbin{\_} (y \not\mapsto x \wedge y \hookrightarrow 1)) & \text{(semantics SL)} \\
\mathsf{true} &
\end{array}
$$

Note that $(y \hookrightarrow 1) \mathrel{-\!\!*} (y \hookrightarrow 1) \quad \mathsf{true}$ and $[\langle x \rangle := 0]\mathsf{true} \quad \mathsf{true}$.

Proof of Lemma 4.4.3.

E9 $h; s \models [\langle x \rangle := e]b$
- i (semantics heap update modality)
- $h[s(x) := s(e)]; s \models b$
- i (b does not depend on the heap)
- $h; s \models b$

E10 $h; s \models [\langle x \rangle := e](e' \hookrightarrow e'')$
- i (semantics heap update modality)
- $h[s(x) := s(e)]; s \models e' \hookrightarrow e''$
- i (semantics points-to)
- $h[s(x) := s(e)](s(e')) = s(e'')$
- i (definition $h[s(x) := s(e)]$)
- if $s(x) = s(e')$ then $s(e) = s(e'')$ else $h(s(e')) = s(e'')$
- i (semantics assertions)
- $h; s \models (x = e' \wedge e'' = e) \mathbin{\_} (x \not\mapsto e' \wedge e' \hookrightarrow e'')$

E11 $h; s \models [\langle x \rangle := e](p \mathrel{-\!\!*} q)$
- i (semantics heap update modality)
- $h[s(x) := s(e)]; s \models p \mathrel{-\!\!*} q$.
- From here we proceed as follows. By the semantics of separating conjunction, there exist h_1 and h_2 such that $h[s(x) := s(e)] = h_1 \mathbin{]} h_2$, $h_1; s \models p$, and

$h_2; s \models q$. Let $s(x) \in dom(h_1)$ (the other case runs similarly). So $h[s(x) := s(e)] = h_1 \rightarrow h_2$ implies $h_1(s(x)) = s(e)$ and $h = h_1[s(x) := h(x)] \rightarrow h_2$, By the semantics of the heap update modality, $h_1(s(x)) = s(e)$ and $h_1; s \models p$ implies $h_1[s(x) := h(x)]; s \models [\langle x \rangle := e]p$. Since $s(x) \notin dom(h_2)$, we have $h_2; s \models q \wedge x \mapsto !$. By the semantics of separation conjunction we conclude that $h; s \models [\langle x \rangle := e]p \ast q'$ (q' denotes $q \wedge x \mapsto !$).

In the other direction, from $h; s \models [\langle x \rangle := e]p \ast q'$ (the other case runs similarly) we derive that there exist h_1 and h_2 such that $h = h_1 \rightarrow h_2$, $h_1; s \models [\langle x \rangle := e]p$ and $h_2; s \models q'$. By the semantics of the heap update modality it follows that $h_1[s(x) := s(e)]; s \models p$. Since $s(x) \notin dom(h_2)$, we have that $h[s(x) := s(e)] = h_1[s(x) := s(e)] \rightarrow h_2$, and so $h[s(x) := s(e)]; s \models p \ast q$, that is, $h; s \models [\langle x \rangle := e](p \ast q)$.

E12 $h; s \models [\langle x \rangle := e](p \ast q)$
i (semantics of heap update modality)
$h[s(x) := s(e)]; s \models p \ast q$
i (semantics separating implication)
for every h' disjoint from $h[s(x) := s(e)]$: if $h'; s \models p$ then $h[s(x) := s(e)] \rightarrow h'; s \models q$
i (since $s(x) \notin dom(h')$)
for every h' disjoint from h: if $h'; s \models p \wedge x \mapsto !$ then $(h \rightarrow h')[s(x) := s(e)]; s \models q$
i (semantics of heap update modality)
for every h' disjoint from h: if $h'; s \models p \wedge x \mapsto !$ then $h \rightarrow h'; s \models [s(x) := s(e)]q$
i (semantics separating implication)
$h; s \models (p \wedge x \mapsto !) \ast [\langle x \rangle := e]q$.

The equivalences for the heap clear modality in the following lemma can be informally explained as follows. Since $\langle x \rangle := ?$ does not affect the store, and the evaluation of a Boolean condition b only depends on the store, we have that $b[\langle x \rangle := ?] = b$. For $e, ! e'$ to hold after executing $\langle x \rangle := ?$, we must initially have that $x \in e$ and $e, ! e'$. As a simple example, we have that $\forall y; z(y \mapsto ! z)$ characterizes the empty heap. It follows that $[\langle x \rangle := ?](\forall y; z(y \mapsto ! z))$ is equivalent to $\forall y; z(\neg(y \in x \wedge y, ! z))$. The latter first-order formula is equivalent to $\forall y; z(y = x \vee y \mapsto ! z)$. This assertion thus states that the domain consists at most of the location x, which indeed ensures that after $\langle x \rangle := ?$ the heap is empty. To ensure that $p \ast q$ holds after clearing x it suffices to show that the initial heap can be split such that both p and q hold in their respective sub-heaps with x cleared. The semantics of $p \ast q$ after clearing x involves universal quantification over all disjoint heaps that do may contain x, whereas before executing $\langle x \rangle := ?$ it involves universal quantification over all disjoint heaps that do *not* contain x, in case x is allocated initially. To formalize in the initial configuration universal quantification over all disjoint heaps we distinguish between all disjoint heaps that do not contain x and *simulate* all disjoint heaps that contain x by interpreting both p and q in $p \ast q$ in the context of heap updates $\langle x \rangle := y$ with *arbitrary* values y for the location x.

As a very basic example, consider $[hxi := ?]((x \mapsto\ !\ 0) \ast (x \mapsto\ !\ 0))$, which should be equivalent to **true**. The left conjunct $((x \mapsto\ !\ 0) \wedge (x \not\mapsto\ !\ ))$ $[hxi := ?](x \mapsto\ !\ 0))$ of the resulting formula after applying E16 is equivalent to **true** (because $(x \mapsto\ !\ 0) \wedge (x \not\mapsto\ !\ )$ is equivalent to **false**). We compute the second conjunct (in the application of E10 we omitted some trivial reasoning steps):

$$\begin{array}{ll} \forall y([hxi := y](x \mapsto\ !\ 0) \quad [hxi := y](x \mapsto\ !\ 0) & \text{(E10)} \\ \forall y(y = 0 \quad y = 0) & \text{(semantics SL)} \\ \textbf{true} & \end{array}$$

Lemma 4.4.4 (Heap clear). *We have the following equivalences for the heap clear modality.*

$$[hxi := ?]b \quad b; \tag{E13}$$

$$[hxi := ?](e \mapsto\ !\ e') \quad (x \not\mapsto e) \wedge (e \mapsto\ !\ e'); \tag{E14}$$

$$[hxi := ?](p \ast q) \quad [hxi := ?]p \ast [hxi := ?]q; \tag{E15}$$

$$[hxi := ?](p \quad q) \quad \begin{array}{l} ((p \wedge x \not\mapsto\ !\ ) \quad [hxi := ?]q) \wedge \\ \forall y([hxi := y]p \quad [hxi := y]q); \end{array} \tag{E16}$$

where y is fresh.

Proof. Here we go.

E13 $[hxi := ?]b \quad b$. As above, it suffices to observe that the evaluation of b does not depend on the heap.

E14 $h; s \models [hxi := ?](e \mapsto\ !\ e')$
iff (semantics heap clear modality)
$h[hs(x)i := ?]; s \models e \mapsto\ !\ e'$
iff (semantics points-to)
$s(e) \in dom(h[hs(x)i := ?])$ and $h[hs(x)i := ?](s(e)) = h(s(e)) = s(e')$
iff (semantics assertions)
$h; s \models x \not\mapsto e \wedge e \mapsto\ !\ e'$

E15 $h; s \models [hxi := ?](p \ast q)$
iff (semantics heap clear modality)
$h[hs(x)i := ?]; s \models p \ast q$
iff (semantics separating conjunction)
$h_1; s \models p$ and $h_2; s \models q$, for some $h_1; h_2$ such that $h[hs(x)i := ?] = h_1 \bullet h_2$
iff (semantics heap clear modality)
$h_1; s \models [hxi := ?]p$ and $h_2; s \models [hxi := ?]q$, for some $h_1; h_2$ such that $h = h_1 \bullet h_2$.
Note: $h = h_1 \bullet h_2$ implies $h[hs(x)i := ?] = h_1[hs(x)i := ?] \bullet h_2[hs(x)i := ?]$, and, conversely, $h[hs(x)i := ?] = h_1 \bullet h_2$ implies there exists $h_1^0; h_2^0$ such that $h = h_1^0 \bullet h_2^0$ and $h_1 = h_1^0[hs(x)i := ?]$ and $h_2 = h_2^0[hs(x)i := ?]$.

E16 $h;s \models [\langle x\rangle := ?](p \to q)$
i□ (semantics heap clear modality)
$h[s(x) := ?];s \models p \to q$.
From here we proceed as follows. First we show that $h;s \models ((p \wedge x \not\in !) \to [\langle x\rangle := ?]q)$ and $h;s \models \forall y([\langle x\rangle := y]p \to [\langle x\rangle := y]q)$ implies $h[s(x) := ?];s \models p \to q$. Let h^0 be disjoint from $h[s(x) := ?]$ and $h^0;s \models p$. We have to show that $h[s(x) := ?] \bullet h^0;s \models q$. We distinguish the following two cases.

- First, let $s(x) \in dom(h^0)$. We then introduce $s^0 = s[y := h^0(s(x))]$. We have $h^0;s^0 \models p$ (since y does not occur in p), so it follows by the semantics of the heap update modality that $h^0[s(x) := ?];s^0 \models [\langle x\rangle := y]p$. Since $h^0[s(x) := ?]$ and h are disjoint (which clearly follows from that h^0 and $h[s(x) := ?]$ are disjoint), and since $h;s^0 \models [\langle x\rangle := y]p \to [\langle x\rangle := y]q$, we have that $h \bullet (h^0[s(x) := ?]);s^0 \models [\langle x\rangle := y]q$. Applying again the semantics of the heap update modality, we obtain $(h \bullet (h^0[s(x) := ?]))[s(x) := s^0(y)];s^0 \models q$. We then can conclude this case observing that y does not occur in q and that $h[s(x) := ?] \bullet h^0 = (h \bullet (h^0[s(x) := ?]))[s(x) := s^0(y)]$.
- Next, let $s(x) \notin dom(h^0)$. So h^0 and h are disjoint, and thus (since $h;s \models (p \wedge x \not\in !) \to [\langle x\rangle := ?]q$) we have $h \bullet h^0;s \models [\langle x\rangle := ?]q$. From which we derive $(h \bullet h^0)[s(x) := ?];s \models q$ by the induction hypothesis. We then can conclude this case by the observation that $h[s(x) := ?] \bullet h^0 = (h \bullet h^0)[s(x) := ?]$.

Conversely, assuming $h[s(x) := ?];s \models p \to q$; we first show that $h;s \models (p \wedge x \not\in !) \to [\langle x\rangle := ?]q$ and then $h;s \models \forall y([\langle x\rangle := y]p \to [\langle x\rangle := y]q)$:

- Let h^0 be disjoint from h and $h^0;s \models p \wedge x \not\in !$. We have to show that $h \bullet h^0;s \models [\langle x\rangle := ?]q$, that is, $(h \bullet h^0)[s(x) := ?];s \models q$ (by the semantics of the heap clear update). Clearly, $h[s(x) := ?]$ and h^0 are disjoint, and so $h[s(x) := ?] \bullet h^0;s \models q$ follows from our assumption. We then can conclude this case by the observation that $(h \bullet h^0)[s(x) := ?] = h[s(x) := ?] \bullet h^0$, because $s(x) \notin dom(h^0)$.
- Let h^0 be disjoint from h and $s^0 = s[y := n]$, for some n such that $h^0;s^0 \models [\langle x\rangle := y]p$. We have to show that $h \bullet h^0;s^0 \models [\langle x\rangle := y]q$. By the semantics of the heap update modality it follows that $h^0[s(x) := n];s^0 \models p$, that is, $h^0[s(x) := n];s \models p$ (since y does not occur in p). Since $h^0[s(x) := n]$ and $h[s(x) := ?]$ are disjoint, we derive from the assumption $h[s(x) := ?];s \models p \to q$ that $h[s(x) := ?] \bullet h^0[s(x) := n];s \models q$. Again by the semantics of the heap update modality we have that $h \bullet h^0;s^0 \models [\langle x\rangle := y]q$ i□ $(h \bullet h^0)[s(x) := n];s^0 \models q$ (that is, $(h \bullet h^0)[s(x) := n];s \models q$, because y does not occur in q). We then can conclude this case by the observation that $(h \bullet h^0)[s(x) := n] = h[s(x) := ?] \bullet h^0[s(x) := n]$. □

We denote by E the *rewrite system* obtained from the equivalences E1-16 by orienting these equivalences from left to right, e.g., equivalence E1 is turned into

a rewrite rule $[S]\mathsf{false} \rightarrow \mathsf{false}$. The following theorem states that the rewrite system E is complete, that is, confluent and strongly normalizing. Its proof is straightforward (using standard techniques) and therefore omitted.

Theorem 4.4.5 (Completeness of E).

- Normal form. *Every standard formula of separation logic is in normal form (which means that it cannot be reduced by the rewrite system* E*).*
- Local confluence. *For any two reductions $p \rightarrow q_1$ and $p \rightarrow q_2$ (p a formula of DSL) there exists a DSL formula q such that $q_1 \rightarrow q$ and $q_2 \rightarrow q$.*
- Termination. *There does not exist an infinite chain of reductions $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow \ldots$.*

We now show an example of the interplay between the modalities for heap update and heap clear. We want to derive

$$\{\forall x((x \mapsto -) \rightarrow p)\}\ x := \mathsf{new}(0); \mathsf{delete}(x)\ \{p\}$$

where statement $x := \mathsf{new}(0); \mathsf{delete}(x)$ simulates the so-called random assignment [107]: the program terminates with a value of x that is chosen non-deterministically. First we apply the axiom E8 for de-allocation to obtain

$$\{(x, -) \wedge [\langle x \rangle := ?]p\}\ \mathsf{delete}(x)\ \{p\}.$$

Next, we apply the axiom E8 for allocation to obtain

$$\begin{array}{c}\{\forall x((x, -) \rightarrow [\langle x \rangle := 0]((x, -) \wedge [\langle x \rangle := ?]p))\}\\ x := \mathsf{new}(0)\\ \{(x, -) \wedge [\langle x \rangle := ?]p\}.\end{array}$$

Applying E10 (after pushing the heap update modality inside), followed by some basic first-order reasoning, we can reduce $[\langle x \rangle := 0](\exists y(x, y))$ to true. So we obtain

$$\begin{array}{c}\{\forall x((x, -) \rightarrow [\langle x \rangle := 0][\langle x \rangle := ?]p)\}\\ x := \mathsf{new}(0)\\ \{(x, -) \wedge [\langle x \rangle := ?]p\}.\end{array}$$

In order to proceed we formalize the interplay between the modalities for heap update and heap clear by the following general equivalence:

$$[\langle x \rangle := e][\langle x \rangle := ?]p \quad [\langle x \rangle := ?]p$$

We then complete the proof by applying the sequential composition rule and consequence rule, using the above equivalence and the following axiomatization of the heap clear modality:

$$(x \mapsto -) \wedge [\langle x \rangle := ?]p \quad (x \mapsto -) \wedge p$$

Now it is possible to define meta-operations on formulas of separation logic .

Definition 4.4.6. We define the meta-operations $[\langle x \rangle := y]$ and $[\langle x \rangle := ?]$:

$$[\langle x \rangle := y] = [\langle x \rangle := y] \ ;$$

and

$$[\langle x \rangle := ?] = [\langle x \rangle := ?] \ :$$

Note that due to Theorem 4.4.5, we can completely eliminate the modality when it is applied to a formula of separation logic. Hence the resulting formulas are again formulas of separation logic, and no longer in the extended language of DSL.

The above axiomatization can be extended in the standard manner to a program logic for sequential while programs, see [107], which does not require the frame rule, nor any other adaptation rule besides the consequence rule. For recursive programs however one does need more adaptation rules: a further discussion about the use of the frame rule in a relative completeness proof for recursive pointer programs is outside the scope of this thesis, and left for future work.

4.5 Alternative axiomatizations

Based on the heap update and heap clear pseudo-instructions of the previous section, we can give two alternative axiomatizations of Reynolds' logic. It is remarkable that these alternative axiomatizations can be proven to be also the weakest preconditions, respectively strongest postconditions, of the primitive operations of pointer programs.

Definition 4.5.1. The set of *alt-backwards* axioms is:

(assign) $\vdash \{p[x := y]\}\ x := y\ \{p\}$,

(lookup) $\vdash \{\exists z((y \mapsto z) \wedge [x := z])\}\ x := [y]\ \{\}$ where z is fresh,

(mutation) $\vdash \{(x \mapsto \_)\}\ [\langle x \rangle := y]\ [x] := y\ \{\}$,

(allocation) $\vdash \{\forall x((x \not\mapsto \_) \rightarrow [\langle x \rangle := y])\}\ x := \mathsf{new}(y)\ \{\}$ where $x \neq y$,

(deallocation) $\vdash \{(x \mapsto \_)\}\ [\langle x \rangle := ?]\ \mathsf{delete}(x)\ \{\}$.

These alternative backwards axioms also express the weakest precondition:

Lemma 4.5.2. *The* alt-backwards *axioms are sound with respect to* MSL*, and describe the weakest precondition with respect to the given primitive operations and postcondition.*

We can also give the set of *alt-forwards* axioms, in the sense that these axioms allow reasoning forwards from a given precondition.

Definition 4.5.3. The set of *alt-forwards* axioms is:

(assign) $\vdash \{\ \}\ x := y\ \{\exists z(\ [x := z] \wedge y[x := z] = x)\}$,

(lookup) $\vdash \{\mathit{ok}(y)\}\ x := [y]\ \{\exists z(\ [x := z] \wedge y[x := z] \mapsto x)\}$,

(mutation) $\vdash \{\mathit{ok}(x)\}\ [x] := y\ \{\forall z(\ [\mathit{hx}i := z]) \wedge (x \mapsto y)\}$,

(allocation) $\vdash \{\ \}\ x := \mathsf{new}(y)\ \{\forall z(\ [x := z]))[\mathit{hx}i := ?] \wedge (x \mapsto y)\}$,

(deallocation) $\vdash \{\mathit{ok}(x)\}\ \mathsf{delete}(x)\ \{\forall z(\ [\mathit{hx}i := z]) \wedge (x \not\mapsto)\}$,

where z is fresh.

Lemma 4.5.4. *The* alt-forwards *axioms are sound with respect to* MSL*, and describe the strongest postcondition with respect to the given primitive operations and precondition.*

Another application of the modality for the heap update and heap clear pseudo-instructions is that we are able to prove the completeness of the local axioms of Reynolds' logic and the frame rule, without employing the separating implication as the invariant formula.

Theorem 4.5.5 (Completeness local axioms). *For any primitive pointer operation S, if $\models \{p\}\ S\ \{q\}$ then $\{\ \}\ S\ \{\ \}$ is derivable from the* local *axioms and the frame rule, the consequence rule, and the invariance rule for basic assignment and look-up.*

Proof. Let $\models \{\ \}\ S\ \{\ \}$.

Basic assignment By the invariance rule for basic assignments, we first derive

$$\{\mathsf{true} \wedge \forall x(\ )\}\ x := e\ \{x = e \wedge \forall x(\ )\}$$

Clearly, $\ $ implies $\forall x(\ )$. Let $h; s \models x = e$, that is, $s(x) = s(e)$, and $h; s[x := n] \models \ $, for some n. From the assumption $\models \{\ \}\ x := e\ \{\ \}$ we then derive $h; s[x := s[x := n](e)] \models \ $, that is, $h; s \models \ $ (since $s[x := n](e) = s(e) = s(x)$).

Look-up By the restricted invariance rule, we first derive

$$\{\forall x(\ ) \wedge (e \mapsto \ )\}\ x := [e]\ \{\forall x(\ ) \wedge (e \mapsto x)\}$$

Since $\models \{\ \}\ x := [e]\ \{\ \}$, we have that $\ $ implies $e \mapsto \ $, and so $\ $ implies $\forall x(\ ) \wedge (e \mapsto \ )$. On the other hand, let $h(s(e)) = s(x)$ and $h; s' \models \ $, where $s' = s[x := n]$, for some n. From the assumption $\models \{\ \}\ x := [e]\ \{\ \}$ we then derive $h; s[x := h(s'(e))] \models \ $, that is, $h; s \models \ $ (since x does not occur in e and $h(s(e)) = s(x)$, we have that $s[x := h(s'(e))] = s[x := h(s(e))] = s$).

Mutation Let $'$ denote $\exists y.(\ [\langle x\rangle := y])$. By the frame rule, we first derive

$$f(x \mapsto \_)\ \ 'g\ [x] := e\ f(x \mapsto e)\ \ 'g$$

Let $h;s \models \_$. We show that $h;s \models (x \mapsto \_)\ \ '$: Since $\models f\ g\ [x] := e\ f\ g$ we have that $s(x) \in dom(h)$. So we can introduce the split $h = h_1 \bullet h_2$ such that $h_1;s \models x \mapsto \_$ and $h_2 = h[s(x) := ?]$. By the above substitution lemma it then suffices to observe that $h_2;s[y := h(s(x))] \models [\langle x\rangle := y]$ if and only if $h_2[s(x) := h(s(x))];s \models \_$ (y does not appear in $\_$), that is, $h;s \models \_$. On the other hand, we have that $(x \mapsto e)\ '$ implies $\_$: Let $h;s \models (x \mapsto e)\ '$. So there exists a split $h = h_1 \bullet h_2$ such that $h_1;s \models x \mapsto e$ and $h_2;s \models '$. Let n be such that $h_2;s[y := n] \models [[x] := y]$. By the above substitution lemma we have that $h_2;s[y := n] \models [\langle x\rangle := y]$ if and only if $h_2[s(x) := n];s \models \_$ (y does not appear in $\_$). Since $\models f\ g\ [x] := e\ f\ g$ it then follows that $h_2[s(x) := s(e)];s \models \_$, that is, $h;s \models \_$ (note that $h = h_2[s(x) := s(e)]$ because $h(s(x)) = s(e)$ and $h_2 = h[s(x) := ?]$).

Allocation By the frame rule, we first derive

$$f\mathsf{emp}\ \ \exists x.(\ \_)g\ x := \mathsf{new}(e)\ f(x \mapsto e)\ \ \exists x.(\ \_)g$$

Clearly, $\_$ implies $\mathsf{emp}\ \ \exists x.(\ \_)$. On the other hand, let $h;s \models (x \mapsto e)\ \ \exists x.(\ \_)$. So there exists a split $h = h_1 \bullet h_2$ such that $h_1;s \models x \mapsto e$ and $h_2;s[x := n] \models \_$, for some n. Since $\models f\ g\ x := \mathsf{new}(e)\ f\ g$, we derive that $h_2[s(x) := s[x := n](e)];s \models \_$, that is, $h;s \models \_$ (note that $s(x) \notin dom(h_2)$ and, since x does not appear in e, we have $s[x := n](e) = s(e)$, and thus $h = h_2[s(x) := s(e)]$).

Dispose Let $'$ denote $(x \not\mapsto \_) \wedge \exists y.(\ [\langle x\rangle := y])$. By the frame rule, we first derive

$$f(x \mapsto \_)\ \ 'g\ [x] := ?\ f\mathsf{emp}\ \ 'g$$

See above (mutation) for the kind of argument that establishes that $\_$ implies $(x \mapsto \_)\ '$. On the other hand, Let $h;s \models \mathsf{emp}\ '$, that is, $h;s \models '$, and so by the above substitution lemma, we have $h[s(x) := n];s \models \_$, for some n (again, y does not appear in $\_$). Since $f\ g\ [x] := ?\ f\ g$, we derive $h[s(x) := ?];s \models \_$, that is, $h;s \models \_$, since $h;s \models x \not\mapsto \_$. □

Chapter 5

Conclusion

In this thesis, we investigated new foundations for separation logic. These new foundations are presented in two parts: in the first part, we presented a model theoretic investigation of separation logic with the aim of finding an adequate semantics, and novel finitary proof systems for separation logic for which we showed soundness and completeness. In the second part, we proposed a new interpretation of Reynolds' logic in such a way that it is compatible with the semantics of the first part: it turned out that all axioms are still sound and relatively complete with respect to the new interpretation. We also introduced dynamic separation logic, and showed that it yields an alternative axiomatization of Reynolds' logic that avoids introducing separating connectives when generating weakest preconditions and strongest postconditions.

Summary

To see more clearly the bigger picture, we summarize this thesis in Table 5.1. The table displays the different subjects studied and references various sections. As a starting point of this thesis we considered background knowledge, shown in the left column of Table 5.1: first-order logic, Hoare's logic, and dynamic logic.

- First-order logic is a well-known assertion language with a rich model theory and proof theory, and Gödel's completeness theorem connects these two views on first-order logic. Chapter A of the appendix revisits the necessary background knowledge on first-order and higher-order logic.
- Hoare's logic is a program logic useful for reasoning about program correctness. A program S is provably correct with respect to a specification that specifies the precondition and postcondition , denoted by the Hoare triple $f\ g\ S\ f\ g$, if it is the case that such Hoare triple can be deduced from Hoare's axioms. The proof system of Hoare's logic can be given a rich program semantics, for which it can be shown that there is a soundness and relative completeness result. Chapter B of the appendix revisits the necessary background knowledge on Hoare's logic, but presents it in a general way that is more suitable for the rest of this thesis.

First-order logic (FOL)	Separation logic (SL)
• well-known assertion language (see Section A.1)	• extends FOL with connectives , (see Section 2.1)
• rich model theory (see Section A.2)	• *model theoretic investigation* (see Section 2.2, 2.3)
• rich proof theory (see Section A.3)	• *general semantics and proof theory* (see Section 3.1, 3.3)
• Gödel's completeness theorem (see Section A.4)	• *soundness and completeness* (see Section 3.2, 3.4)
Hoare's logic (HL)	**Reynolds' logic (RL)**
• Hoare triple $f\ g\ S\ f\ g$ (see Section B.1)	• *semantics relative to memory models* (see Section 4.1)
• rich program semantics (see Section B.2, B.3)	• *richer* program semantics (see Section 4.2)
• Cook's relative completeness (see Section B.4)	• *soundness and relative completeness* (see Section 4.3)
Dynamic logic (DL)	**Dynamic separation logic (DSL)**
• extends FOL with modality [*S*]	• *extends DL with connectives ,* (see Section 4.4)
• HL is embedded in DL	• *RL is embedded in DSL* (see Section 4.5)
• DL is more expressive than HL	

Table 5.1: One-page summary of this thesis. Novel contributions are *emphasized*.

- Dynamic logic is an extension of first-order logic, by adding a so-called program modality to the assertion language. It is the case that Hoare's logic can be embedded in dynamic logic: every program that can be proven to be correct in Hoare's logic also can be proven to be correct in dynamic logic. However, dynamic logic is more expressive than Hoare's logic. The reader can consult [107] for more background knowledge on dynamic logic.

This thesis continued study of the subjects shown in the right column of Table 5.1: separation logic, Reynolds' logic, and dynamic separation logic. The first contribution is nominal, by disambiguating *separation logic* from *Reynolds' logic* (where the former strictly speaks of the logic of the assertion language, and the latter speaks of the program logic).

- Separation logic is an extension of first-order logic that introduces two new connectives. We recalled the syntax of separation logic, and have shown that its standard semantics is inadequate. Searching for an adequate semantics has given several interesting results: full separation logic is inadequate, and several fragments of the syntax of separation logic are also inadequate. We then have introduced a more general semantics, akin to Henkin's general semantics, for separation logic and defined two proof systems for deducing consequences: one sequent calculus, and one natural deduction system. We have shown that for both we have soundness and completeness, and this settles the open problem of adequacy for separation logic (the separation logic counterpart to Gödel's completeness theorem connecting model theory and proof theory).

- Reynolds' logic is an extension of Hoare's logic in two ways: Hoare triples $f\ g\ S\ f\ g$ now speak about pointer programs S too, and the precondition and postcondition are now assertions of separation logic. All Hoare triples provable from Hoare's axioms are also provable in Reynolds' logic, but Reynolds' logic proves more triples than in Hoare's logic (e.g. those involving separating connectives). We have given a general semantics to Reynolds' logic relative to memory models, and we have based the program semantics on top of the semantics of the primitive operations of pointer programs: lookup, mutation, allocation, deallocation. Reynolds' axioms, and the frame rule in particular, are still sound and relatively complete in the more general setting, even when we allow for infinite heaps.

- Dynamic separation logic is an extension of first-order dynamic logic by adding the separating connectives, similar to how separation logic adds the separating connectives to first-order logic. First-order dynamic separation logic is a novel subject, not studied in the literature before. We have seen how to give semantics to dynamic separation logic, and we have investigated a calculus for reducing program modalities involving the primitive pointer-manipulating operations. Since it is also the case that Reynolds' logic can be embedded in dynamic separation logic, this resulted in an alternative axiomatization of Reynolds' logic that analyzes the logical structure of pre- or postconditions.

Future work

In this section, we discuss suggestions of directions for further study. One direction is to systematically compare the different semantics for separation logic that were introduced in this thesis. The next direction is to fully formalize the results presented in this thesis in a proof assistant, and the soundness and completeness of the proof systems for separation logic in particular, to increase our confidence in the results. The third direction is to enhance tool support for reasoning about separation logic. Another direction is to investigate relative completeness in the case of recursive procedures in such a setting that the frame rule is necessary. Further directions are investigating other separation logic variants, such as intuitionistic separation logic and permission-based separation logic, and other program logics, such as concurrent separation logic. A last direction is to investigate the Curry-Howard correspondence for our new natural deduction proof system for separation logic, and whether our new semantics also works with proof systems such as the logic of bunched implications.

Different semantics

In the chapters introducing new semantics for separation logic and Reynolds' logic, we have seen the following interpretations of separation logic:

- SSL for Standard Separation Logic which interprets separation logic formulas with respect to all finite heaps (see Definition 2.2.1),
- FSL for Full Separation Logic which interprets separation logic formulas with respect to all (finite or infinite) heaps (see Definition 2.3.1),
- GSL for General Separation Logic which interprets separation logic formulas with respect to a fixed set of (finite or infinite) heaps (see Definition 4.1.2),
- MSL for Memory Separation Logic which interprets separation logic formulas with respect to memory models, a set of heaps that satisfies a number of closure conditions (see Definition 4.1.6) necessary for showing the soundness and relative completeness of Reynolds' logic.

The valid formulas of separation logic are included in the following ways:

- the set of valid formulas according to GSL are included in the set of valid formulas according to MSL,
- the set of valid formulas according to MSL are included in the set of valid formulas according to SSL and also in the set of valid formulas according to FSL.

So, it is possible to see MSL as a generalized interpretation that does not depend on the finiteness condition of the heap.

To obtain a sound and complete proof system, we relaxed the condition that the heap is functional (in the sense of a functional relation) to obtain the following

relational separation logic interpretations (not to be confused with relational separation logic as investigated by Yang [231]):

- WRSL for Weak Relational Separation Logic which interprets separation logic formulas with respect to relational structures with respect to all finite relations (see Definition 2.5.1),
- FRSL for Full Relational Separation Logic which interprets separation logic formulas with respect to relational structures with respect to all relations (see Definition 2.5.2),
- GRSL for General Relational Separation Logic which interprets separation logic formulas with respect to relational structures and a fixed set of relations (see Definition 3.4.2),
- RSL for Relational Separation Logic which interprets separation logic formulas with respect to comprehensive relational structures (see Definition 3.4.4) necessary for obtaining an adequate and finitary proof system.

The interpretations of SSL, FSL and GSL can be embedded in a natural way in the interpretations of WRSL, FRSL and GRSL, respectively.

The interpretations RSL and MSL are important, since the finitary proof system for separation logic we present in this thesis is sound and complete with respect to RSL, whereas Reynolds' logic is sound and relatively complete with respect to MSL. Hence, by taking the intersection of these two interpretations, we obtain our final desired interpretation of separation logic and Reynolds' logic.

Since it is possible to instantiate Reynolds' logic with different interpretations of separation logic (such as the standard interpretation, full interpretation, or general interpretations such as those restricted to first-order definable heaps and memory models), each interpretation of separation logic induces a different set of valid formulas which can be used in the consequence rule. This situation is similar to that of the different interpretations of higher-order logic (weak interpretation, full interpretation, Henkin interpretation), where each interpretation induces a different set of valid formulas. The same formula can thus be interpreted differently, and it remains future work to study how these interpretations are related. It also remains future work to study the semantic relation between the different instances of Reynolds' program logic with different interpretations of separation logic.

Formalization

From the position of the formalist, who rejects meaning and truth in mathematics that transcends literal symbols and their formation rules, much of mathematics can be only understood by being based on some axiomatic treatment of set theory (or any other foundational theory). As such, a piece of mathematics is only valid to the formalist in so far that it can be formalized and shown to follow from axioms. The ideal formalist never skips any detail and never needs an appeal to intuition.

Although much formal detail is provided, also in this thesis some details are only sketched out and there were some appeals to intuition. To further increase

confidence in results, one could formalize mathematical statements using a proof assistant and meticulously check the argument by also formalizing its proof and filling in all the details. As such, we have indeed formalized important definitions and results of Chapter 3 and Chapter 4 in the Coq proof assistant, see Chapter D in the appendix. However, not all results presented in this thesis have been formalized and checked by a proof assistant, and as such there remains a threat to validity.

Another direction of future work is to formalize the semantics of separation logic and our suggested proof systems, and verify the soundness and completeness result by means of a proof assistant. Such future work could increase confidence in the claimed results of this thesis.

Tool support

Since the alternative axiomatization of Reynolds' logic we arrived at by introducing dynamic separation logic and the original axiomatization given by Reynolds himself are both weakest precondition axiomatizations, we have that they must be equivalent. However, evaluating existing tools for reasoning about separation logic on simple programs and postconditions, on formulas that express the equivalence between Reynolds' weakest precondition and our alternative weakest precondition, results in bugs or incompleteness.

Recall from the introduction that we have seen the generation of two weakest preconditions for the program $[x] := 0$ and the postcondition $(y \mapsto z)$:

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

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

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

We have that (1.1) and (1.3) are equivalent. Surprisingly, a proof of the equivalence exceeds the capability of all the automatic separation logic provers in the benchmark competition for separation logic [201]. In particular, of the automatic provers, only the CVC4-SL tool [186] supports the fragment of separation logic that includes the separating implication connective. However, from our own experiments with that tool, we found that it produces an incorrect counter-example and reported this as a bug to one of the maintainers of the project (Andrew Reynolds). In fact, the latest version, CVC5-SL, reports the same input as 'unknown', indicating that the tool is incomplete. In the case of (semi-)interactive separation logic provers or proof assistants (such as Iris [132], and VerCors [12, 156] that uses Viper [159] as a back-end) we sought out expertise and collaborated in our search for a tool-supported proof of the above equivalence. Even after personally visiting the Iris team in Nijmegen (lead by Robbert Krebbers) and the VerCors team in Twente (lead by Marieke Huisman), we were unable to guide the tools to produce a proof of equivalence. The problem here seems similar to that of [122], in that their semantics of the separating connectives, which are formalized in terms of

abstract monoids, are not compatible with the set-theoretic interpretation of the points-to relation. However, only further investigation can explain what is the reason existing tools contain bugs or are incomplete.

Another direction of future work is to develop proof assistants and automatic provers based on the novel semantics and proof theory introduced in this thesis. A concrete direction of future study is to improve the KeY system, which is based on a variant of dynamic logic supporting Java programs called JavaDL, to extend its implementation of dynamic logic to also work with separation logic formulas, which could improve the e□ciency for reasoning about aliasing in Java programs.

Recursive procedures

In this thesis we have only shown the relative completeness result of Reynolds' logic for the primitive operations of pointer programs. Our result can be readily extended to also cover relative completeness for the entire programming language, including the complex programs, as in [81, 209]. In fact, in the relative completeness result of [81], there is no need for the frame rule by arithmetically encoding finite heaps. It remains future work to show how to extend Reynolds' proof system to recursive procedures, in which it is possible to give a relative completeness result that is based on the frame rule and does not depend on an encoding of the heap (preliminary results are submitted to a conference for peer review).

Other separation logics

The results presented in this thesis are based on the classical interpretation of separation logic. There is also an intuitionistic interpretation of separation logic [187], based on which one could conceive a corresponding and novel intuitionistic dynamic separation logic. In fact, preliminary results have already indicated that much of the work presented in this thesis can be repeated: a novel alternative weakest precondition for intuitionistic separation logic can be defined. In fact, by following the approach of this thesis, also a novel strongest postcondition can be given for intuitionistic separation logic—which, as far as the author knows, has not yet been given in the literature before. These preliminary results are presented in Appendix C.

Also there are many variants of separation logic, such as probabilistic separation logic [16], permission-based separation logic [33], set-based separation logic [110], strong-separation logic [172]. It remains future work to investigate how to adapt the semantics introduced in this thesis to those di□erent settings.

Other program logics

O'Hearn and Brookes have introduced concurrent separation logic, which is an extension of Reynolds' logic for reasoning about concurrent shared-memory pointer programs [163, 36, 37, 38]. The logic introduced by O'Hearn and Brookes is a program logic: they use the same assertion language, separation logic, as is investigated in this thesis. Their approach is based on the Owicki-Gries proof

method [171] published in 1976. There were two papers published by S. Owicki and D. Gries in 1976, one published by the ACM [171] and the other published by Springer [170]. Both papers were based on the work presented in Owicki's Ph.D. thesis. The ACM paper explains her work in a more simplified setting, and makes use of resources which protect program variables and ensures that the execution of critical sections are mutually exclusive, i.e. never two critical sections are executed at the same time. This is also the starting point for O'Hearn and Brookes when introducing concurrent separation logic.

The Springer paper, however, o□ers a more 'primitive tool, so primitive that other methods for synchronization such as semaphores and events can be easily described using it.' The Springer paper is more general and allows 'to prove correctness for programs of such a fine degree of interleaving that the only mutual exclusion need be the memory reference.' [170] In this general setting, proving correctness involves showing that parallel programs are interference free, by using an interference freedom test on the proof outlines of the parallel components. In contrast, interference freedom is a property that comes for free by using resources and certain syntactic restrictions, as presented in the ACM paper: 'complexity of parallel programs stems from the way processes can interfere with each other as they use shared variables. The critical section statement reduces these problems by guaranteeing that only one process at a time has access to the variables in a resource. The following syntax restrictions ensure that...' [171]

It remains future work to see whether Reynolds' logic as presented in this thesis can also be extended to concurrent pointer programs, where the only mutual exclusion that is provided by the programming language is at the level of accessing or mutating a single memory reference and interference freedom tests now involves checking proof outlines where assertions are formulated in the language of separation logic (preliminary results are submitted to a conference for peer review). Also other program logics, such as separation logic with higher-order store [28], can be investigated.

Curry-Howard correspondence

A well-known connection between logic and computation, the Curry-Howard correspondence, is often summarized by the slogans 'propositions as types' and 'programs as proofs' [220]. In general, one speaks of a Curry-Howard correspondence whenever it is possible to relate proofs in a proof theory on the one hand with terms in a calculus on the other hand, in such a way that proof normalization of the proof theory corresponds with term reduction in the calculus.

In this thesis we have introduced a natural deduction proof system for reasoning about separation logic. Further work can be done in investigating meta-properties of this proof system, such as proof normalization and normalization strategies. One could investigate this by asking the question: what are the terms and rewrite rules of the lambda-like calculus such that the Curry-Howard correspondence holds with respect to the natural deduction proof system we introduced in this thesis?

Furthermore, one could investigate existing proof systems for separation logic,

such as the proof system of bunched implications by O'Hearn and Pym [167, 181]. Is it possible to adapt existing proof systems to work with the Henkin-like semantics introduced in this thesis, and is it also there possible to obtain soundness and completeness results? What are the ramifications of our novel semantics to existing work [88] that investigates the Curry-Howard correspondence for the proof system of bunched implications?

Appendix A

Classical (higher-order) logic

In this chapter we recall the definitions and results of classical logic, upon which the rest of this thesis depends (see also [135, 217, 213, 214, 203, 31, 152, 91, 9, 82]). Readers already familiar with classical logic may quickly skim this chapter: no new results are presented. However, this chapter is included for the purpose of completeness.[1]

Logic is used as a formal description language. With such language, we give descriptions with an intention to *describe* or *assert* what is the case in a universe. Logic has versatile uses in computer science. For example, logic can be used to describe the universe as seen from the perspective of a computer. To be more specific, we can use logic to describe the possible memory states of a computer at a particular instant in time. Primitive descriptions can fix what values are held in certain places of memory, or describe relations between the values such places of memory hold. Complex descriptions are constructed by composing simpler descriptions, e.g. by conditionals between descriptions or quantification over possible values.

There are different orders of languages. In a zeroth-order language one can speak of primitive propositions and their logical connection. In a first-order language one furthermore has the ability to speak about, and quantify over, *individuals*, which are the elements in a universe, or domain of discourse. This distinguishes a first-order language from a zeroth-order language, since in the latter one cannot quantify over individuals. Sometimes first-order logic is called predicate logic, whereas zeroth-order logic is called propositional logic. In a second-order language one goes beyond the ability to quantify over individuals, and one can also quantify over properties of individuals (see also [216]). In a third-order language, one can quantify over properties of properties of individuals, and so on for higher-order languages.

[1] After submitting some parts of this thesis to a conference on logic in computer science, one of the peer-reviewers made a claim that was directly in contradiction with a well-known result, also included in this chapter: Gödel's completeness theorem. Thus, without recalling the completeness theorem, this thesis would not be complete.

We consider logic from three perspectives. In the first perspective, the *syntactic perspective*, one looks at the formal structure of the description language: how to form sequences of symbols called formulas, and systems to symbolically transform formulas and to derive formulas from other formulas. In the second perspective, the *semantic perspective*, one is interested in the meaning of formulas by interpretation of the symbols and how symbolic transformations and deductions preserve meaning. In the last perspective, the *perspective of significance*, one is interested in how the syntactic and semantic perspectives are related. In some sense, the syntax and semantics of a language can be chosen freely, and making such choices are called design choices. To motivate some of these design choices, it is the relation between the syntactic and semantic perspectives which bears significance.

The significance of logic is that it is a language that is useful for describing the universe and to reason about such descriptions. We distinguish two levels: the level in which we speak about the universe using logic, and the level in which we speak about the logic itself. The first level is the *object-level*, in which elements of the universe, their transformations and interrelations, are the prime subject. On this level, formulas are used to describe properties about elements of the universe, and we are interested in the logical connections between different properties (of the elements of the universe) that formulas can describe. The significance, or usefulness, of a logic depends on the richness of the object-level, in what properties can and can not be expressed by formulas. The second level is the *meta-level*, in which we study the properties of the syntax and semantics themselves. These properties are called meta-properties, to distinguish them from the properties which are described by formulas on the object-level. In particular, first-order logic is a well-known logic with very rich meta-properties. An example meta-property is the relationship between the so-called syntactic consequence relation between sets of formulas and the semantic consequence relation between sets of formulas.

There are many different logics described in the scientific and philosophic literature, including: classical logic, modal logic, intuitionistic logic. In this chapter we keep ourselves to classical logic: the logic in which the *law of the excluded middle* holds generally. This logic is most familiar to mathematicians and computer scientists.

This chapter proceeds as follows. Section A.1 presents the languages of logic, mostly from a syntactical perspective. Section A.2 introduces structures in which to interpret formulas, so to speak, and thus takes a semantic perspective. This section also introduces the concepts of validity and entailment. In Section A.3, going back to a syntactical perspective, we then introduce a proof system for deducing formulas. In Section A.5, we introduce terms as a shorthand for particular formulas and contexts. Section A.4 demonstrates the significance of logic by recalling important meta-properties that relates the semantic concept of entailment to the syntactic concept of deduction.

The ideas presented in this chapter are largely based on material as presented by any competent book on mathematical logic, such as [79, 31]. An introduction to higher-order logic can be found in [82]. The model theory is based on [47, 119]. The proof theory is based on [212, 26]. See also [17, 199, 83].

A.1 Assertion language

In this section we introduce the languages of logic. In later chapters we also speak of programming languages and program logics, so to avoid ambiguity, we may also speak of *assertion languages* to mean the languages introduced here.

We shall introduce not a single language, but a family of languages that is parameterized by *variables* and a *signature*. A signature consists of non-logical symbols which are taken as primitive, out of which a particular language is constructed. A language consists of *formulas*, or synonymously *assertions*, which are certain (finite) sequences of symbols. The formulas are formed using syntactic rules that depend on the signature one chooses.

Moreover, we restrict ourselves to recursive languages, meaning that for each language there must exist an algorithm that can decide whether a given sequence of symbols is a formula or not. Phrased differently, for each language we could systematically generate all sequences of symbols that are formulas and we could systematically generate all sequences of symbols that are non-formulas. This restriction is useful for computer scientists who want to implement such languages. We shall properly define the set of formulas below, and in such a way that the set is recursive. Before doing so, we introduce the concepts of arity and variables.

An *arity* is a (finite) sequence of arities. An arity is typically associated to a symbol to represent how many arguments, and the arity of each argument, are expected to be following that symbol. We write $(\alpha_1; \ldots; \alpha_n)$ for a sequence of $n > 0$ arities, where each of $\alpha_1; \ldots; \alpha_n$ are again arities. We write $()$ for the empty sequence. We say nullary to mean an arity of length 0, unary to mean an arity of length 1, binary to mean an arity of length 2, and so on. Note that, if we ignore all commas, the set of arities is the full Dyck language consisting of all strings of balanced parentheses.

The order of an arity is its maximal nesting depth, starting from first-order. We have that $()$ is a first-order arity since no arity is nested, $(())$ is a second-order arity since it has a first-order arity nested, $((); ())$ is also a second-order arity since all nested arities are first-order, and $(((); ()); ())$ is a third-order arity since it has a nested second-order arity, and so on. We may also treat a natural number n as an arity, which has precisely n directly nested first-order arities $()$. As a special case, 0 is the arity $()$ since it contains no other nested arities. Arity 0 is first-order, whereas arity n for $n > 0$ is second-order. For example, $1 = (())$ and $2 = ((); ())$. All second-order arities are a natural number arity n for some $n > 0$. We may mix parenthesis and natural numbers, for example $((); ()) = (0; 0)$.

Variables (or, more precisely, variable symbols) can be understood as names or as value placeholder symbols. Given two variables, we are able to recognize whether they are (syntactically) identical or not. Each variable has an arity associated to it. Two variables of different arity are necessarily different.

Definition A.1.1 (Variables). There is a recursive set V of *variables*, such that each variable is associated to an arity, and for each arity there are infinitely many variables associated to that arity.

The order of a variable is the order of the arity of the variable. We write v

to mean that v is a variable with as associated arity . Note that 'v' itself is not a variable, but a meta-variable standing for a variable symbol. We also use w , where 'w' is again a meta-variable standing for variable symbols, and we may use subscripts to obtain any number of such variables. The variables of arity () are called the *first-order* variables. These are also called the *individual variables*, and are typically denoted $x; y; z$ (without superscript). The set of first-order variables is also denoted by V_1. Note that there are no zeroth-order variables, since in a zeroth-order language there is no need for variables. Variables that have a second-order arity are called *second-order* variables. These are also called *predicate variables* (if its arity is 1) or *relation variables* (if its arity is greater than 1) typically denoted $P; Q; R$. For example, Q^2 (or, equally, $Q^{(0;0)}$) is a binary relation variable (with arity 2). The set of second-order variables is denoted by V_2, and so on for higher-order variables. We may leave out arity superscripts if the arity of a variable is clear from context; otherwise, we leave the superscript in place.

Intuitively, variables are called that way since their meaning depends on the context and thus vary. In contrast, one may think of a signature as consisting of constant symbols. These symbols are called constant since their meaning does not depend on the context and remains fixed within one language. Variable symbols and constant symbols are separate, and both together are the *non-logical symbols*.

Definition A.1.2 (Signatures). A *signature* is a recursive set of *constant symbols* such that each constant symbol is associated to a non-zero arity.

We typically denote a signature by . The signatures as defined above are signatures *without parameters*. The order of a constant symbol is the order of its arity. We speak of constants to mean constant symbols of a signature.

The order of a signature is one less than the maximum order of its constants. A first-order signature thus has constants with at most second-order arity. In other words, a first-order signature has no constants with a third or higher-order arity. A second-order signature has constants with at most third-order arity. And so on for higher-order signatures.

There are no first-order constant symbols in any signature, since the only first-order arity is zero. The constant symbols of (second-order) arity 1 are called *predicate symbols*. The constant symbols of (second-order) arity n where $n > 1$ are called n-ary *relation symbols*.

Constant symbols of a signature are typically denoted C (with arity , which may be dropped under the same proviso that holds for variable symbols), but specific signatures may also introduce additional notational conventions. For first-order signatures we may also use $P; Q; R$ as constant symbols (but care must be taken not to confuse constant symbols and variable symbols).

Remark A.1.3. In some texts about logic, signatures also include '[individual] constant symbols' and 'function symbols' that are separate from predicate and relation symbols and used to build complex terms. We do not yet (need to) introduce terms, individual symbols and function symbols at this point, but we shall introduce them later in Section A.5. In here, we consider all symbols of a

signature to be constant symbols, in the sense that their meaning does not depend on the context and remains fixed within one language.

For the remainder of this section, we fix a particular signature . We now continue to define the fundamental concept of logic: formulas. More precisely, we define -formulas, since their formulation depends on the chosen signature . We may speak of formulas instead of -formulas, if is clear from context. In the sequel, and are meta-variables standing for arbitrary formulas.

Definition A.1.4 (Formulas). A *formula* is constructed inductively as follows:

1. $\bot$ is a formula,
2. $(x \doteq y)$ is a formula if x and y are individual variable symbols,
3. $(v_1^1, \ldots, v_n^n)$ is a formula if is a non-logical symbol of arity $(_1, \ldots, _n)$ and $v_1^1, \ldots, v_n^n$ are variable symbols of the corresponding arity,
4. $(\to)$ is a formula if and are formulas,
5. $(\forall v \ )$ is a formula if is a formula and v a variable symbol with arity .

All formulas are constructed by one of these five clauses. Alternatively, we can define formulas by the following abstract grammar:

$$; \quad ::= \bot \mid (x \doteq y) \mid (v_1^1, \ldots, v_n^n) \mid (\to) \mid (\forall v \ )$$

The first three clauses construct primitive formulas, the last two clauses construct complex formulas. Note that in the third clause, variables with non-zero arity can be used in the place of the non-logical symbol. Parentheses, comma, $\bot$, $\doteq$, $\to$ and $\forall$ are logical symbols (and thus not part of the signature nor used as variables). The symbol $\doteq$ is called *identity*. We put the dot on the equality sign to distinguish (object-level) identity from (meta-level) equality. A formula can thus be seen as a finite sequence of logical and non-logical symbols.

We speak of the logical symbols in the following manner. Of the primitive formulas, $\bot$ is called *false*, $(x \doteq y)$ is called *identity* (as in 'x and y are identical'). If in $(v_1^1, \ldots, v_n^n)$ we have that is a variable, then we speak of an *application*. Of the complex formulas, $(\to)$ is called (logical) *implication*, and $(\forall v \ )$ is called *universal quantification*.

Often when proving meta-properties of formulas, we proceed by induction on the *complexity* of formulas. There are different measures of complexity. Typically, we use as complexity measure the *height of a formula*, by viewing the formula as a parse tree and taking the height of that tree. Alternatively, one could take as complexity the *length of a formula*, by viewing a formula as a sequence of symbols and taking the length of that sequence.

We further have the logical symbols $\top$, $\neg$, $\leftrightarrow$, $\wedge$, $\vee$, $\exists$, $\exists!$ to construct formulas that are commonly used in classical logic. These logical symbols are given as abbreviations. Sometimes, we write **true** instead of $\top$, and **false** instead of $\bot$.

Definition A.1.5 (Logical abbreviations).

$$
\begin{aligned}
\top &\text{ abbreviates } (\bot \rightarrow \bot)\\
(\neg\ ) &\text{ abbreviates } (\ \rightarrow \bot)\\
(x \neq y) &\text{ abbreviates } (\neg(x \doteq y))\\
(\ \wedge\ ) &\text{ abbreviates } (\neg(\ \rightarrow \neg(\ )))\\
(\ \vee\ ) &\text{ abbreviates } (\neg(\neg(\ ) \wedge \neg(\ )))\\
(\ \leftrightarrow\ ) &\text{ abbreviates } ((\ \rightarrow\ ) \wedge (\ \rightarrow\ ))\\
(\forall v\ ) &\text{ abbreviates } (\neg(\exists v\ (\neg(\ ))))
\end{aligned}
$$

The logical symbols $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ are called (logical) connectives, since they describe a connection between two formulas. Formulas of the form $(\ \wedge\ )$ are called *conjunctions*, and $(\ \vee\ )$ are called *disjunctions*, and $(\ \leftrightarrow\ )$ are called *bi-implications*. The logical symbols $\forall$ and $\exists$ are called (logical) quantifiers. We have two kinds of quantifiers: $\forall$ quantifies *universally*, and $\exists$ quantifies *existentially*. Note that some of the above syntactic abbreviations are arbitrary: it is also possible to introduce these logical connectives in alternative but equivalent ways.

To reduce the use of parentheses, we employ syntactical conventions for resolving ambiguity in case parentheses are dropped. The precedence of logical symbols, from strongest to weakest binding force, is: $\neg$, $\forall$, $\exists$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. All connectives associate to the right. The process of adding back parentheses is called disambiguation. For example, $\forall x P(x) \wedge Q(x) \wedge P(x)$ disambiguates to $((\forall x P(x)) \wedge (Q(x) \wedge P(x)))$.

To further reduce the use of parentheses, we may employ a single dot after the variable that immediately follows a quantifier, that disambiguates into a pair of parentheses of which the closing parenthesis is placed as far as possible to the right without interfering with the parenthesis already present in the surrounding context. For example, $\forall x.\ P(x) \wedge Q(x)$ disambiguates to $\forall x(P(x) \wedge Q(x))$, and $(\forall x.\ P(x)) \wedge Q(x)$ disambiguates to $(\forall x(P(x))) \wedge Q(x)$.

We also have the syntactic convention that, given directly nested quantifiers of the same kind, such sequence of quantified variables may be listed as a (non-empty) sequence directly after the quantifier symbols. For example $\forall v_1{}^1; v_2{}^2; \ldots; v_n{}^n$ expands to $(\forall v_1{}^1(\forall v_2{}^2(\ldots(\forall v_n{}^n\ )\ldots)))$.

How are variable occurrences bound to quantifiers? The scope of a quantifier in the formula $(\forall v\ )$ is the variable v and the formula $\ $ that follows it, and we say that v is bound to that quantifier. Informally, we can imagine the parse tree of how a formula is constructed. The possible leaves are variables with an associated arity. An occurrence of a symbol in a formula is a path in the parse tree leading to that symbol. A variable occurrence is an occurrence of a variable in a given formula. A variable occurs bound if it occurs as the immediate left child of a quantifier symbol, e.g. the formula $(\forall v\ P(v\ ))$ has $\forall$ as root and v as left child and P as right child with v nested under it. A variable v occurs free if, following the occurrence of v as a path, from the root of the parse tree towards the leaf we do not encounter any quantifier with v as immediate left child. A variable occurrence of v falls under the scope of a quantifier if either it immediately follows a quantifier, or if we follow the path back to the root we encounter a quantifier

binding the same variable. In other words, a variable occurs free if each of that variable occurrence does not fall under the scope of a quantifier.

We now define the set of free variables of a formula, being the set of all variables that occur free in it. Similarly, we define the set of bound variables of a formula.

Definition A.1.6 (Free and bound variables). Given a formula φ. We define both the set of free variables $FV(\varphi)$ and the set of bound variables $BV(\varphi)$ inductively on the structure of φ as follows:

- $FV(\bot) = \emptyset = BV(\bot)$,
- $FV(x \doteq y) = \{x, y\}$ and $BV(x \doteq y) = \emptyset$,
- $FV(C(v_1^1, \ldots, v_n^n)) = \{v_1^1, \ldots, v_n^n\}$ and $BV(C(v_1^1, \ldots, v_n^n)) = \emptyset$,
- $FV(w(v_1^1, \ldots, v_n^n)) = \{w, v_1^1, \ldots, v_n^n\}$ and $BV(w(v_1^1, \ldots, v_n^n)) = \emptyset$ where $\mathbf{i} = (i_1, \ldots, i_n)$,
- $FV(\varphi \to \psi) = FV(\varphi) \cup FV(\psi)$ and $BV(\varphi \to \psi) = BV(\varphi) \cup BV(\psi)$,
- $FV(\forall v\, \varphi) = FV(\varphi) \setminus \{v\}$ and $BV(\forall v\, \varphi) = BV(\varphi) \cup \{v\}$.

The set of variables $V(\varphi)$ is defined $V(\varphi) = FV(\varphi) \cup BV(\varphi)$.

Note that the second and third clause are discriminated by the non-logical symbol being either a constant symbol in our signature or a variable symbol. In the second clause, the constant symbol C must have arity $(i_1, \ldots, i_n)$, and in the third clause, the variable w must have arity $(i_1, \ldots, i_n)$: both constraints follow from the construction of formulas. Other concepts that are inductively defined on the structure of formulas follow a similar pattern.

Note that for every formula φ, we have that $V(\varphi)$ is a finite set. This is easily seen, since every formula is a finite sequence of symbols, thus there can only be finitely many variables that occur in it. If $V(\varphi) = \{v_1^1, \ldots, v_n^n\}$ we also write $\varphi(v_1^1, \ldots, v_n^n)$. For example, $\varphi(x)$ is a formula in which at most x occurs free.

Definition A.1.7 (Sentences). A *sentence* is a formula without free variables.

It is important to note that the context $\forall v \ldots$ is so-called *referentially opaque* [205], meaning that the value each variable refers to in formulas under a quantifier may change. For example, $(x \doteq y)$ and $\forall x(x \doteq y)$ may have a different meaning: in one formula x could refer to a different value than the value x refers to in the other formula. This distinction is especially important when substitutivity comes in play, e.g. when replacing x by z, since referential opacity breaks our naïve principle of 'substitution of equals for equals'.

Convention A.1.8 (Barendregt's variable convention). As a convention, we separate the names used for free variables and bound variables. Formally, a formula φ complies to this convention whenever it is the case that $FV(\varphi) \cap BV(\varphi) = \emptyset$.

It is possible to transform every formula into a formula that complies to the above convention. For that we introduce two concepts: fresh variables and variable renaming. Fresh variables are variables that do not occur in some context. A renaming allows us to transform a given formula into another formula in which variables are uniformly replaced. Now, by choosing appropriate fresh variables for bound variables, we can separate the free and bound variables of a given formula.

We can now motivate our choice in Definition A.1.1 to have for each arity an infinite supply of variables associated to that arity. This allows us to always be able, given a formula φ, to give a *fresh variable* of some arity. A variable v is fresh if it does not occur in a formula, i.e. $v \notin V(\varphi)$. For every variable that occurs bound in a formula, there are infinitely many fresh variables.

Definition A.1.9 (Variable renaming). A *variable renaming* is a mapping ρ of variable symbols, such that variables of a given arity are mapped to variables of the same arity. We define the application $\rho(\varphi)$ of a renaming ρ to a formula φ inductively on the structure of φ as follows:

- $\rho(\bot) = \bot$,
- $\rho(x \doteq y) = (\rho(x) \doteq \rho(y))$,
- $\rho(C(v_1^1, \ldots, v_n^n)) = C(\rho(v_1^1), \ldots, \rho(v_n^n))$ for constant symbol C,
- $\rho(w(v_1^1, \ldots, v_n^n)) = \rho(w)(\rho(v_1^1), \ldots, \rho(v_n^n))$ where $\rho = (\rho_1, \ldots, \rho_n)$,
- $\rho(\varphi \to \psi) = \rho(\varphi) \to \rho(\psi)$,
- $\rho(\forall v\, \varphi) = \forall\, \rho(v)\, \rho(\varphi)$.

It is worth pointing out that a renaming can potentially change multiple variables simultaneously. For example, $\exists y(\forall x P(x) \wedge Q(y))$ can be renamed to $\exists x(\forall y P(y) \wedge Q(x))$ by swapping x and y simultaneously. We may leave the exact mapping used to rename implicit. To explicitly denote a renaming, we use the notation $\begin{smallmatrix} v_1^1 & \cdots & v_n^n \\ w_1^1 & \cdots & w_n^n \end{smallmatrix}$ to denote the renaming which simultaneously renames v_1^1 into w_1^1, $\ldots$, v_n^n into w_n^n (from top to bottom) and leaves all other variables identical. In the previous example, the renaming $\left(\begin{smallmatrix} x & y \\ y & x \end{smallmatrix}\right)$ is used.

When performing renaming, it is sometimes important that the variable that is renamed to is not captured by a quantifier. For example, in the formula $\forall x P(x, y) \wedge Q(y)$ we have that both occurrences of y have the same referents (they are both the same free variable), whereas if we rename y to x to obtain $\forall x P(x, x) \wedge Q(x)$, the one occurrence of variable x falls under the scope of the quantifier $\forall x$ whereas the other occurrence remains free. We say that v^0 *remains free for* w^1 *in* φ if all occurrences of w^1 in φ do not fall under the scope of a quantifier binding v^0.

Note that for formulas that comply to Barendregt's convention, where the bound and free variables are separate, we do not have a problem with renaming of variables if the resulting formula also complies to Barendregt's convention. To ensure this, one could partition the variables into two sets: those potentially used

for free variables, and those potentially used for bound variables. If a renaming retains the status of each variable (i.e. renaming free variables to other potentially free variables, and renaming bound variables to other potentially bound variables), no variable ever gets captured since free variables are never used under quantifiers. When one formula is obtained from another by the application of a renaming of the bound variables, we say the formulas are *alphabetic variants*.

Given non-empty lists of variables $\vec{v} = v_1{}^1;\ldots;v_n{}^n$ and $\vec{w} = w_1{}^1;\ldots;w_n{}^n$ such that the lists match up in length and arity. We write $[\vec{v} := \vec{w}]$ to mean the operation where first is suitable renamed to avoid capture of the variables in $\vec{w}$, and then the renaming of the variables $\vec{v}$ into $\vec{w}$, respectively. This operation is also called (capture-avoiding) substitution.

As a convention, if we are given a formula $(v_1{}^1;\ldots;v_n{}^n)$ with its free variables among the listed variables, then writing $(w_1{}^1;\ldots;w_n{}^n)$ denotes a formula obtained from by substituting the variables $v_1{}^1;\ldots;v_n{}^n$ by respectively $w_1{}^1;\ldots;w_n{}^n$, leaving all other variables identical. For example, given $(x;y)$, then $(y;z)$ is the result obtained from simultaneously substitution x to y and y to z in . We need to take care to avoid variable capturing: if $(x;y)$ is $\forall z(x \doteq y)$, then $(y;z)$ must be $\forall w(y \doteq z)$ where we have renamed the bound variable z appropriately. We may sometimes be unclear, e.g. where (x) can have two meanings: either it declares that has the free variable x, or by (x) we mean with the identity renaming applied (which results in itself).

The order of a formula is determined as the maximum order of the arities of the variable symbols that occur in it, and a formula is called a *zeroth-order formula* if no variable occurs in it. So, if there is at least one variable occurrence and all variables that occur are first-order, then the formula in question is *first-order*. And so on for second and higher-order formulas.

In general, we can classify formulas by their order, and in doing so we also include the formulas of lower order. In a zeroth-order formula, no variables occur. In a first-order formula, all variables that occur are first-order and all constant symbols that occur have a second-order arity. Hence, the set of first-order formulas contains the set of zeroth-order formulas. In a second-order formula, all variables that occur are at most second-order and all constant symbols that occur are at most third-order. Hence, the set of second-order formulas contains the set of first-order formulas. And so on for higher-order formulas.

A *context* is a list of formulas, typically denoted $_1;\ldots;_n$. We also have the empty list of formulas, for which we do not need any special notation. We may treat a single formula as a list of formulas of length 1 consisting of just that formula. If and are lists of formulas, then by $;$ we mean the list formed by adjoining the formulas in the first list to the formulas in the second list. Consequently, by $;$ and $;$ we mean the lists formed by su□xing or prefixing the list consisting of a single formula to the list of formulas , respectively.

On the one hand, contexts are syntactic and finitary objects: every formula is a finitary object, and every list of formulas can be seen as a finite sequence of formulas. On the other hand, we now introduce theories, which are possibly infinitary objects and in some cases are entirely semantic.

Definition A.1.10 (Theories). A *theory* is a set of sentences.

A *first-order theory* is a set of first-order sentences, a *second-order* theory is a set of second-order sentences, and so on for higher orders. A *finite* theory is a theory where its set of sentences is finite, and an *enumerable* theory is a theory with a countable set of sentences. In particular, we shall look at three classes of theories: satisfiable theories, consistent theories, and complete theories.

In Section A.2, we introduce the semantics of formulas, and in particular we define when a theory is *satisfiable* (also called *semantically consistent*). A theory is satisfiable (viz. semantically consistent) if there exists a structure for which all sentences in the theory are satisfied (one may think that the theory 'consists of' at least one such structure). In a satisfiable theory , it must be the case that not all sentences are in . However, the converse, if not all sentences are in , does not necessarily imply that is satisfiable.

Then, in Section A.3, we introduce syntactic proof systems and we define when a theory is *deductively closed*. A theory is *deductively consistent* (or consistent in short) if it is not possible to deduce false from it. Equivalently, the theory is consistent if there exists some sentence that is not contained in the deductive closure of .

A theory is *complete* if for every sentence , either *2* or (*:*) *2* . It is called complete in the sense that it is no longer possible to add any more sentences without the theory turning inconsistent: if one adds a sentence to a complete theory that was not yet contained in it, closing the resulting theory deductively would result in an inconsistent theory.

The significance of these definitions is described in Section A.4, where we give the main result that the syntactic proof systems are, in a precise sense, adequate for our semantics: the notion of syntactic consistency coincides exactly with the notion of semantic consistency. Sometimes this is also called soundness and completeness (not to be confused with complete theories).

A.2 Basic model theory

In this section we introduce models of the languages we introduced above, in the style of Tarksi. Models are also called *structures*. Structures are used to give meaning to formulas of a language. We thus take a semantic perspective in this section. The meaning of formulas builds on two concepts: interpretations and valuations. Each structure fixes a *domain* of discourse, also called the universe. The domain restricts the values that are possible. Each structure also fixes the interpretation of constant symbols, assigning to each constant symbol a value. Structures further induce valuations, which assign to each variable symbol a value. Given both an interpretation (for the constant symbols) and valuation (for the variable symbols), we are able to give meaning to formulas relative to an ambient structure.

Both interpretations and valuations employ the concept of value, albeit assigning them to constants or variables, respectively. Values are structured by an arity and

range over the domain. The values of first-order symbols range directly over the elements of our domain. There are no first-order constant symbols, only first-order variables. The values of second-order symbols range over particular sets comprising elements of our domain, depending on their arity. Moreover, the role of quantifiers in our language is to modify valuations, and thereby varying the value of variables depending on their context.

Definition A.2.1 (Values). Given a domain D. Let $D[\ ]$ denote the set of values of D of arity . A *value* of D of arity is constructed inductively:

1. An element $d \in D$ is a value of D of arity 0.
2. A set $S \subseteq D[\ _1] \times \cdots \times D[\ _n]$ is a value of D of arity $(\ _1, \ldots, \ _n)$.

All values are constructed by one of these two clauses.

It is easy to see that $D[0] = D$ and $D[(\ _1, \ldots, \ _n)] = P(D[\ _1] \times \cdots \times D[\ _n])$ where P denotes the powerset operator on sets. Hence we have $D[1] = P(D)$, $D[2] = P(D \times D)$, and $D[n] = P(D^n)$ for arities $n > 1$.

Example A.2.2. Let N be our domain. We then have the following values of N:

- *First-order values:*

  $0, 1, 2, \ldots$ are in N[0] = N. Every natural number is a first-order value.

- *Second-order values:*

  $\{1, 3, 5, \ldots\}$ is in N[1] = P(N), $\{(0, 1), (1, 2), (2, 3), \ldots\}$ is in N[2] = P(N × N), et cetera. Every possible set of natural numbers is a second-order value in N[1], and every possible binary relation on natural numbers is a second-order value in N[2], and so on.

- *Third-order values:*

  $\{(f, S) \mid f : \mathrm{N} \rightharpoonup \mathrm{N} \text{ and } S = dom(f)\}$ in N[(2, 1)] = $P(P(\mathrm{N} \times \mathrm{N}) \times P(\mathrm{N}))$. This value describes a relational between a relation f and a set S, where $f : \mathrm{N} \rightharpoonup \mathrm{N}$ means that f is a partial function on N, and $dom(f)$ is the set of elements on which f is defined. *End of Example.*

We have introduced values in this way to be able to attend to the higher-order aspect of languages, namely how to give a value to second and higher-order variables. We first introduce the so-called *standard* model theory of logic. In the standard model, variables range over all values. It is also possible to consider a different semantics of logic, resulting in the so-called *general* model theory of logic, where variables range over a restricted set of values.

As can be seen in the example above, the first-order values are elements of the domain. Sometimes we use *elementary* (or the adjective *elementarily*) as a synonym of first-order, to remind the reader of this fact.

For the remainder of this section, we again fix a particular signature .

Definition A.2.3 (Structures). A *structure* A is a pair of a *domain* A (a set of elements) and an *interpretation* I. An interpretation I assigns every constant symbol of arity to a value of A of arity .

Given that C is a constant symbol, we write C^I to mean the value given to C by the interpretation I (and it should be clear from context to which structure an interpretation is part of). Sometimes we also speak of the *extension* of a constant symbol, to mean its value given by an interpretation. Note that constant symbols are never of arity zero, hence the extension of a constant symbol is always a set. Further, as a convention, when we describe some structure using Gothic letters, e.g. A; B, then we simply refer to the underlying domain using an uppercase roman typeface, e.g. $A; B$, respectively.

We introduce two basic concepts involving structures: isomorphisms and substructures. To do so, it is necessary to transport values of one domain to another domain. Let f be a bijection between the domains A and B. We can use f directly as a bijection between the values in $A[0]$ and $B[0]$. We can construct a lifting of the bijection f to higher-order values inductively. Let $f_1 : A[\ _1] \ ! \ B[\ _1]$, $:::; f_n : A[\ _n] \ ! \ B[\ _n]$ be bijections on values of arities $_1;:::; \ _n$. Then there exists a bijection $f^0 : A[\ _1] \ ::: \ A[\ _n] \ ! \ B[\ _1] \ ::: \ B[\ _n]$ by mapping each component of the Cartesian product using $f_1;:::; f_n$, respectively. Consequently, there exists a bijection $f^{00} : P(A[\ _1] \ ::: \ A[\ _n]) \ ! \ P(B[\ _1] \ ::: \ B[\ _n])$ between the values in $A[(\ _1;:::; \ _n)]$ and $B[(\ _1;:::; \ _n)]$.

Given two structures A and B. The structures are *isomorphic*, written A = B, if and only if there is a bijection f between the domains A and B such that $C^J = f^{00}(C^I)$ for every constant symbol C, where f^{00} is f lifted to a bijection between the values of A and B of the arity associated to C, I is the interpretation of A, and J is the interpretation of B.

The cardinality of the domains of isomorphic structures are equal. In other words, structures with a finite domain are isomorphic only to structures with also a finite domain, and similar for structures with countable or uncountable domains.

Given two structures A and B. Structure A is a *substructure* of B, written A B, if and only if $A \ B$ and $C^I = C^J \setminus A(\ )$ for every constant symbol C, where I is the interpretation of A, and J is the interpretation of B.

Definition A.2.4 (Valuations). Given a structure A. A *valuation* of A assigns every variable symbol of arity to a value of A of arity .

For a variable v , by $(v\ )$ we mean the value given to v by the valuation . Note that if the domain of A is empty, there are no first-order values and thus there cannot be a valuation, since we have (infinitely many) first-order variables that need to be assigned a value. Hence, in the context of a valuation, we may assume the domain of A to be non-empty.

Given a valuation and a variable v , if a is a value of A of arity , then $[v := a]$ is the *updated* valuation obtained so to satisfy the following two equations:

$$[v := a](v) = a$$

$$[v := a](w) = \ (w) \text{ if } v \text{ and } w \text{ are di erent}$$

where w is a meta-variable standing for a variable symbol. Note that if v and w have a different arity, they are necessarily different.

Definition A.2.5 (Satisfaction relation). Given a structure $\mathcal{A}$ and a valuation α of $\mathcal{A}$, and a formula φ. The satisfaction relation $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$ is defined inductively on the structure of φ:

1. $\mathcal{A}, \alpha \not\models^{\mathbf{CL}} \bot$ never holds,
2. $\mathcal{A}, \alpha \models^{\mathbf{CL}} (x \doteq y)$ holds iff $\alpha(x) = \alpha(y)$,
3. $\mathcal{A}, \alpha \models^{\mathbf{CL}} C(v_1^1, \ldots, v_n^n)$ holds iff $(\alpha(v_1^1), \ldots, \alpha(v_n^n)) \in P^I$,
4. $\mathcal{A}, \alpha \models^{\mathbf{CL}} w(v_1^1, \ldots, v_n^n)$ holds iff $(\alpha(v_1^1), \ldots, \alpha(v_n^n)) \in \beta(w)$ where $\beta = (\beta_1, \ldots, \beta_n)$,
5. $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi \to \psi$ holds iff $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$ implies $\mathcal{A}, \alpha \models^{\mathbf{CL}} \psi$,
6. $\mathcal{A}, \alpha \models^{\mathbf{CL}} \forall v \varphi$ holds iff $\mathcal{A}, [v := a] \models^{\mathbf{CL}} \varphi$ holds for every $a \in A[\,]$.

The superscript CL stands for Classical Logic. Instead of writing $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$, we may also speak of 'A and α (classically) satisfy φ', or 'φ is (classically) satisfied by A and α', or 'φ is (classically) satisfiable' if there is some A and α. We may leave out the superscript CL or the word 'classically', if no confusion can arise about the logic we use. In the remainder of this section we drop CL and 'classically'.

Note that the definition of the satisfaction relation above breaks down for structures with an empty domain, since there does not exist a valuation if the domain is empty. We still, however, have that some formulas could be considered satisfied in such empty structures (e.g. $\varphi \to \bot \to \varphi$ and $\forall x \varphi$) whereas other formulas should not (e.g. $\forall P(P(x))$). This technical inconvenience can be resolved by introducing a pseudo valuation for use in empty structures only, which does not assign first-order variables a value. However, we shall leave out the tedious details, and keep ourselves to non-empty structures.

Also the abbreviations can be given a semantics, which follow easily from the definition above:

- $\mathcal{A}, \alpha \models^{\mathbf{CL}} \top$ always holds,
- $\mathcal{A}, \alpha \models^{\mathbf{CL}} \neg\varphi$ holds iff $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$ does not hold,
- $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi \wedge \psi$ holds iff $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$ and $\mathcal{A}, \alpha \models^{\mathbf{CL}} \psi$ holds,
- $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi \vee \psi$ holds iff $\mathcal{A}, \alpha \models^{\mathbf{CL}} \varphi$ or $\mathcal{A}, \alpha \models^{\mathbf{CL}} \psi$ holds,
- $\mathcal{A}, \alpha \models^{\mathbf{CL}} \exists v \varphi$ holds iff $\mathcal{A}, [v := a] \models^{\mathbf{CL}} \varphi$ holds for some $a \in A[\,]$.

The satisfaction relation depends on only finitely many variables being assigned a value by a valuation. This is formally captured by the following proposition. Given a set of variables X, by $\alpha[X] = \alpha'[X]$ we mean that the valuations α and α' *coincide* on X, that is, α and α' assign the same values to variables in X.

Proposition A.2.6 (Coincidence condition). *Given that* $[FV(\ )] = {}^{0}[FV(\ )]$, *it follows that* $\mathfrak{A}; \ \models \ $ *if and only if* $\mathfrak{A}; {}^{0} \models \ $.

Similarly, the choice of bound variables bears no significance on the meaning of a formula. This is formally captured by the following proposition.

Proposition A.2.7 (Invariance under renaming). *Given a formula* , *and a renaming* *such that all free variables of* *stay the same, i.e.* $(v) = v$ *for all* $v \in FV(\ )$. *It follows that* $\mathfrak{A}; \ \models \ $ *if and only if* $\mathfrak{A}; \ \models \ (\ )$.

The proposition above is significant, as it provides a semantic justification for Barendregt's variable convention (see Convention A.1.8). It is always possible to rename the bound variables of a formula, so that the bound variables and free variables are separated, without changing the meaning of a formula.

Lemma A.2.8 (Substitution lemma). *Given a formula* *and variables* $v \ ; w$, *then* $\mathfrak{A}; \ \models \ [v \ := w \ ]$ *if and only if* $\mathfrak{A}; \ [v \ := \ (w \ )] \models \ $.

Sometimes, it is more convenient to work with the set of valuations by which a formula is satisfied given a particular structure.

Definition A.2.9 (Denotation). The *denotation* of a formula $\mathfrak{A}\llbracket \ \rrbracket^{\mathbf{CL}}$ is defined:

$$\mathfrak{A}\llbracket \ \rrbracket^{\mathbf{CL}} = \{ \ | \ \mathfrak{A}; \ \models^{\mathbf{CL}} \ \}:$$

Similar as before, we may drop CL if clear from context. We write $\equiv_{\mathfrak{A}}$ for $\mathfrak{A}\llbracket \ \rrbracket = \mathfrak{A}\llbracket \ \rrbracket$, and say that and are equivalent.

We write $\mathfrak{A} \models$ to mean $\mathfrak{A}; \ \models$ for all valuations , and we say that is *true* in $\mathfrak{A}$. If is a sentence that is satisfied by $\mathfrak{A}$ and some valuation, using the coincidence condition, we can obtain that it is also satisfied by the same structure but with any other valuation: the valuation has no influence on whether a sentence is satisfied by the structure. So if is a sentence, it is true in $\mathfrak{A}$ if and only if it is satisfied by $\mathfrak{A}$, that is, $\mathfrak{A} \models$ if and only if $\mathfrak{A}; \ \models$ for some valuation .

Given a sentence , we write $\models$ to mean that $\mathfrak{A} \models$ for all structures $\mathfrak{A}$, and we then say that is *valid*.

Given a theory, i.e. a set of sentences , we write $\mathfrak{A} \models$ to mean that all sentences in are true in $\mathfrak{A}$, that is, $\mathfrak{A} \models$ for all $\in$. We may then also speak of ' is satisfied by $\mathfrak{A}$'. A theory is *satisfiable* if there exists a structure $\mathfrak{A}$ such that $\mathfrak{A} \models$. A theory is *finitely satisfiable* if every finite subset of is satisfiable.

Theorem A.2.10 (Compactness). *Given a first-order theory* . *is satisfiable if and only if* *is finitely satisfiable.*

Proof. Follows from Loś's theorem and an ultraproduct construction, see [87, Theorem 2.10]. □

We write $\models$ to mean $\mathfrak{A} \models$ for all structures $\mathfrak{A}$ such that $\mathfrak{A} \models$, and say that is a *semantic consequence* of . As an additional case of semantic consequence, we consider a context, i.e. a list of formulas , and a formula . We

write $\models$ to mean A; $\models$ for all structures A and valuations such that A; $\models$ for every formula 2 . Note that for contexts, we deal with formulas that may contain free variables. As such, there is one valuation that is used in both checking the satisfaction of all formulas of the context, and in satisfaction of the given formula . If we have only sentences in and is also a sentence, then both readings of $\models$ coincide.

By Th^{CL}(A) we mean the set of all sentences such that A $\models^{CL}$, and we speak of the *higher-order theory of* A. (Again, we may drop the superscript CL if clear from context.) If we restrict Th(A) to the first-order formulas, denoted Th_1(A), we speak of the *first-order theory of* A. If we restrict Th(A) to the second-order formulas, denoted Th_2(A), we speak of the *second-order theory of* A. And so on for higher orders. By the way we classify formulas, higher-order theories always include lower-order theories, i.e. Th_1(A) Th_2(A) $:::$

In general, we have for every structure A and formula that either A $\models$ or A $\models$: . Thus, the (first-order, second-order, $:::$, higher-order) theory of a structure is necessarily complete.

Given two structures A and B. The structures are *elementarily equivalent*, written A $\equiv_1^{CL}$ B, if and only if for every first-order sentence we have A $\models^{CL}$ if and only if B $\models^{CL}$. (We may drop the superscript CL under the same proviso.) In other words, two elementarily equivalent structures satisfy exactly the same first-order sentences, i.e. Th_1(A) = Th_1(B).

Given a set of sentences , by $\mathrm{Mod}^{CL}(\ )$ we mean the class of all structures A such that A $\models^{CL}$. (Same treatment of the superscript CL.) Gaining insight in the classification of structures is the main goal of model theory. A first result of model theory is given below.

Proposition A.2.11. *A first-order theory is complete if and only if all structures* A; B *2* Mod( ) *are elementarily equivalent, i.e.* A $\equiv_1$ B*.*

The class of *finite structures* consists of structures A = (A; I) where the domain A is finite. A natural question to ask is: is it possible to give a sentence that characterizes finite structures? To be able to answer that question, it is worthwhile to give an informal proof of the following proposition.

Proposition A.2.12. *A set D is finite if and only if every injective total function $f : D \to D$ is a surjection.*

Proof. Suppose D is finite, and let f be an injective total function. Suppose, towards contradiction, that f is not a surjection. Then there is an unreachable element, i.e. some x for which there is no input y such that the output $f(y) = x$. Since f is a total function, the function f must be defined for every input: there is exactly one outgoing pointer for every element in D. Since D is finite, there are n points. So there are n pointers, but at most n 1 points are reached. Then, according to the pigeonhole principle, there must be one point which can be reached through f from two different inputs. This is in contradiction with the fact that f is injective.

Suppose every injective total function $f : D \to D$ is a surjection. Suppose, towards contradiction, that D is infinite. Then D must be non-empty, and let d be

some element of D (it does not matter which one you choose). Now consider the set $D \setminus \{d\}$, which is still infinite. We can place every element of $D \setminus \{d\}$ next to precisely one element of D, and thus there is a bijection between D and $D \setminus \{d\}$. However, this shows that we have a total function that is an injection from D to D, but not a surjection, since d is never reached. Contradiction! □

Let R be a 2-ary variable, and x, y, z distinct individual variables. We have the following abbreviations:

- $\mathit{fun}(R)$ abbreviates $\forall x, y, z.\ R(x, y) \wedge R(x, z) \Rightarrow y \doteq z$,
- $\mathit{inj}(R)$ abbreviates $\forall x, y, z.\ R(x, z) \wedge R(y, z) \Rightarrow x \doteq y$,
- $\mathit{tot}(R)$ abbreviates $\forall x \exists y R(x, y)$,
- $\mathit{surj}(R)$ abbreviates $\forall y \exists x R(x, y)$.

Our characterizing sentence is now the following.

Proposition A.2.13 (Characterization of finite structures). *A is a finite structure if and only if*

$$\mathfrak{A} \models \forall R.\ \mathit{fun}(R) \wedge \mathit{tot}(R) \wedge \mathit{inj}(R) \Rightarrow \mathit{surj}(R).$$

Proposition A.2.14. *Given structures* $\mathfrak{A}$ *and* $\mathfrak{B}$*. Then* $\mathfrak{A} \equiv \mathfrak{B}$ *implies* $\mathfrak{A} \equiv_1 \mathfrak{B}$*. The converse also holds for finite structures.*

Thus, first-order logic is sufficiently powerful for classifying the finite structures (up to isomorphism).

An important class of structures is that of *countable structures*, which are structures with a countable domain. We say that a set X is *countable* if there exists an *enumeration* function $f : \mathbb{N} \to X \cup \{?\}$ from the natural numbers to the set $(X \cup \{?\})$ in which $?$ is a dummy element not in X, such that for every element $x \in X$ there exists a natural number n such that $f(n) = x$. We make use of a dummy element to ensure that finite sets are also considered countable.

In a countable structure, the first-order values are countable although the second and higher-order values are not countable. Although it is the case that for given countable sets their finitary Cartesian product is again countable, this fails for power sets. For a given countable set its power set is not countable (which follows from a diagonalization argument).

Lemma A.2.15. *Given a finite signature and countable structures* $\mathfrak{A}$ *and* $\mathfrak{B}$*. Then* $\mathfrak{A} \equiv_2 \mathfrak{B}$ *implies* $\mathfrak{A} = \mathfrak{B}$.

Proof. The proof requires the axiom of constructibility, see also [5]. □

Thus, second-order logic is sufficiently powerful for classifying the countable structures (up to isomorphism).

If we restrict ourselves to first-order signatures, we also have an important class of structures called data structures. Essentially, a *data structure* is a countable structure with a computable interpretation. Formally, this amounts to the following conditions, where X is the domain:

- there exists an enumeration function $f : \mathbb{N} \to X \cup \{?\}$ such that for each $x \in X$ there exists a *unique* natural number n such that $f(n) = x$,
- the set $\{n \mid f(n) \neq ?\}$ is computable (i.e. it is decidable whether a natural number represents an element of the domain of the data structure or not),
- the interpretation is computable (i.e. the extension of every constant symbol is a decidable set).

The unique natural number corresponding to each element of the domain is called its *encoding*. One uses the encoding of an element in showing that the interpretation is computable, since in data structures one can easily go back and forth between the elements of the domain and their encoding as a natural number.

Proposition A.2.16. *It is decidable whether a quantifier-free formula is satisfied in a given data structure and valuation.*

Note that data structures induce a natural order relation on its element, by their enumeration order. With some additional effort, one could also define bounded formulas (in which existential and universal quantification is always bounded) and extend the above decidability property to bounded formulas as well. Further, given a bounded formula φ, it is semi-decidable whether the formula $\exists x \varphi$ is satisfied in a given data structure and valuation.

A.3 Basic proof theory

We now investigate syntactical systems for deduction, also called *proof systems*. First, we introduce proof systems in abstracto, in the sense that we abstract from the (syntactic) objects which are involved in deductions. Many interesting properties of proof systems can already be stated in the abstract, regardless of the syntactic objects [26]. Then, we investigate a particular proof system for classical logic, by instantiating objects by the formulas of our assertion language. Later in this thesis, we also introduce proof systems for reasoning about separation logic and program correctness, thus further motivating the approach of giving proof systems in the abstract first.

Definition A.3.1 (Proof system). A *proof system* $\mathsf{D} = (O, \vdash)$ consists of a class of objects O and a deduction relation $\vdash$ on lists of objects and objects, that satisfy the following conditions:

(Rg) $a_1, \ldots, a_n \vdash a_i$ for any $1 \leq i \leq n$,

(Tg) $\left.\begin{array}{c} a_1, \ldots, a_n \vdash b_1 \\ \vdots \\ a_1, \ldots, a_n \vdash b_m \end{array}\right\}$ and $b_1, \ldots, b_m \vdash c$ implies $a_1, \ldots, a_n \vdash c$.

Whenever a list of objects $a_1, \ldots, a_n$ and an object b are related by the deduction relation $\vdash$, we say that 'b follows from $a_1, \ldots, a_n$' or '$a_1, \ldots, a_n$ leads to b'. If that is

the case, the objects $a_1;:::;a_n$ are called the *premises* and b is called the *conclusion*. The witness that the deduction relation between premises and a conclusion holds is called a *deduction*. It should be clear from context to which proof system the deduction relation = belongs. We fix some proof system D = $(O;=)$ until Definition A.3.4.

The condition (Rg) is called *generalized reflexivity*, and the condition (Tg) is called *generalized transitivity*. Note that the we require an expressive meta language due to the many ellipsis: the condition (Tg) reads as 'if b_i follows from $a_1;:::;a_n$ for all $1 \quad i \quad m$ and c follows from $b_1;:::;b_m$, then c also follows from $a_1;:::;a_n$'. Both conditions imply the non-generalized facts:

(R) $a = a$,

(T) $a = b$ and $b = c$ implies $a = c$,

which establishes that the deduction relation is reflexive and transitive.

In some logic texts, deductions are depicted in a di□erent way. Instead of writing $a_1;:::;a_n = b_1$ up to $a_1;:::;a_n = b_m$, deductions are rendered as

$$\begin{array}{ccc} a_1 ::: a_n & & a_1 ::: a_n \\ D_1 & & D_m \\ b_1 & ::: & b_m \end{array}$$

Deductions are tree shaped, where a conclusion is at the root of the tree and the premises are its leaves. Note that in the depiction above the premises and the conclusions are all part of the deductions. So, above, $a_1;:::;a_n$ and b_1 are all part of D_1. With this perspective in mind, we may also call deductions *proof trees*. Using this notation we can depict generalized transitivity as follows. Assuming the deduction given above, and the deduction given below

$$\begin{array}{c} b_1 ::: b_m \\ D^0 \\ c \end{array}$$

we can imagine pasting the proof trees D_1 up to D_n in the place of the leaves $b_1;:::;b_m$ of the proof tree of D^0, where the conclusions of the former proof trees overlap with the premises of the latter, to finally obtain the deduction:

$$\begin{array}{ccc} a_1 ::: a_n & & a_1 ::: a_n \\ D_1 & & D_m \\ b_1 & ::: & b_m \\ & D^0 & \\ & c & \end{array}$$

which is to say, there exists a deduction D:

$$\begin{array}{c} a_1 ::: a_n \\ D \\ c \end{array}$$

A proof system is called *finitary* if the class of objects is a decidable set and if there are finitary means to establish that the deduction relation holds. We do not require that the class of objects is a finite set, but we do require a recursive deduction relation. This can be imagined by having finite *certificates* that serve as witnesses for establishing that the deduction relation holds. Finitary proof systems play an essential rôle in computer science, since the certificates that establish deductions of a finitary proof system can be checked by a computer. This allows for the development of tools for constructing and checking deductions. In the remainder, we shall pay attention mostly to finitary proof systems.

Lemma A.3.2 (Exchange, weakening, contraction).

(E) $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n = b$ *implies* $a_1, \ldots, a_{i+1}, a_i, \ldots, a_n = b$,

(W) $a_1, \ldots, a_n = c$ *implies* $a_1, \ldots, a_n, b_1, \ldots, b_m = c$,

(C) $a_1, \ldots, a_n, b, b, c_1, \ldots, c_m = d$ *implies* $a_1, \ldots, a_n, b, c_1, \ldots, c_m = d$.

The proof follows easily from generalized reflexivity and transitivity. The conditional (E) is called the property of *exchange*. It describes, intuitively, that if there is a deduction from a list of premises, then we must also have a deduction in which the premises are permuted. The conditional (W) is called the property of *weakening*. Intuitively it says, if there is a deduction from a list of premises, we must also have a deduction in which additional (but unused) premises are present. Finally, the conditional (C) is called the property of *contraction*. Intuitively, the multiplicity of premises do not matter. Thus, it is possible to see the list of premises of any deduction as a finite set of objects (where duplicates and order do not matter), which can always be extended with additional premises.

We now introduce concepts that are derived from the deduction relation. If we have two objects and one object follows from the other and vice versa, then we say that the two objects are mutually deducible. For that purpose we introduce the following abbreviation,

$$a = b \text{ abbreviates } a = b \text{ and } b = a$$

Mutual deducibility has two important properties, namely that we can replace any conclusion or premise with an object which is mutually deducible. The proof of the following lemma is again simple.

Lemma A.3.3 (Substitutivity). $a = b$ *and* $c_1, \ldots, c_n = a$ *implies* $c_1, \ldots, c_n = b$, $a = b$ *and* $c_1, \ldots, c_n, a, d_1, \ldots, d_m = e$ *implies* $c_1, \ldots, c_n, b, d_1, \ldots, d_m = e$.

An important derived concept is that of *relative demonstrability*. When describing proof systems, one is foremost interested in this concept. We introduce the following notation: $\vdash^{\mathsf{D}}$. The symbol $\vdash$ is called a *turnstile*. The superscript annotates which proof system is used, and may be dropped if the proof system is clear from context. We first define the concept of relative demonstrability, which can then be refined into four concepts familiar to most users of logic: complementarity, demonstrability, contradictoriness, and refutability.

Definition A.3.4 (Relative demonstrability). Let D = $(O; =)$ be a proof system. We define *relative demonstrability* as a relation $\vdash^D$ on lists of objects, as follows: $a_1; \ldots; a_n \vdash^D b_1; \ldots; b_m$ if and only if

$$b_1; d_1; \ldots; d_k = c \text{ and } \ldots \text{ and } b_m; d_1; \ldots; d_k = c \text{ implies } a_1; \ldots; a_n; d_1; \ldots; d_k = c$$

for all $d_1; \ldots; d_k$ and for all c.

Given that a list of objects $a_1; \ldots; a_n$ and a list of objects $b_1; \ldots; b_m$ are related by the relative demonstrability relation, i.e. $a_1; \ldots; a_n \vdash b_1; \ldots; b_m$, then we call the objects $a_1; \ldots; a_n$ the *antecedents* and the objects $b_1; \ldots; b_m$ the *succedents*. We shall talk about the exceptional cases, when either of the two lists is empty, as follows: if $\vdash b_1; \ldots; b_m$ then we call $b_1; \ldots; b_m$ *complementary*, if $\vdash b$ then we call b *provable*, if $a_1; \ldots; a_n \vdash$ then we call $a_1; \ldots; a_n$ *contradictory*, and if $a \vdash$ then we call a *refutable*.

We have the following important property of relative demonstrability:

$$a_1; \ldots; a_n \vdash b \text{ if and only if } a_1; \ldots; a_n = b.$$

It captures that relative demonstrability and the deduction relation coincide in case there is a single succedent. Furthermore, we have

$$a_1; \ldots; a_n = b_1 \text{ or } \ldots \text{ or } a_1; \ldots; a_n = b_m \text{ implies } a_1; \ldots; a_n \vdash b_1; \ldots; b_m.$$

Note that the converse does not hold in general. While the deduction relation must be recursive, relative demonstrability is not necessarily recursive (this can be seen from its definition, where we have an unbounded universal quantification over sequences of objects).

Example A.3.5. Construct a proof system: take O to be two distinct objects, say a and b, and take the smallest deduction relation that satisfies generalized reflexivity (and thus generalized transitivity). Then we do have that $= a$ or $= b$ implies $\vdash a; b$. But the converse fails. Clearly $\vdash a; b$ holds (consider on the meta-level that only instances of generalized reflexivity satisfy the premise when both a and b are in $d_1; \ldots; d_n$). But we have neither $= a$ nor $= b$.

Similar to Lemma A.3.2 we also have properties of exchange, weakening, and contraction, but on either sides of the turnstile.

Lemma A.3.6 (Left/right exchange, weakening, contraction).

(LE) $a_1; \ldots; a_i; a_{i+1}; \ldots; a_n \vdash b_1; \ldots; b_m$ *implies* $a_1; \ldots; a_{i+1}; a_i; \ldots; a_n \vdash b_1; \ldots; b_m$,

(RE) $a_1; \ldots; a_n \vdash b_1; \ldots; b_i; b_{i+1}; \ldots; b_m$ *implies* $a_1; \ldots; a_n \vdash b_1; \ldots; b_{i+1}; b_i; \ldots; b_m$,

(LW) $a_1; \ldots; a_n \vdash c_1; \ldots; c_k$ *implies* $a_1; \ldots; a_n; b_1; \ldots; b_m \vdash c_1; \ldots; c_k$,

(RW) $a_1; \ldots; a_n \vdash c_1; \ldots; c_k$ *implies* $a_1; \ldots; a_n \vdash c_1; \ldots; c_k; b_1; \ldots; b_m$,

(LC) $a_1,\ldots,a_n,b,b,c_1,\ldots,c_m \vdash d_1,\ldots,d_k$ *implies* $a_1,\ldots,a_n,b,c_1,\ldots,c_m \vdash d_1,\ldots,d_k$,

(RC) $a_1,\ldots,a_n \vdash c_1,\ldots,c_m,b,b,d_1,\ldots,d_k$ *implies* $a_1,\ldots,a_n \vdash c_1,\ldots,c_m,b,d_1,\ldots,d_k$.

Note that an easy corollary that follows from right exchange and right weakening is that we also have generalized reflexivity for $\vdash$, as follows:

$$a_1,\ldots,a_n \vdash b_1,\ldots,b_m \text{ if } a_i = b_j \text{ for some } i,j.$$

Another important consequence of the definition of relative demonstrability is:

Lemma A.3.7 (Cut). *If* $a_1,\ldots,a_n \vdash b_1,\ldots,b_m,e$ *and* $e,a_1,\ldots,a_n \vdash b_1,\ldots,b_m$ *then* $a_1,\ldots,a_n \vdash b_1,\ldots,b_m$.

Proof. Fix arbitrary $d_1,\ldots,d_k$ and c. It is sufficient, assuming $b_i,d_1,\ldots,d_k = c$ for $1 \quad i \quad m$, to show $a_1,\ldots,a_n,d_1,\ldots,d_k = c$. Applying (LW) and (LE) on our assumptions, we obtain $b_i,a_1,\ldots,a_n,d_1,\ldots,d_k = c$ for $1 \quad i \quad m$. From the second premise we know that $e,a_1,\ldots,a_n,d_1,\ldots,d_k = c$. Applying all these facts to the first premise we obtain $a_1,\ldots,a_n,a_1,\ldots,a_n,d_1,\ldots,d_k = c$. After applying (LE) and (LC) we reach our goal. □

Now that we have explored proof systems in the abstract, we can construct particular and concrete proof systems. Typically, one constructs a finitary proof system by following three steps:

1. One first specifies what is the class of objects. The class of objects typically has a certain structure, e.g. there are operations defined on objects such that the class of objects is closed under application of the operations.

2. One defines the deduction relation: this can be done by introducing *axioms* and *proof rules*, often in the form of axiom and proof rule *schemata*.

3. One checks that the resulting proof system is finitary (i.e. the class of objects and the deduction relation are recursive), by showing there is an algorithm that can decide the deduction relation.

The third step is easy if one takes a recursive set of objects and defines the deduction relation inductively. However, an alternative to the second step above is by imposing constraints on the deduction relation, e.g. in terms of relative demonstrability. Then the third step is non-trivial.

We consider two classes of proof rules: simple and complex. Axioms and simple proof rules are often depicted as follows:

$$\frac{}{b} \qquad \frac{a_1 \quad \ldots \quad a_n}{b}$$

where the axiom on the left denotes $\vdash b$ (the conclusion b follows from no premises) and the proof rule on the right denotes $a_1,\ldots,a_n \vdash b$ (the conclusion follows from

the premises $a_1;:::;a_n$). One may consider an axiom to be a proof rule without premises. Contrastingly, one may depict complex proof rules as follows:

$$\frac{\begin{array}{ccc} D_1 & & D_n \\ a_1 & \cdots & a_n \end{array}}{b}$$

which expresses how to construct a deduction with conclusion b, given deductions D_1 up to D_n with conclusions a_1 up to a_n, respectively. In such constructions, one also has to describe how to treat the premises of the deductions on top of the rule. For example, the constructed deduction may have less premises than the premises of D_1 up to D_n combined. In such cases, the object used as premise in the deduction on top is called an *assumption*, which is *closed* by the complex proof rule. Such situations are also depicted as follows:

$$\frac{a_1 \quad \cdots \quad \boxed{\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}} \; a_n}{b}$$

to indicate that the premises of the deduction on top of a_1 are also taken to be premises of the resulting deduction, but that the objects $c_1;:::;c_k$ in the deduction on top of a_n are assumptions, and hence not premises of the resulting deduction.

A simple proof rule can be considered to be a complex proof rule where the premises are shared among all deductions and no assumptions are closed: this is what generalized transitivity ensures. Axiom and (simple or complex) proof rule schemata employ meta-linguistic constructs to describe constraints that must hold for its instances.

Example A.3.8. Consider the set N of objects, where 0 is zero and $s : \mathbb{N} \to \mathbb{N}$ the successor function. We have the following axiom and simple proof rule schema:

$$\frac{}{0} \qquad \frac{x}{s(x)}$$

The proof rule is a schema, where x is a meta-linguistic variable standing for any object $x \in \mathbb{N}$, and $s(x)$ is the object obtained after applying the successor function s at the meta-level. The deduction relation can now inductively be defined: only the above axiom and proof rules (instances of the above proof rule schema) may be employed, next to generalized reflexivity and generalized transitivity that hold for every proof system. As an example, we have $1 = 3$ as shown by its deduction:

$$\begin{array}{c} 1 \\ \hline 2 \\ \hline 3 \end{array}$$

which we read as stating that both 2 follows from 1 (that is, $1 = 2$), and 3 follows from 2 (that is, $2 = 3$), which can be combined in one deduction by transitivity. The above is one of the many possible deductions establishing the same conclusion (in this case 3) with the same premises (in this case just 1). *End of Example.*

An important aspect of proof theory is the analysis of proof systems. Recall, if we have $\vdash a$, we say that a is *provable*. Given a proof system D, one could consider the class of all provable objects $\{a \mid \vdash_D a\}$. Proof systems can be compared by comparing their classes of provable objects. In particular, one can ask whether a given proof rule is redundant. A proof rule is redundant if in a proof system *without* that proof rule, the class of provable objects is the same as for the proof system *with* that proof rule. We introduce two concepts that capture such redundancy, in essentially different ways: *derivability* and *admissibility* of proof rules.

Given a proof system D which has a simple proof rule

$$\frac{a_1 \quad \cdots \quad a_n}{b}$$

then if $a_1, \ldots, a_n \vdash_{D'} b$ in a proof system D$'$ without the above proof rule, we call that proof rule *derivable*. Clearly, a derivable proof rule is redundant, since the class of all provable objects of D and D$'$ are the same. Indeed, for any deduction in D where the rule is used, we can 'cut' out the rule and 'paste' in the deduction witnessing $a_1, \ldots, a_n \vdash_{D'} b$ at the place where the rule is used. If this 'cut and paste'-procedure is applied for every occurrence of the proof rule, one ends up with a deduction in D$'$.

Similarly, given a proof system D which has a complex proof rule

$$\frac{\begin{matrix} D_1 & & D_n \\ a_1 & \cdots & a_n \end{matrix}}{b}$$

then if $c_1, \ldots, c_{m_i} \vdash_{D'} a_i$ for all $1 \leq i \leq n$ implies $d_1, \ldots, d_k \vdash_{D'} b$ in a proof system D$'$ without the above proof rule, we call that proof rule *admissible* (the complex proof rule imposes conditions on how the premises of the involved deductions $c_1, \ldots, c_{m_i}$ are related to the premises of the constructed deduction $d_1, \ldots, d_k$). Informally, a complex proof rule is admissible if it can be mimicked by a construction involving the deductions D_1 up to D_n. Also an admissible proof rule is redundant. Consider a deduction of D in which the above complex proof rule is used. Consider the context where the proof rule occurs, such that the proof rule does not occur in the deductions of the premises, but each deduction D_i has a list of premises $c_1, \ldots, c_{m_i}$. We cut out the derivations with as conclusion the premises $a_1, \ldots, a_n$, to establish (possibly after weakening) $c_1, \ldots, c_{m_1} \vdash_{D'} a_1$ and $\cdots$ and $c_1, \ldots, c_{m_n} \vdash_{D'} a_n$, from which we then obtain a new deduction by the fact that the rule is admissible, which can be placed in the place of the rule. After this procedure is applied for every occurrence of the proof rule, from the leaves to the root, one ends up with a deduction in D$'$.

A proof rule is *weakly admissible* if we have that $\vdash_{D'} a_i$ for all $1 \leq i \leq n$ implies $\vdash_{D'} b$. Thus weakly admissible proof rules are not necessarily eliminable in arbitrary contexts. If a proof rule is not weakly admissible, it is not admissible either. Further, in some cases, the requirement of admissible proof rules that the resulting deduction $d_1, \ldots, d_k \vdash_{D'} b$ does not contain the proof rule *at all* is too strong. If our goal is to eliminate the rule from any deduction, it is sufficient that

the proof rule can be pushed upward in every deduction in which it occurs, and eliminated when it occurs near the top: so that either the size of the deductions used as premises become smaller, or the resulting deduction is indeed in D .

When comparing derivability with admissibility, there is an important distinction: derivability can be applied to any rule occurrence, but admissibility can only be applied to rule occurrences where the deductions of the premises are already free from occurrences of the redundant rule. In admissibility, the resulting deduction with the redundant rule eliminated is constructed from the leaves back to the root, whereas no such order is imposed when eliminating derivable rules. Moreover, to obtain a deduction where the redundant rule is eliminated, for admissible proof rules, the original deductions of the premises can be changed (except for for their premises and conclusion), whereas for derivable rules those deductions remain intact.

In fact, derivability of a proof rule is stronger than admissibility. This can be demonstrated by the following example, where we analyze a proof rule with conclusion b and a single premise a. Consider the following two properties:

$$\text{if } a = b \text{ then } \vdash a \text{ implies } \vdash b \tag{A.1}$$

$$\text{if } \vdash a \text{ implies } \vdash b \text{ then } a = b \tag{A.2}$$

The first property (A.1) states that derivability implies weak admissibility. Given a deduction $a = b$, we know that the provability of a implies the provability of b. This is a consequence of transitivity, by applying the deduction without premises of a in the place of the premise a in the deduction $a = b$ to obtain $\vdash b$. This argument also works when there are arbitrary additional premises $c_1; \ldots; c_m$ from which a follows, hence derivability also implies admissibility. Hence, the first property holds for every proof system.

The second property (A.2) states that weak admissibility implies derivability. However it can be shown that the second property does not hold in general: there is a proof system in which it fails. Hence, admissibility does not imply derivability, since weak admissibility already does not imply derivability.

Example A.3.9. Consider the set Z of objects, where 0 is zero, $s : Z \to Z$ the successor function, and $p : Z \to Z$ the predecessor function. Take the same axiom and proof rule schema as we did in the last example (so we have only the instances where $x \in N$):

$$\overline{0} \qquad \frac{x}{s(x)}$$

Comparing this proof system to the previous example, we see that their sets of provable objects must be the same. We now consider the simple proof rule schema

$$\frac{p(x)}{x}$$

with the question: is this proof rule admissible? Consider a deduction and the top-most occurrence of an instance of this proof rule (i.e. those occurrences where deductions of the premise do not have an instance of this proof rule as an occurrence).

Then it must be the case that the conclusion $p(x)$ is in N. We can then eliminate the proof rule by replacing its instance by an instance of the other rule, which deduces $s(p(x))$ from $p(x)$. Keep working downwards and we eventually obtain a deduction in which this proof rule no longer occurs. Hence the proof rule is admissible.

But is it derivable? Consider the following instance: 0 follows from 1. Since the instances of the remaining proof rule are limited to $x \in \mathbb{N}$, we cannot apply it to construct a deduction with the conclusion 0. Hence the new proof rule is not derivable. *End of Example.*

When considering the relative demonstrability relation $\vdash_D$ of a proof system $D = (O; =)$, one typically considers Γ to denote a sequence of objects in the case of $\Gamma \vdash_D a$. It is natural to extend the relative demonstrability relation to sets of objects too.

Definition A.3.10. Let O be a set of objects. Then $\Gamma \vdash_D a$ holds if and only if there exists a sequence Γ_0 of elements in Γ such that $\Gamma_0 \vdash_D a$.

In fact, we can generalize the succedent too.

Definition A.3.11. Let $\Gamma; \Delta \subseteq O$ be sets of objects. Then $\Gamma \vdash_D \Delta$ holds if and only if there exists sequence Γ_0 of elements in Γ and sequence Δ_0 of elements in Δ such that $\Gamma_0 \vdash_D \Delta_0$.

We construct a proof system for first-order classical logic in the style of Hilbert. Let Γ be a context (a finite sequence) of first-order formulas, and $\alpha; \beta; \gamma$ be first-order formulas.

Definition A.3.12. Let CL be a proof system consisting of:

1. the first-order formulas of classical logic as objects,
2. the smallest deduction relation $\vdash_{\mathbf{CL}}$ satisfying the conditions:

   (MP) $\Gamma \vdash_{\mathbf{CL}} (\alpha \to \beta)$ and $\Gamma \vdash_{\mathbf{CL}} \alpha$ implies $\Gamma \vdash_{\mathbf{CL}} \beta$,

   (G) $\Gamma \vdash_{\mathbf{CL}} \alpha$ implies $\Gamma \vdash_{\mathbf{CL}} (\forall y \binom{x}{y} \alpha)$
   where $x \notin FV(\Gamma)$ and either $y = x$ or $y \notin FV(\alpha)$,

   (A1) $\Gamma \vdash_{\mathbf{CL}} (\alpha \to (\beta \to \alpha))$,

   (A2) $\Gamma \vdash_{\mathbf{CL}} ((\alpha \to \beta) \to ((\alpha \to (\beta \to \gamma)) \to (\alpha \to \gamma)))$,

   (A3) $\Gamma \vdash_{\mathbf{CL}} ((\forall x(\alpha \to \beta)) \to (\alpha \to (\forall y \binom{x}{y} \beta)))$
   where $x \notin FV(\alpha)$ and either $y = x$ or $y \notin FV(\alpha)$,

   (DN) $\Gamma \vdash_{\mathbf{CL}} (\neg\neg\alpha \to \alpha)$,

   ($\forall$E) $\Gamma \vdash_{\mathbf{CL}} ((\forall x \alpha) \to \binom{x}{y}\alpha)$ where y remains free for x in α,

   (=I) $\Gamma \vdash_{\mathbf{CL}} (x \doteq x)$,

   (=E) $\Gamma \vdash_{\mathbf{CL}} ((x \doteq y) \to (\binom{z}{x}\alpha \to \binom{z}{y}\alpha))$ where $x; y$ remain free for z in α.

It is easy to see that the above proof system is finitary. Each first-order formula is a finite object. Further, the witness of a deduction, $\Gamma \vdash_{\mathbf{CL}} \varphi$, is a proof tree with premises in Γ and conclusion φ. The leaves of the proof tree are instances of an axiom scheme or a premise in Γ, and the internal nodes of the proof tree are either obtained from the proof rule (MP) where there are two branches, or the proof rule (G) where there is one branch. These proof trees also satisfy generalized reflexivity and generalized transitivity: compositions of proof trees are themselves proof trees.

This proof system is in the style of Hilbert, since the only proof rules are *modus ponens* (MP) and *generalization* (G). Proof rule (MP) is also called implication elimination ($\to$E), and (G) is also called universal introduction ($\forall$I). Note the distinction between (MP) stated above with additional premises in the context Γ, and the stronger condition below:

$$\vdash_{\mathbf{CL}} (\varphi \to \psi) \text{ and } \vdash_{\mathbf{CL}} \varphi \text{ implies } \vdash_{\mathbf{CL}} \psi \qquad \text{(MP')}$$

which is stated only at the level of provability, i.e. without context. We call (MP') a 'rule of provability', whereas (MP) is called a 'proof rule' (since it is equivalent to $(\varphi \to \psi);\ \varphi \vdash_{\mathbf{CL}} \psi$). The difference between these two is that only for a proof system with (MP) we can establish the following property, since in a proof system that has (MP') instead of (MP) we have only deductions from premises obtained from generalized reflexivity or generalization.

Lemma A.3.13. *If* $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi)$ *implies* $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi$.

Proof. Applying weakening we obtain $\Gamma;\ \varphi \vdash_{\mathbf{CL}} (\varphi \to \psi)$, and by generalized reflexivity we have $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \varphi$. Hence by (MP) we obtain $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi$. □

In fact, the converse also holds.

Theorem A.3.14 (Deduction theorem). *If* $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi$ *then* $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi)$.

Proof. Consider the proof tree corresponding to $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi$, and perform induction on the structure of that tree. Either the proof tree is a leave (base case), or it is an instance of the (MP) proof rule with two smaller proof trees on top, or an instance of the (G) proof rule with one smaller proof tree on top.

Base case 1. If $\psi = \varphi$ then we obtain $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi)$ from (MP), (A1) and (A2).

Base case 2. If ψ was obtained by reflexivity from Γ or ψ is an instance of an axiom scheme, then we obtain $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi)$ from (MP) and (A1).

Induction step (MP). Let ψ be the conclusion, where $\Gamma;\ \varphi \vdash_{\mathbf{CL}} (\psi' \to \psi)$ and $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi'$ are on top. Our induction hypotheses are $\Gamma;\ \varphi \vdash_{\mathbf{CL}} (\psi' \to \psi)$ implies $\Gamma \vdash_{\mathbf{CL}} (\varphi \to (\psi' \to \psi))$, and $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi'$ implies $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi')$. Then we obtain $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi)$ from (MP) and (A2).

Induction step (G). Let $\psi = (\forall y\, (\frac{x}{y})\, \psi')$ be the conclusion, where $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi'$ is on top and $x \notin FV(\varphi)$. Our induction hypothesis is $\Gamma;\ \varphi \vdash_{\mathbf{CL}} \psi'$ implies $\Gamma \vdash_{\mathbf{CL}} (\varphi \to \psi')$. We obtain the result from (MP) and (A3). □

The deduction theorem proves that the following proof rule is admissible:

$$\frac{\boxed{}}{!}\ (\,!\ \mathrm{I})$$

where the deduction (inside the box) with conclusion may use the assumption , which may not be a premise of the overall deduction. The at the top of the box means that the assumption is closed, that is, although is a premise of the inner deduction it is not a premise in the resulting, outer deduction. Another consequence of the deduction theorem is that we can clear out the context. Let $= {}_1; : : : ; {}_n$, then (!) is abbreviates ($_1$! (: : : ! ($_n$!) : : :)). In case is an empty sequence, then (!) is just . Note that this does not work for arbitrary sets, since our formulas are finitary.

Corollary A.3.15. *`CL if and only if `CL (!).*

Lemma A.3.16. *We have the following derived axioms and proof rules:*

(>I) *`CL >,*

(?E) *? `CL ,*

(DN) *: : `CL ,*

(! I) *; `CL implies `CL (!),*

(! E) *(!); `CL ,*

(^I) *; `CL (^),*

(^EL) *(^) `CL ,*

(^ER) *(^) `CL ,*

(_IL) *`CL (_),*

(_IR) *`CL (_),*

(_E) *(!); (!); (_) `CL ,*

(8I) *`CL implies `CL (8y (^x_y))*
where x 2 FV ( ) and either y = x or y 2 FV ( ),

(8E) *(8x) `CL (^x_y) where y remains free for x in ,*

(9I) *(^x_y) `CL (9x) where y remains free for x in ,*

(9E) *`CL (9x ) and ; ($^x_y$) `CL implies `CL*
where y 2 FV (;) and either x = y or x 2 FV ( ),

(=I) $\vdash_{\mathbf{CL}} (x \doteq x)$,

(=E) $(x \doteq y) \vdash_{\mathbf{CL}} (\varphi^z_x) \to (\varphi^z_y)$ *where* x, y *remain free for* z *in* φ.

Proof. See *Basic Proof Theory* by Troelstra and Schwitchenberg, Section 2.4. □

The proof system consisting of the axiom and proof rules of Lemma A.3.16 is called *natural deduction*. From this proof system it is also possible to derive the axioms and proof rules of Definition A.3.12.

Further, employing the fact that relative demonstrability is a relation on two lists of formulas, we have the following properties. In the following, let Γ and Δ be contexts (finite sequences of formulas). The antecedent context can be seen as a conjunction of its formulas, and the succedent context can be seen as a disjunction of its formulas.

Corollary A.3.17. $\Gamma \vdash_{\mathbf{CL}} \Delta$ *if and only if* $(\bigwedge \Gamma) \vdash_{\mathbf{CL}} \bigvee \Delta$.

Let $\Gamma = \gamma_1, \ldots, \gamma_n$, then $\bigwedge \Gamma$ is an abbreviation for the formula $(\gamma_1 \wedge (\ldots \wedge (\gamma_n \wedge \top)\ldots))$. In case Γ is empty, $\bigwedge \Gamma$ is just $\top$.

Corollary A.3.18. $\vdash_{\mathbf{CL}} \Delta$ *if and only if* $\top \vdash_{\mathbf{CL}} \bigvee \Delta$.

Lemma A.3.19. $\Gamma \vdash_{\mathbf{CL}} \Delta$ *if and only if* $\vdash_{\mathbf{CL}} (\bigwedge \Gamma) \vee \bigvee \Delta$.

Proof. Given $\Gamma \vdash_{\mathbf{CL}} \Delta$, and apply $(\vee$IL) and $(\vee$IR), to obtain $\vdash_{\mathbf{CL}} (\bigwedge \Gamma) \vee \bigvee \Delta$. The other direction is more interesting. Given $\vdash_{\mathbf{CL}} (\bigwedge \Gamma) \vee \bigvee \Delta$, and let Γ and Δ be arbitrary. We assume $\Gamma \vdash_{\mathbf{CL}}$ and $\Delta \vdash_{\mathbf{CL}}$. From these, we have $\Gamma \vdash_{\mathbf{CL}} (\bigwedge \Gamma \to \bot)$ and $\Delta \vdash_{\mathbf{CL}} (\bigwedge \Gamma \to \bot)$ by Corollary A.3.15. Hence $\Gamma \vdash_{\mathbf{CL}} (\bigwedge \Gamma \to \bot)$ by $(\vee$E), and thus $\Gamma \vdash_{\mathbf{CL}} \Delta$. □

Let $\Delta = \delta_1, \ldots, \delta_n$, then $\bigvee \Delta$ is an abbreviation for the formula $(\delta_1 \vee (\ldots \vee (\delta_n \vee \bot)\ldots))$. In case Δ is empty, $\bigvee \Delta$ is just $\bot$.

Corollary A.3.20. $\Gamma \vdash_{\mathbf{CL}} \Delta$ *if and only if* $\Gamma \vdash_{\mathbf{CL}} \bigvee \Delta$.

The observations above together motivate the introduction of another abbreviation. Let $\Gamma \Rightarrow \Delta$ abbreviate $\bigwedge \Gamma \to \bigvee \Delta$. We call $\Gamma \Rightarrow \Delta$ a *sequent*.

Lemma A.3.21. $\Gamma \vdash_{\mathbf{CL}} \Delta$ *if and only if* $\vdash_{\mathbf{CL}} \Gamma \Rightarrow \Delta$.

Lemma A.3.22. *We have the following derived axioms and rules of proof:*

(L$\bot$) $\Gamma, \bot, \Delta \vdash_{\mathbf{CL}}$,

(L$\wedge$) $\Gamma, \varphi, \psi, \Delta \vdash_{\mathbf{CL}}$ *implies* $\Gamma, (\varphi \wedge \psi), \Delta \vdash_{\mathbf{CL}}$,

(R$\wedge$) $\Gamma \vdash_{\mathbf{CL}} \varphi, \Delta$ *and* $\Gamma \vdash_{\mathbf{CL}} \psi, \Delta$ *implies* $\Gamma \vdash_{\mathbf{CL}} (\varphi \wedge \psi), \Delta$,

(L$\vee$) $\Gamma, \varphi, \Delta \vdash_{\mathbf{CL}}$ *and* $\Gamma, \psi, \Delta \vdash_{\mathbf{CL}}$ *implies* $\Gamma, (\varphi \vee \psi), \Delta \vdash_{\mathbf{CL}}$,

(R$\vee$) $\Gamma \vdash_{\mathbf{CL}} \varphi, \psi, \Delta$ *implies* $\Gamma \vdash_{\mathbf{CL}} (\varphi \vee \psi), \Delta$,

(L $!$) $\vdash_{\mathbf{CL}}$ $\quad$; $\quad$ and $\vdash_{\mathbf{CL}}$; $\quad$ implies $\vdash_{\mathbf{CL}}$ ($!$); $\quad$,

(R $!$) $\vdash_{\mathbf{CL}}$; $\quad$; implies $\vdash_{\mathbf{CL}}$ $\quad$;($!$),

(L8) $\vdash_{\mathbf{CL}}$ $(8x\ ); \binom{x}{y}$; $\quad$ implies $\vdash_{\mathbf{CL}}$ $(8x\ );$ $\quad$
where y remains free for x in ,

(R8) $\vdash_{\mathbf{CL}}$ $\quad$; $\binom{x}{y}$ implies $\vdash_{\mathbf{CL}}$ $\quad$; $(8x\ )$ where y is fresh,

(L9) $\vdash_{\mathbf{CL}}$ $\binom{x}{y}$; $\quad$ implies $\vdash_{\mathbf{CL}}$ $(9x\ );$ $\quad$ where y is fresh,

(R9) $\vdash_{\mathbf{CL}}$ $\quad$; $\binom{x}{y}$; $(9x\ )$ implies $\vdash_{\mathbf{CL}}$ $\quad$; $(9x\ )$
where y remains free for x in ,

(Ref) $\vdash_{\mathbf{CL}} x \doteq x;$ $\quad$ implies $\vdash_{\mathbf{CL}}$ $\quad$,

(Rep) $\vdash_{\mathbf{CL}} x \doteq y; \binom{z}{y}$; $\binom{z}{x}$; $\quad$ implies $\vdash_{\mathbf{CL}} x \doteq y; \binom{z}{x}$; $\quad$
where is a primitive formula.

where the condition of freshness of y means that y does not occur free in the contexts ; nor is y = x.

Proof. See *Basic Proof Theory* by Troelstra and Schwitchenberg, Section 3.5 and Section 4.7. □

The proof system consisting of the axioms and proof rules of Lemma A.3.21 and Lemma A.3.22 is called *sequent calculus*. The fact that this is indeed a proof system is non-trivial, since to show that generalized transitivity holds it relies on the elimination of cuts (not further discussed here). In fact, as can be easily seen from the definition above, this proof system has a recursive relative demonstrability relation, an important result due to Gentzen. From this proof system it is also possible to derive the axioms and proof rules of Lemma A.3.16.

The classically provable formulas are formulas that can be proven using one of the proof systems described above.

Definition A.3.23. A theory T of first-order formulas is *deductively closed* if for every subset T and first-order formula such that $\vdash_{\mathbf{CL}}$ also $2\ T$.

The notion of deductively closed can be generalized to other proof systems as well, and is not specific to CL.

Proposition A.3.24. *Every first-order theory* $Th_1(\mathfrak{A})$ *of a structure* $\mathfrak{A}$ *is deductively closed (with respect to* CL*).*

A proof system is *finitary* whenever we have recursive enumerability of the provable formulas. Finitary proof systems are useful in practice, since they allow a computer to systematically generate proofs.

A.4 Soundness and completeness

The main results of first-order logic are now given. We assume Γ is a set of first-order formulas, and φ is a first-order formula.

Lemma A.4.1 (Soundness). $\Gamma \vdash^{\mathbf{CL}} \varphi$ *implies* $\Gamma \models^{\mathbf{CL}} \varphi$.

Lemma A.4.2 (Completeness). $\Gamma \models^{\mathbf{CL}} \varphi$ *implies* $\Gamma \vdash^{\mathbf{CL}} \varphi$.

Proof. Originally proven by Gödel [93]. See also the proof by Henkin [108], and Kleene's overview [136]. It can also be formally established using the interactive theorem provers Isabelle/HOL [191] and Coq [134]. □

Although we already established compactness of the satisfiability relation, in the sense of the compactness theorem (see Theorem A.2.10), we can now also show compactness of the semantic consequence relation in an alternative way. This follows easily from the fact that we have a finitary proof system that is sound and complete.

Theorem A.4.3. $\Gamma \models^{\mathbf{CL}} \varphi$ *if and only if* $\Gamma_0 \models^{\mathbf{CL}} \varphi$ *for some finite subset* $\Gamma_0 \subseteq \Gamma$.

Proof. Given that $\Gamma \models^{\mathbf{CL}} \varphi$ holds, by completeness we have that $\Gamma \vdash^{\mathbf{CL}} \varphi$. Since our proof system is finitary, there are only finitely many formulas in Γ used in the deduction of φ. Let Γ_0 be those formulas. Hence $\Gamma_0 \vdash^{\mathbf{CL}} \varphi$, and by soundness $\Gamma_0 \models^{\mathbf{CL}} \varphi$. The other direction is easy: given that $\Gamma_0 \models^{\mathbf{CL}} \varphi$ for some finite subset $\Gamma_0 \subseteq \Gamma$, then we may always add more formulas to obtain $\Gamma \models^{\mathbf{CL}} \varphi$. □

Theorem A.4.4 (Undecidability). *There is no algorithm that can decide whether* $\Gamma \models^{\mathbf{CL}} \varphi$ *or not, for every signature, theory* Γ*, and formula* φ.

Proof. This is Church's theorem, see [23]. This result can also be established using Coq [120]. □

Note that the undecidability result is an existential statement: it does not mean there are no some signatures, theories, and formulas, for which the satisfiability relation is decidable. In fact, there are signatures and theories, for which the satisfiability relation is decidable.

A.5 Adding back terms

Up until now we have only considered signatures that consists of constant symbols, which are associated to a non-zero arity. In particular, signatures may contain predicate symbols (of arity 1) and relations symbols (of arity n for some $n > 1$) or constant symbols with third-order or higher-order arity. From a practical perspective, however, this set-up limits our ability to directly refer to individuals of the domain. Although we are able to give a name to individuals, e.g. in the context of a quantifier where an individual variable ranges over elements of the domain, and we are able to identify two individuals, we lack the ability to directly

refer to individuals by some name, that denotes the individual regardless of the context in which that name appears.

In this section we consider an extension of the assertion language of classical logic in which we add facilities for referring to individuals of the domain directly. In our discussion we shall not formalize all aspects of our extension explicitly, leaving some details to the reader, and focus on first-order assertion languages. The purpose of our exposition is to show that this extension does not change the expressive power of first-order logic.

We have added $\doteq$ as a logical symbol, but consider for a moment an alternative approach: what if $\doteq$ is a 2-ary relation symbol? In second-order languages it is not needed to add such a relation to the signature, since the concept of identity is indirectly definable. The second-order sentence

$$\forall x, y.\ (x \doteq y) \leftrightarrow \forall P.\ P(x) \leftrightarrow P(y)$$

expresses that identity satisfies Leibniz's law of the *identity of indiscernibles* and its converse, the *indiscernability of identicals*: two elements share every property if and only if the two elements are identical. However, in first-order languages without identity as logical symbol, this cannot be expressed. Every structure gives an interpretation to the relation symbols, including $\doteq$ taken as relation symbol. A structure $\mathcal{A} = (A, I)$ has a *standard interpretation of identity* if I assigns to the identity relation symbol $\doteq$ the value $\{(x, y) \mid x = y\} \subseteq A \times A$. Thus, in the standard interpretation, the extension of identity coincides with our meta-level concept of equality. This amounts to the same what we have accomplished in our definition of the satisfaction relation for the logical symbol $\doteq$.

Now we can introduce the derived concept of *unique existence*. We introduce the following abbreviation:

$$\exists! x\ \varphi \quad \text{abbreviates} \quad \exists x(\varphi \wedge \forall y(\varphi[y/x] \rightarrow x \doteq y))$$

where y is fresh (i.e. y is not x and does not occur in φ), and $\varphi[y/x]$ is obtained from φ by renaming x to y. Semantically, we have that

$$\mathcal{A}, \beta \models \exists! x\ \varphi \quad \text{holds iff} \quad \mathcal{A}, \beta[x := a] \models \varphi \quad \text{for a unique } a \in A.$$

The ability to express unique existence has two important consequences. Suppose that $\mathcal{A} \models \exists! x\ \varphi$ holds. Consider the case in which formula φ has only x as a free variable. By the coincidence condition, we then have that there must exists a unique $a \in A$ regardless of the valuation β. As such we say that φ *defines* the element a (in $\mathcal{A}$). Similarly, consider the cases where all the free variables of formula φ are in the sequence of variables $x_1, \ldots, x_n, x$ (where we assume that all variables are distinct). Again by the coincidence condition we have that there must exists a unique total function $f : A^n \to A$ regardless of the valuation β:

$$f = f(a_1, \ldots, a_n, a) \text{ iff } \mathcal{A}, \beta[x_1 := a_1] \ldots [x_n := a_n][x := a] \models \varphi$$

and we can again say that φ *defines* the total function f (in $\mathcal{A}$).

Example A.5.1. Suppose we have a signature comprising the following constant symbols (and nothing else):

- we have the predicate symbol Z of arity 1,
- we have the relation symbol S of arity 2.

We can now state that there is *exactly* one element of the domain x such that $Z(x)$ holds by asserting

$$\exists!x Z(x).$$

Further, we can state that for every element of the domain x, there must be a *unique* y such that $S(x,y)$ holds, by asserting

$$\forall x \exists!y S(x,y).$$

However, can we also state that every element of the domain x is *reachable* either directly through Z, i.e. $Z(x)$ holds, or through a chain of S, i.e. $S(y,x)$ for some other reachable y? The following second-order sentence states this property:

$$\forall R.(\forall x.Z(x) \Rightarrow R(x)) \wedge (\forall x,y.R(y) \wedge S(y,x) \Rightarrow R(x)) \Rightarrow \forall x R(x)$$

Note that not all structures (with a standard interpretation of identity) that satisfy the two first-order sentences also satisfy the second-order sentence.
End of Example.

A useful pattern emerges: if a predicate holds for exactly one element d of the domain, we can use that predicate as a way to identify element d. Intuitively, such a predicate identifies the element for which it holds. Similarly, if for a relation R, we have exactly one element of the domain d given elements $d_1,\ldots,d_{n-1}$ and $(d_1,\ldots,d_{n-1},d) \in R$, then we can use that relation as a way to identify d once we are also able to identify the values of the first $n-1$ places. Taking both cases together, we say that a second-order constant symbol C of arity n has the property of *functionality* if

$$\forall x_1,\ldots,x_{n-1} \exists!x C(x_1,\ldots,x_{n-1},x),$$

holds (if $n = 1$ the universal quantifiers are dropped).

We now extend our definition of signature in which we explicitly *declare* which constant symbols must have the above property of functionality. Predicate symbols of arity 1 that have the property of functionality are called *individual symbols* (or, more precisely, individual constant symbols). Relation symbols of arity n for $n > 1$ that have the property of functionality are called *function symbols*.

Remark A.5.2. We must make sure not to confuse constant symbols and individual (constant) symbols. Although individual symbols are constant symbols, the converse is not the case: all predicate symbols, relation symbols and function symbols are also constant symbols in the sense that their meaning does not depend on the context and remains fixed.

We fix a first-order signature . As a syntactical convention, the uppercase constant symbols C are used for the constant symbols of our signature. If a constant symbol, say B, has been declared to have the property of functionality, then we use the lowercase symbol b instead. For individual constant symbols, we typically use the lowercase symbol c, and for function symbols we typically use the lowercase symbol f.

We now define terms and formulas and revisit some of the concepts introduced earlier. Note that we restrict ourselves to first-order assertion languages.

Definition A.5.3 (Terms). A *term* of is constructed inductively as follows:

- any individual variable symbol x is a term,
- any individual constant symbol c is a term,
- given terms $t_1;:::;t_n$ and function symbol f of arity $n + 1$ then $f(t_1;:::;t_n)$ is a term.

In the third clause, the terms $t_1;:::;t_n$ are called the *arguments* of the function symbol f in the term $f(t_1;:::;t_n)$. To be able to speak of the arity of a term, we let every term have the arity 0. We may see a term as a parse tree in which individual variable symbols or individual constant symbols appear at the leaves and function symbols at the branches.

Definition A.5.4 (Formulas). A *formula* is constructed inductively as follows:

- $?$ is a formula,
- $P(t_1;:::;t_n)$ is a formula if P is a constant symbol of arity n and $t_1;:::;t_n$ are terms,
- (!) is a formula if and are formulas,
- $(8x\ )$ is a formula if is a formula and x an individual variable.

In the second clause, we do not restrict ourselves to only predicate symbols and relation symbols. All constant symbols (including those that have the property of functionality) are allowed at this level. This turns out to be useful when we show that extending first-order logic with terms does not increase the expressive power.

We can again define the set of free variables $FV(\ )$ and bound variables $BV(\ )$, but to do so we need to also introduce the set of free variables $FV(t)$ and bound variables $BV(t)$ for a given term t. The set of free variables $FV(t)$ consists precisely of all variables that occur in the term t, and the set of bound variables $BV(t)$ is empty. We can also extend variable renaming to terms, so that if is a renaming of variable symbols (that preserves arity) we can define (t) by simultaneously replacing all the variable occurrences in t according to .

A more general operation than variable renaming is the operation of substitution. Substitution is not simply a matter of replacing variables by terms. Rather, we define substitution as a transformation of formulas in two stages. This can be

motivated by the following example. Given the formula $(\forall x{:}y = z)$, if we naively replace y by a term in which x occurs as a variable, then that variable now falls under the scope of the quantifier that binds x. The resulting formula thus has a term which may now have a different meaning than when considering that term in a different context, outside of the scope of the quantifier. That x would fall under the scope of the quantifier is called *variable capturing*. We intend to avoid variable capturing to ensure we can show the relation between (syntactic) substitutions and the semantics of terms, later on.

We also have substitutions of variables for terms that prevents variable capturing. Capture-avoiding substitution can be generalized to arbitrary formulas by first performing a renaming to obtain an alphabetic variant, for which the condition of the capture-avoiding substitution is always satisfied.

Definition A.5.5 (Capture-avoiding substitutions). A substitution $[x := t]$ is defined by replacing in all free occurrences of x by t, as long as the bound variables of are disjoint from the free variables of t.

It is now also possible to extend the definition of structures, so that the interpretation is extended to also interpret individual symbols and function symbols in a certain way to guarantee that the declared properties hold. For a given interpretation I, we let c^I denote the interpretation of the constant symbol c, and f^I denote the interpretation of the function symbol f.

Definition A.5.6 (Evaluation function). Given a structure $\mathcal{A} = (A; I)$ and a valuation of $\mathcal{A}$, and a term t. The evaluation function $\mathcal{A}\llbracket t \rrbracket^{\mathbf{CL}}$ is defined inductively on the structure of t:

- $\mathcal{A}\llbracket x \rrbracket^{\mathbf{CL}} = (x)$,
- $\mathcal{A}\llbracket c \rrbracket^{\mathbf{CL}} = a$ where $a \in c^I$,
- $\mathcal{A}\llbracket f(t_1, \ldots, t_n) \rrbracket^{\mathbf{CL}} = a$ where $(\mathcal{A}\llbracket t_1 \rrbracket^{\mathbf{CL}}, \ldots, \mathcal{A}\llbracket t_n \rrbracket^{\mathbf{CL}}, a) \in f^I$.

We may drop the superscript $\mathbf{CL}$ if clear from context. In the second and third clause, we can pick *any* such a. Due to the restriction on the interpretation I on constant symbols which have the property of functionality, we know there exists a unique one. We may also simply write (t) for $\mathcal{A}\llbracket t \rrbracket^{\mathbf{CL}}$, since every valuation is defined in a context where there is a known structure $\mathcal{A}$.

It is also possible to redefine the satisfaction relation for formulas with terms, but we shall leave out the details. An important consequence of defining such satisfaction relation is the following lemma.

Lemma A.5.7 (Substitution lemma).

$$\mathcal{A}; \quad \models^{\mathbf{CL}} \quad [x := t] \textit{ if and only if } \mathcal{A}; \quad [x := \quad (t)] \models^{\mathbf{CL}} \quad :$$

There is also a translation of formulas with terms to formulas without terms which has the same denotation, which allows us to eliminate all terms.

Bibliographic notes

C. Grabmayer's PhD thesis [97] contains a more detailed analysis of abstract proof systems and introduces the notions of derivability and admissibility of proof rules in an abstract setting [98].

Appendix B

Hoare's logic

Hoare's logic (some authors write: Hoare logic) was introduced by C.A.R. Hoare in 1969 [118], based on the inductive assertion method on flow charts as was introduced in 1967 by R.W. Floyd [84, 62]. See also the collection of fundamental papers on program verification by T.T.R. Colburn, J.H. Fetzer, and R.L. Rankin [54].

The main philosophical idea underlying both Hoare's logic and Floyd's inductive assertion method is that we have two 'modes' of description: that of *being*, and that of *change*. These two concepts are also understood as the *statics* and *dynamics* of a system, respectively. By making use of a logical language (e.g. first-order logic or separation logic) we can describe the state of being, by which one can think of a static snapshot in time of the currently realized memory state of a computer. By making use of a control structuring language (e.g. a flow chart or a program in a programming language), we can describe the change of state, which essentially describes a dynamic process which transforms initial states into final states or chains together such transformations. By combining descriptions of both modes, being and change, we obtain the program specification (also called the Hoare triple):

$$f \; g S f \; g$$

where is a description of the possible initial states (the precondition), S is a description of the program that transforms an initial state into a final state, and is a description of the possible final states (the postcondition). A program S is correct with respect to a specification whenever it is indeed the case that, after executing the program from an initial state that satisfies the precondition, we obtain a state as prescribed by the postcondition. In some sense, a program specification describes the expected behavior of a program, whereas a program simply describes behavior.

Ideally there is no need for program specifications: by making sure a program exactly describes the change intended, it is always correct. Popularly, this is known by the phrase: "it is not a bug, it is a feature." However, the program specification adds redundancy to the program. The program describes *how* the state changes from an initial state into a final state, whereas the pre- and postconditions in

a program specification describes logically *what* is the state of being before and after the execution of the program. The basic premise of program verification is that humans err—and by means of program specifications, where we combine two different languages in which one describes both the 'intended' program behavior and the 'actual' program behavior, we can detect errors if these two behaviors do not match.

Concretely, during execution of a program, there exists a current state of the values of memory. A basic program either performs a test, or performs an operation. A test is an inspection of the current state to check whether a condition on the state holds or not. An operation performs an action that may or may not change the state. By performing tests and operations, programs direct or control the flow of states, from an initial state to possibly a final state. Complex programs are composed out of programs with the intention to structure the flow of control, where tests can be used to influence the direction of flow of control. A processor is the component of a computer which, step by step, either performs a test or an operation, as specified by a program. The program thus is an input to the processor, and the program indirectly controls which tests and operations are performed by the processor. The result of the tests consequently direct the flow of control as described by a program, thus resulting in a feedback loop.

This conception of program originated in 1945, by the initial designs of John von Neumann's computer architecture [94]. Already in this early work, we can see that memory and program are separate, and we follow this design choice and also exclude so-called self-modifying programs. For practical reasons, we also see differences in the storage locations of memory: the current state of the memory of a computer can be divided into the internal state and the external state. The internal state stores the value of registers, the external state stores the value of addressable memory. The internal state is always directly accessible in tests, and can be operated on. The external state can be loaded and stored, being special operations. Thus, we do not have direct access to external state, only indirectly through load and store operations.

This division between internal and external state is more blurred in modern computer architectures of the past decades, by the introduction of cached memory—where parts of the external state is duplicated into hidden registers, which cannot be directly controlled by the program but mirror the behavior of the external memory. This architectural choice is mainly motivated by a further division of the use of the external memory into so-called stack memory and heap memory [111].

Since memory management is error-prone, high-level languages feature automatic memory management, while some high-level languages also still allow manual memory management. One form of automatic memory management is by using block scopes, in which a local variable is temporarily allocated on the stack and deallocated once the block has finished executing. Another form of automatic memory management is using garbage-collected heaps, in which all heap memory is scanned and regions of memory automatically deallocated if they no longer can influence the outcome of the execution of a program. We shall consider a programming language which features automatic stack memory management, but

manual heap memory management without garbage collection.

In this chapter we shall introduce the abstract syntax of programs, introduce three different styles of giving programs semantics (operational semantics, denotational semantics, and axiomatic semantics), and discuss the proof system for deriving correct program specifications called Hoare's logic. The ideas presented in Sections B.1, B.2, B.3 can be found in any competent book on program verification, such as [71, 61, 227, 100, 86, 10]. The use of program signatures and machine models, however, may be novel. The main motivation for revisiting the basic material is to present it in such a way to make it easy to adapt to separation logic, in Chapter 4. The material presented in Section B.5 is largely based on [29], but the presentation here is novel and the proof system more modular than in [29].

B.1 Syntax of programs

This section describes the syntax of a simplified programming language, necessary for supporting the semantics and proof system for reasoning about correctness of programs. Although we restrict ourselves to a simplified programming language, the expressivity of programs is nonetheless interesting: we include the Turing-complete programming languages.

When considering the syntax of programs, there is a design choice in formulating the programming language. One could give the concrete syntax of programs being particular constructions of statements, or one abstracts from the primitive operations and tests. In the latter case one can obtain the concrete syntax as a particular instance of the abstract syntax. This set-up of the syntax of programs is not much different than the set-up of formulas in the assertion language, where one separates the logical from the non-logical symbols by the introduction of a signature. Similarly, we could introduce the concept of a *program signature* which collects the primitive operations and tests out of which the statements are constructed.

Given a program signature that consists of the primitive operations and tests, we can form the statements of a program. Complex statements are constructed from compositions of simpler statements, recursively. One can represent statements by their parse trees in which at the leaves of the tree primitive operations and tests occur. One may compare statements of the programming language to formulas of the assertion language.

In fact, when one wants to extend our programming language to include recursive procedures, the abstract syntax approach is beneficial, since procedures can be considered particular primitive operations with a fixed interpretation given by a system of procedure declarations. This is not much different than having recursive predicates in the assertion language. We shall first focus on programs without recursive procedures, and can later add recursive procedures: in a sense, recursive procedures are an orthogonal concern.

Before introducing the formal definition of statements, we consider the possible effects that the executions of programs have on the states of a machine. We model these effects as a state transition, and the behavior of a program essentially is the possible sequences of state transitions. Each such sequence is also called an

execution. An important design choice, however, is to contain the e ect of primitive operations and tests, by limiting what part of the state an operation or test can (at most) be accessed, and what part of the state can (at most) be changed by an operation.

We reuse the concept of variables (see De nition A.1.1) to denote parts of the state. The accessible variables of a program restricts what part of the state can in uence program behavior, and the changed variables of a program restricts what part of the state can be modi ed in the state transitions that constitute program behavior. In the context of a program we speak of *program variables*, whereas in the context of assertions we speak of *logical variables*. Although it is possible to also consider higher-order program variables, also called subscripted variables, we restrict ourselves to rst-order program variables.

Given a program variable x, a *polarized variable* is either x or $\overline{x}$ (we say `input $x$' or `output x', respectively). The absence of the line indicates that the program variable is accessible, and the presence of the line indicates that the program variable is changed. Note that the presence of the line on top of a program variable makes it a di erent polarized variable, i.e. $x \neq \overline{x}$.

De nition B.1.1 (Program signature). A *program signature* consists of a recursive set of operations and tests, such that each operation is associated with a nite set of polarized variables, and each test is associated with a nite set of (accessible) program variables.

We typically denote a program signature by Σ. If P is an operation of Σ, then we may speak of the accessible program variables $x_1, \ldots, x_n$ of P and the changed program variables $y_1, \ldots, y_m$ of P to mean that P is associated to the nite set $\{x_1, \ldots, x_n, \overline{y_1}, \ldots, \overline{y_m}\}$ of polarized variables. If T is a test of Σ, we write $T(x_1, \ldots, x_n)$ to mean that $x_1, \ldots, x_n$ are the accessible program variables associated to T.

Every rst-order signature Σ induces a program signature, where all the tests are quanti er-free formulas. Although the tests are xed, it still remains a design decision to select the appropriate operations: the selection of primitive operations thus a ects what computations can be expressed by a program.

De nition B.1.2. A rst-order program signature $\mathrm{FPS}(\Sigma)$ is a program signature such that every test with accessible program variables $x_1, \ldots, x_n$ corresponds to a quanti er-free formula $\varphi(x_1, \ldots, x_n)$, and also includes:

- the assignment operation $y := x$
  (where x is an accessible and y is a changed program variable).

The rst-order program signature corresponds to register machines, where program variables are registers of the machine, and tests work on the registers of the machine. Note that the operations of the inherited program signature may access and change arbitrary program variables.

Block programs manipulate registers but may also temporarily store values by pushing them on the stack, and later retrieve old values by popping them from

the stack. This is useful for implementing local variables, which are needed for introducing terms later on.

De nition B.1.3. A block program signature BPS( ) is a rst-order program signature FPS( ) that also includes:

- the parallel assignment operation $\vec{y} := \vec{x}$
  (where $\vec{x} = x_1; \ldots; x_n$ are accessible and $\vec{y} = y_1; \ldots; y_n$ are changed),
- the push operation push (x)
  (where x is an accessible program variable),
- the pop operation pop (x)
  (where x is a changed program variable).

Note that for a push operation push (x) there are no changed variables. Although the stack is modi ed by this operation, the stack is left implicit (as an implementation detail) and as such not represented by a program variable.

Pointer programs not only manipulate values assigned to program variables but also values assigned to locations on the heap. As such, we consider an extension of rst-order program signatures which includes operations for manipulating the heap. A pointer program signature also includes operations for looking up a value from the heap (lookup), modifying a value on the heap (mutation), allocating a new location with an initial value (allocation), and deallocating a location (deallocation).

De nition B.1.4. A pointer program signature PPS( ) is a rst-order program signature FPS( ) that also includes:

- the lookup operation x := [y]
  (where y is an accessible and x is a changed program variable),
- the mutation operation [x] := y
  (where x and y are accessible program variables),
- the allocation operation x := new (y)
  (where y is an accessible and x is a changed program variable),
- the deallocation operation delete (x)
  (where x is an accessible program variable).

Note that for a mutation operation [x] := y there are no changed variables. Although the heap is modi ed by this operation, the heap is implicit and as such not represented by a program variable. Also the lookup, allocation and deallocation operations above have a side-e ect, namely they modify the implicit heap. This phenomenon, where no variables (or not all) are changed but there is a (hidden) state change, is in general called a side-e ect.

Each program signature (including the standard, pointer and block program signatures) can be used to generate statements. We x some program signature for the remainder of this section, unless explicitly mentioned otherwise.

De nition B.1.5 (Statements). Given a program signature . A statement is constructed inductively as follows:

1. O is a statement (called primitive operation) where O is an operation of ,
2. skip is a statement (called no operation),
3. halt is a statement (called halt operation),
4. $S_1; S_2$ is a statement (called sequential composition) given that S_1 and S_2 are statements,
5. if T then S_1 else S_2 is a statement (called conditional statement) given that T is a test of , and S_1 and S_2 are statements,
6. while T do S od is a statement (called looping statement) given that T is a test of and S is a statement.

All statements are constructed by one of these ve clauses. Alternatively, we can de ne statements by the following abstract grammar:

$$S; S_1; S_2 ::= O \mid \text{skip} \mid \text{halt} \mid S_1; S_2 \mid \text{if } T \text{ then } S_1 \text{ else } S_2 \mid \text{while } T \text{ do } S \text{ od}:$$

The rst two clauses construct primitive statements, the last three clauses construct complex statements. Sequential composition $S_1; S_2; S_3$ is ambiguous, but harmless as turns out later, and we may use parentheses around statements to disambiguate $(S_1; S_2); S_3$ and $S_1; (S_2; S_3)$. We also consider a statement context, denoted $S[\ ]$, which is a statement with exactly one hole in the place of a statement. The hole is denoted by . Then $S[S_1]$ is a statement in which S_1 is plugged into the hole of the statement context.

The notion of accessible and changed variables can be lifted to statements S. We write $S(x_1; \ldots; x_n; y_1; \ldots; y_m)$ to mean that the accessible program variables of S are $x_1; \ldots; x_n$ and the changed program variables of S are $y_1; \ldots; y_m$. Sometimes it is easier to work with the nite sets of accessible and changed variables, denoted by access(S) and change(S), respectively. The accessible and changed variables of a statement are an over-approximation, in the sense that these are the possible accessed and changed variables. The set of program variables occurring in S is denoted var(S) and is the union of the accessible and changed variables.

De nition B.1.6 (Accessible and changed variables) The accessible and changed program variables of a statement S is de ned inductively on the structure of S:

- access(O) = $\{x_1; \ldots; x_n\}$ and change(O) = $\{y_1; \ldots; y_m\}$ given that the operation O is associated to the polarized variables $\{x_1; \ldots; x_n; \overline{y_1}; \ldots; \overline{y_m}\}$,
- access(skip) = change(skip) = ;,
- access(halt) = change(halt) = ;,
- access($S_1; S_2$) = access(S_1) [access(S_2),

- $\mathrm{change}(S_1; S_2) = \mathrm{change}(S_1) \cup \mathrm{change}(S_2)$,
- $\mathrm{access}(\text{if } T \text{ then } S_1 \text{ else } S_2) = \mathrm{access}(S_1) \cup \mathrm{access}(S_2) \cup \{x_1, \ldots, x_n\}$ given $T(x_1, \ldots, x_n)$,
- $\mathrm{change}(\text{if } T \text{ then } S_1 \text{ else } S_2) = \mathrm{change}(S_1) \cup \mathrm{change}(S_2)$,
- $\mathrm{access}(\text{while } T \text{ do } S \text{ od}) = \mathrm{access}(S) \cup \{x_1, \ldots, x_n\}$ given $T(x_1, \ldots, x_n)$,
- $\mathrm{change}(\text{while } T \text{ do } S \text{ od}) = \mathrm{change}(S)$.

Note that the definitions of access and change do not depend on each other. The accessible program variables can be approximated more precisely by stating

$$\mathrm{access}(S_1; S_2) = \mathrm{access}(S_1) \cup (\mathrm{access}(S_2) \cap \mathrm{change}(S_1))$$

where now the changed program variables of S_1 act as `binders' with respect to the statement S_2. In fact,

$$\begin{aligned}
&\mathrm{access}(S_1; (S_2; S_3)) \\
&= \mathrm{access}(S_1) \cup (\mathrm{access}(S_2; S_3) \cap \mathrm{change}(S_1)) \\
&= \mathrm{access}(S_1) \cup ((\mathrm{access}(S_2) \cup (\mathrm{access}(S_3) \cap \mathrm{change}(S_2))) \cap \mathrm{change}(S_1)) \\
&= \mathrm{access}(S_1) \cup (\mathrm{access}(S_2) \cap \mathrm{change}(S_1)) \cup (\mathrm{access}(S_3) \cap \mathrm{change}(S_1; S_2)) \\
&= \mathrm{access}(S_1; S_2) \cup (\mathrm{access}(S_3) \cap \mathrm{change}(S_1; S_2)) \\
&= \mathrm{access}((S_1; S_2); S_3)
\end{aligned}$$

However, this may complicate the proofs involving accessible and changed variables later on. Hence, we opt for the simpler definition given above, which is a cruder approximation of the accessible variables.

We could define complex tests by the following abstract grammar:

$$B; B_1; B_2 ::= T \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2$$

where negation binds strongest, conjunction binds more strongly than disjunction, and the other ambiguities are harmless. We can then introduce abbreviations for statements:

$$\begin{aligned}
\text{if } \neg B \text{ then } S_1 \text{ else } S_2 &:= \text{if } B \text{ then } S_2 \text{ else } S_1 \\
\text{if } B_1 \wedge B_2 \text{ then } S_1 \text{ else } S_2 &:= \text{if } B_1 \text{ then if } B_2 \text{ then } S_1 \text{ else } S_2 \text{ else } S_2 \\
\text{if } B_1 \vee B_2 \text{ then } S_1 \text{ else } S_2 &:= \text{if } B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_1 \text{ else } S_2 \\
\text{if } B \text{ then } S_1 &:= \text{if } B \text{ then } S_1 \text{ else skip}
\end{aligned}$$

In the case of a first-order program signature, where tests are quantifier-free formulas, this may lead to an ambiguous interpretation of tests. However, it turns out that the semantics we give later on assigns the same meaning to both readings, so this ambiguity is harmless.

We can introduce assertions by the following abbreviation:

$$\text{assert}(B) := \text{if } B \text{ then skip else halt} \quad :$$

B.2 Operational semantics

The first approach of giving semantics to programs is that of operational semantics In our operational semantics, we introduce machine models that consists of a state space and an operationalization of the primitive operations and tests. The semantics of a program is understood as a sequence of steps, which are taken as the processor instructs the machine to perform primitive operations or tests. We here abstract from the particular machine by the means of a machine model, hence our operational semantics is too an abstraction of actual processor behavior.

This approach of abstractly giving semantics to programs is similar to giving semantics to formulas, in which we have introduced structures that consists of a domain of values and an interpretation of the non-logical symbols. A program denotes behavior relative to a given machine model and initial state, similar to how a formula denotes a truth value relative to a given structure and valuation.

Machine models can, for example, be used to show that program transformations preserve the semantics of programs. Low-level programming languages are used to instruct hardware to perform operations and tests, whereas high-level programming languages abstract away from intricate or irrelevant low-level details (such as ordering or encoding details). These programming languages have different program signatures and machine models. A compiler transforms high-level programs into low-level programs, and the preservation of behavior of the respective programs can be shown by relating machine models (e.g. the behavior relative to one machine model can be simulated by behavior of another machine model).

The syntactical structure by which statements are formed is chosen in such way that it is suitable for giving semantics to programs, which is based on the execution behavior of the statements of a program. This allows for structured programming, or control-structured programming, where it is possible to recognize from the syntax of a program what are the so-called control points for which it is possible to reason about the possible states of the underlying machine model. Making the state of the underlying machine model predictable is an important property of the semantics, which ensures that one is able to reason about program correctness.

The set of control points of a program can be understood as follows. One may consider a program to be the top-level statement containing sub-statements. Any position before or after a sub-statement of a program is a control point. The semantics of programs is given in terms of a current control point, which moves around the program as statements are executed, one after the other. The intuitive idea of control points are formalized by the continuation of a statement, that is, what remains to be executed after a small-step in the execution of a statement is taken. The continuation of a statement is either another statement, or the termination marker X.

It is important to realize that the models we consider are abstractions of the primitive operations and tests that can be performed on actual machines that execute programs. Based on a machine model, we can define a state transition system that abstractly models the behavior of programs. The accuracy of the analysis of program behavior thus relies on the precision of the chosen machine model.

Therefore, it is a design decision how much we are concerned about the possibility of blocking, non-determinism, and failure. The execution of primitive operations may result in a blocking (e.g. the machine hangs and cannot progress), in an un(der)speci ed next state (e.g. the next state is not fully determined by the previous state and the operation performed), or in an explicit failure (e.g. the machine signals an error). We introduce failure-sensitive machine models which take these possibilities into account.

De nition B.2.1 (Failure-sensitive machine model) A failure-sensitive machine model M is a pair of a state space S (a set of states), and an operationalization consisting of:

- for each operation O, a transition function which is a partial function O^M of states to a set of states,
- for each test T, a set of states T^M.

Given input state s and operation O, we write $s' \in O^M(s)$ if $O^M(s)$ is de ned and s' is in the set $O^M(s)$. In that case, s' is an output state. We write $O^M(s) = \{\}$ if $O^M(s)$ is de ned and is empty. We write $O^M(s) = \text{fail}$ if $O^M(s)$ is unde ned.

For a complex test B, we also have the induced set of states B^M as follows: T^M as base case:, $\neg B^M$ is the complement of B^M, $(B_1 \wedge B_2)^M$ is the intersection of B_1^M and B_2^M, and $(B_1 \vee B_2)^M$ is the union of B_1^M and B_2^M.

One may picture a failure-sensitive machine model by means of a graph, in which the states are vertices and directed edges, labeled by a primitive operation, represent transitions from one state to another. The graph also may have loose ends, which are outgoing edges from one state that leads to no state at all. The transition function of a machine model determines the edges (possibly with a loose end) that are present in the graph. We can then picture the following properties of the transition function O^M of a machine model M:

- If $O^M(s) = \{\}$, then the state s is an indeterminate state or a blocked state (with respect to operation O), that is, the next state is implicitly unde ned. This may be picture by having no outgoing edges from the state s. We may think of hanging at the indeterminate or blocked state, being unable to go to the next state. Indeterminacy of the machine model M means that there exists an indeterminate state, and a machine model M is progressive if there are no blocked states (so it is impossible to hang by performing an operation).
- If $O^M(s) = \{s'\}$ for some s', then the state s is a deterministic state (with respect to operation O), that is, there is exactly one outgoing edge from s into the next state s'. If every state of machine model M is deterministic, then M is also called deterministic.
- If $s' \in O^M(s)$ and $s'' \in O^M(s)$ for di erent s' and s'', then the state s is a non-deterministic state (with respect to operation O), that is, there are multiple outgoing edges from s. We may think of the next state of s to be arbitrary selected from the non-empty set $O^M(s)$. The machine model M is called non-deterministic if there is a non-deterministic state.

- If $O^M(s) = \mathit{fail}$, then the state s is a *failing state* (with respect to operation O), that is, the next state is explicitly undefined. This may be pictured by a loose end in the graph. A machine model which has a failing state is called *failing*, and a machine model without any failing state is called *non-failing*.

To avoid confusion, we do not speak of the `determinacy' of a machine model $M$. Some authors use `determinacy' to mean deterministic. However, `determinacy' may also mean the lack of indeterminacy of a machine model, i.e. that performing an operation in every state leads to some next state or is a failing state.

Now consider the processor, which consists of a programmable controller and a machine which is being controlled. The controller takes a program that specifies what operations the machine must perform, and what tests of the machine influence the control flow. After the controller reaches the end of the program, the processor terminates. Given a failure-sensitive machine model, we can also model the behavior of programs as being executed step-by-step by an *abstract processor*. The processor is abstractly modeled using configurations and transitions between configurations.

Definition B.2.2 (Configuration)**.** Given a failure-sensitive machine model M. A configuration is a pair of continuation and a state of M, or a failure signal fail.

Given a statement S and state s, we thus have that $(S; s)$ is a configuration. The configuration $(X; s)$ is called a *terminal configuration*, and fail is called the *failure configuration*.

There are different approaches for modeling processor behavior, from fine-grained to coarse-grained. In the fine-grained approach we transition from configuration to configuration in small steps, and this approach is also called *small-step operational semantics* where the intermediate configuration between initial and final configuration are taken into account. In the coarse-grained approach we transition from initial configuration directly to (one possible) final configuration. There, the initial configuration is not related to any intermediary configurations.

The small-step semantics is closer to the behavior of an actual processor, whereas the big-step semantics is easier to reason about and allows us to show important closure properties of the semantics, such as compositionality. We take both approaches, and, in fact, both approaches are equivalent in a certain sense.

Definition B.2.3 (Small-step operational semantics)**.** Given a machine model M. We define the binary relation $\to$ on configurations as the smallest relation satisfying the following conditions:

$$
\begin{aligned}
(O; s) &\to (X; s') \text{ if } s' \in O^M(s) \\
(O; s) &\to \mathit{fail} \text{ if } O^M(s) = \mathit{fail} \\
(\mathit{skip}; s) &\to (X; s) \\
(S_1; S_2; s) &\to (S_1'; S_2; s') \text{ if } (S_1; s) \to (S_1'; s') \\
(S_1; S_2; s) &\to (S_2; s') \text{ if } (S_1; s) \to (X; s') \\
(S_1; S_2; s) &\to \mathit{fail} \text{ if } (S_1; s) \to \mathit{fail}
\end{aligned}
$$

$$(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; s) \to (S_1; s) \text{ if } s \in B^M$$
$$(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; s) \to (S_2; s) \text{ if } s \notin B^M$$
$$(\textbf{while } B \textbf{ do } S \textbf{ od}; s) \to (S; \textbf{while } B \textbf{ do } S \textbf{ od}; s) \text{ if } s \in B^M$$
$$(\textbf{while } B \textbf{ do } S \textbf{ od}; s) \to (X; s) \text{ if } s \notin B^M$$

where s is a state and $S_1; S_2; S$ are statements.

We write $(S; s) \not\to$ if there is no configuration C such that $(S; s) \to C$. We have $(\textbf{halt}; s) \not\to$.

Proposition B.2.4. If $(S_1; s) \not\to$ then $(S_1; S_2; s) \not\to$.

Our intuition is that the small-step relation operates on a single statement at a time. The sub-statement S' of S on which we operate, is called the primed statement of S. For each statement S, we also have a primed context $S[\ ]'$. It is the case that $S = S[S']'$, meaning that the statement S is obtained from its primed context $S[\ ]'$ for which in the hole the primed statement S' is plugged. The primed statement of S is any statement that is not a sequential composition. For any statement that is not sequential composition, the primed statement is that statement itself and the primed context is just a single hole $[\ ]$. For the sequential composition $S_1; S_2$, we have that the primed statement of $S_1; S_2$ is the primed statement of S_1 and the primed context is $S_1[\ ]'; S_2$ where $S_1[\ ]'$ is the primed context of S_1.

Proposition B.2.5. Let S' be the primed statement of S. We have the following:

- $(S; s) \to (S[S''[\ ]']'; s')$ if $(S'; s) \to (S''; s')$,
- $(S; s) \to \textbf{fail}$ if $(S'; s) \to \textbf{fail}$,
- $(S; s) \not\to$ if $(S'; s) \not\to$.

Proof. We have $S = S[S']'$. By structural induction on S. If $S[\ ]'$ is just a hole, the result follows immediately. Otherwise, we apply the small-step semantics for sequential composition and the induction hypothesis. □

After execution of the primed statement S' of S is finished execution continues with the remainder of statement S, which we denote by $R(S)$. The remainder of a statement is a continuation, to take into account the possibility that the remainder is not a statement. For any statement that is not sequential composition, the remainder is X. For the sequential composition $S_1; S_2$, there are two cases. The remainder of $S_1; S_2$ is S_2 if the remainder of S_1 is X. The remainder of $S_1; S_2$ is $S_1'; S_2$ if the remainder of S_1 is S_1'.

Proposition B.2.6. Let S' be the primed statement of S. $(S; s) \to (R(S); s')$ if $(S'; s) \to (X; s')$.

Proof. Again by structural induction on S: all cases except sequential composition are trivial. For the remaining case where $S = S_1; S_2$, we distinguish the two cases whether $R(S_1) = X$ or not, and apply the relevant small-step semantics for sequential composition. □

By the above proposition, we now have established that the small step semantics either takes a step directly at top-level for any statement that is not a sequential composition, or takes a step at the primed statement. Since the primed statement is never a sequential composition, the step taken at the primed statement is a top-level step as well.

In fact, we can say something stronger about the small-step semantics defined above.

Proposition B.2.7 (Determinism). Given a deterministic machine model M, then the small-step operational semantics satisfies the following property:

$$\text{if } C_1 \to C_2 \text{ and } C_1 \to C_3 \text{ then } C_2 = C_3.$$

The above definition of the small-step operation semantics defines a relation between two configurations. We can imagine a chain of configurations as related by the small-step relation $\to$:

$$C_1 \to C_2 \to \cdots \to C_n \to \cdots$$

Definition B.2.8. An execution is a chain of configurations related by $\to$, which is either finite or infinite.

A complete execution is a finite chain with no further step possible, that is, there is no C_{n+1} such that $C_n \to C_{n+1}$. A complete execution is necessarily finite. A complete execution leading to termination is a complete execution that ends in some terminal configuration (X, s). A complete execution leading to failure is a complete execution that ends in the failure configuration fail. A complete execution leading nowhere is a complete execution that is neither leading to termination, nor leading to failure. In particular a configuration $C = (S, s)$, where s is an indeterminate state with respect to O and O is a primed statement of S, is called an *indeterminate configuration*. An complete execution that ends in an indeterminate configuration leads to nowhere. Executions leading nowhere are also called stuck. A diverging execution is an execution with an infinite chain of configurations.

For a given execution, the first configuration is called the *initial configuration*. Conversely, an initial configuration I induces the set of complete or diverging executions that start in the configuration I. For a complete execution, the last configuration is called the *final configuration*. We may also speak of a reachable configuration C from an initial configuration I if there exists an execution that starts in configuration I which contains the configuration C in its chain. A configuration is unreachable from an initial configuration if it is not reachable.

We may imagine the set of all executions, of a given machine model, to form a forest: the roots are initial configurations which induce (possibly infinite) trees

formed by executions that share the same initial con guration, and for which shared pre xes of two di erent executions forms the trunk of a tree. The trunk splits in two or more branches by taking a small-step of an operation on a state that is non-deterministic. There are three kinds of leaves. First, a leave represents that a small-step of an operation is performed on a blocking state: there is no next con guration. Second, a leave represents the failure con guration $fail$. Third, the leave represents a terminal con guration (X, s). The height of a con guration in a tree represents the number of small steps taken from the initial con guration (and as such there are no cycles back to earlier con gurations), and this height is often used to inductively reason about properties of executions.

The concepts of two executions reaching the same con guration is important enough that it warrants its own de nition.

De nition B.2.9 (Computations). Given an initial con guration I and a con guration C, the set of executions starting in I and reaching C is a computation from I to C.

A computation consists of zero or more executions, all reaching the same intermediary con guration. A computation thus abstracts from the particular way of reaching this con guration. A computation for which in the reached con guration C there is still a further step possible may also consists of diverging executions. If for the reached con guration C no further step is possible, we call the computation *complete*. A complete computation necessarily consists of complete executions. A complete computation leading to termination is a complete computation that ends in some terminal con guration (X, s). A complete computation leading to failure is a complete computation that ends in the failure con guration $fail$. A complete computation leading nowhereis a complete computation that is neither leading to termination, nor leading to failure.

There may be di erent computations starting from the initial con guration. In fact, an initial con guration I also induces a set of computations that start in the con guration I. It may be the case that for the same initial con guration I, there is a complete computation but also a computation which consists of diverging executions.

We now introduce the big-step semantics, which captures the notion of complete computations leading to termination or failure directly.

De nition B.2.10 (Big-step operational semantics) Given a machine model M. We de ne the binary relation $\Rightarrow$ on con gurations as the smallest relation satisfying the following conditions:

$$
\begin{array}{rll}
(O, s) & \Rightarrow (X, s') & \text{if } s' \in O^M(s) \\
(O, s) & \Rightarrow fail & \text{if } O^M(s) = \emptyset \\
(skip ; s) & \Rightarrow (X, s) & \\
(S_1 ; S_2, s) & \Rightarrow (X, s'') & \text{if } (S_1, s) \Rightarrow (X, s') \text{ and } (S_2, s') \Rightarrow (X, s'') \\
(S_1 ; S_2, s) & \Rightarrow fail & \text{if } (S_1, s) \Rightarrow fail \\
(S_1 ; S_2, s) & \Rightarrow fail & \text{if } (S_1, s) \Rightarrow (X, s') \text{ and } (S_2, s') \Rightarrow fail
\end{array}
$$

(if B then S_1 else S_2 ; s) C if (S_1; s) C and s 2 B^M

(if B then S_1 else S_2 ; s) C if (S_2; s) C and s 62 B^M

(while B do S od; s) C if (S; while B do S od; s) C and s 2 B^M

(while B do S od; s) (X; s) if s 62 T^M

In the big-step semantics we can no longer distinguish between a diverging execution or an execution leading to nowhere. In both cases we have that, for an initial con guration I, there is no nal con guration C such that I C. It is possible to observe this di erence from the small-step semantics: for the initial con guration there is either a diverging execution, in the rst case, or a complete execution I ! ::: ! C that is stuck in C, when there is no step possible from C, in the second case. However, this distinction disappears if we only consider observing nal con gurations of either the form (X; s) or fail.

In fact, there is a correspondence between big-step and small-step semantics. Let ! $^+$ be the transitive closure of the binary relation !. And let I ! ^{n} C denote that C can be reached in exactly n small steps from I. We then have that I ! $^+$ C holds if and only if there exists $n > 0$ such that I ! ^{n} C.

Proposition B.2.11. The following holds:

1. (S_1; S_2; s) ! $^+$ (S_1'; S_2; s') if (S_1; s) ! $^+$ (S_1'; s'),
2. (S_1; S_2; s) ! $^+$ (S_2; s') if (S_1; s) ! $^+$ (X; s'),
3. (S_1; S_2; s) ! $^+$ (X; s'') if (S_1; s) ! $^+$ (X; s') and (S_2; s') ! $^+$ (X; s''),
4. (S_1; S_2; s) ! $^+$ fail if (S_1; s) ! $^+$ fail.

Proposition B.2.12. C_1 C_3 if C_1 ! C_2 and C_2 C_3.

Lemma B.2.13 (Correspondence small-step and big-step semantics)
I ! $^+$ C if and only if I C for any terminal or failure con guration C.

Proof. I cannot be of the form (X; s) or fail since both the small-step and big-step operational semantics do not have terminal or failure con gurations on the left-hand side, while C must be of that form. So, we have that I = (S; s) for some S; s, and C = (X; s') for some s' or C = fail. ((=) By induction on the way (S; s) (X; s') is established, transitivity of ! $^+$, and Proposition B.2.11. (=)) Assume (S; s) ! ^{n} (X; s') for some $n > 0$, we proceed by induction on n, and use Proposition B.2.12. □

Corollary B.2.14. Given a deterministic machine model M, then the big-step operational semantics satis es the following property:

if C_1 C_2 and C_1 C_3 then $C_2 = C_3$:

Observe that the nal con gurations obtained by the big-step semantics are either (X; s) or fail. Given an initial con guration, we have a result set of possible

nal con gurations related to that initial con guration. The result set is either empty (divergence or stuck), or it contains the failure con guration fail alongside terminal con gurations of the form $(X; s)$ for some state $s \in S$. Note that, due to possible non-deterministic operationalizations of the machine model, the result set may contain at the same time di erent terminal con gurations and the failure con guration. Each con guration in the result set is obtained by a di erent computation.

Instead of considering only computations that begin in an initial con guration with a xed state, it is useful to abstract from the initial state. Thus, each statement S induces a result set that is indexed by an initial state $s \in S$, i.e. $\{C \mid (S; s) \to C\}_{s \in S}$. We can lift this construction to a set of initial states $X \subseteq S$, i.e. $\{C \mid (S; s) \to C \text{ for some } s \in X\}_{X \subseteq S}$.

We are interested in composing multiple results sets: given that a result set determines the possible outcomes of parts of a statement, can we compose result sets into the result set of the overall statement? To do so, we represent result sets more abstractly as an element of $\mathcal{P}(S + \{\text{fail}\})$, viz. as a subset of the disjoint union of states and a failure marker. We could see the failure marker fail as an improper state, and every state $s \in S$ as a proper state. Every statement induces a result set indexed by a (proper or improper) state. Equivalently, a statement induces a function $S + \{\text{fail}\}$ to $\mathcal{P}(S + \{\text{fail}\})$, which we denote by $\mathcal{M}[\![S]\!]$, with the following speci cation:

$$\mathcal{M}[\![S]\!](\text{fail}) = \{\text{fail}\}, \quad \mathcal{M}[\![S]\!](s) = \{s' \mid (S; s) \to (X; s')\} \cup \{\text{fail} \mid (S; s) \to \text{fail}\}$$

and we can lift this function to result sets, being a set of (proper or improper) states $Y \subseteq S + \{\text{fail}\}$, with the following speci cation:

$$\mathcal{M}[\![S]\!](Y) = \bigcup_{y \in Y} \mathcal{M}[\![S]\!](y)$$

where y is either the improper state fail or a proper state in S. We thus obtain our desired form: every statement induces a result set indexed by a result set.

B.3 Denotational semantics

We now see a second approach of giving semantics to statements, called *denotational semantics*. We can use functions on result sets as the *domain of denotation* of statements, and we work towards characterizing our denotation of statements in a syntax-directed manner. This allows for equational-style reasoning about the semantics of statements.

We rst introduce an approximate denotation of a statement, and then de ne the denotation of a statement as the limit of the approximate denotation.

De nition B.3.1 (Denotational semantics). Given a failure-sensitive machine model $\mathcal{M}$ and statement S. The approximate denotation of a statement $\mathcal{M}\llbracket S \rrbracket^n$ is a function on result sets, de ned inductively on n and structurally on S as follows:

- $\hat{M}\llbracket O \rrbracket^n(Y) = (Y \setminus \{fail\}) \cup \bigcup \{f\, O^M(s) \mid s \in (Y \setminus S)\}$,
- $\hat{M}\llbracket skip \rrbracket^n = id$,
- $\hat{M}\llbracket halt \rrbracket^n(Y) = Y \setminus \{fail\}$,
- $\hat{M}\llbracket S_1; S_2 \rrbracket^n = \hat{M}\llbracket S_2 \rrbracket^n \circ \hat{M}\llbracket S_1 \rrbracket^n$,
- $\hat{M}\llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket^n(Y) = (Y \setminus \{fail\}) \cup \hat{M}\llbracket S_1 \rrbracket^n(Y \cap B^M) \cup \hat{M}\llbracket S_1 \rrbracket^n(Y \cap (\neg B)^M)$,
- $\hat{M}\llbracket \text{while } B \text{ do } S \text{ od} \rrbracket^0(Y) = Y \setminus \{fail\}$,
- $\hat{M}\llbracket \text{while } B \text{ do } S \text{ od} \rrbracket^{n+1} = \hat{M}\llbracket \text{if } B \text{ then } S; (\text{while } B \text{ do } S \text{ od}) \rrbracket^n$,

where $Y \cup S \cup \{fail\}$ is a result set, and $Y \cap X$ is the intersection of Y and a set of proper states X but which propagates failure, so $fail \in (Y \cap X)$ if $fail \in Y$. We define the denotation of a statement as the limit of the approximate denotation:

$$\hat{M}\llbracket S \rrbracket(Y) = \bigcup_{n=0}^{\infty} \hat{M}\llbracket S \rrbracket^n(Y).$$

We may also write $\hat{M}\llbracket S \rrbracket(s)$ to mean $\hat{M}\llbracket S \rrbracket(\{s\})$ for a singleton proper state s. One may think of the parameter n as the maximal number of loop iterations for the outer while -statements, where the parameter decreases for statements that are directly nested under a while -statement. By using this parameter we ensure that the approximate denotational semantics is well-defined.

It is easy to see that the first three clauses of the approximate denotation also hold for the limit of the approximate denotation, that is:

- $\hat{M}\llbracket O \rrbracket(Y) = (Y \setminus \{fail\}) \cup \bigcup \{f\, O^M(s) \mid s \in (Y \setminus S)\}$,
- $\hat{M}\llbracket skip \rrbracket = id$,
- $\hat{M}\llbracket halt \rrbracket(Y) = Y \setminus \{fail\}$.

However, to establish that the other clauses also hold for the complex statements, we need to introduce further technical intermediary results. This is important to do, since it allows for syntax-directed reasoning about the limit of the approximate denotation

The approximate denotation given above has important properties, namely:

Lemma B.3.2 (Monotonicity) .

1. Given result sets $X \subseteq Y$, $\hat{M}\llbracket S \rrbracket^n(X) \subseteq \hat{M}\llbracket S \rrbracket^n(Y)$ if $X \subseteq Y$.
2. Given a result set Y, $\hat{M}\llbracket S \rrbracket^n(Y) \subseteq \hat{M}\llbracket S \rrbracket^{n+1}(Y)$.
3. Given a result set Y and $n \geq m$, $\hat{M}\llbracket S \rrbracket^n(Y) \subseteq \hat{M}\llbracket S \rrbracket^m(Y)$.

4. $M\llbracket S_2 \rrbracket^n \; M\llbracket S_1 \rrbracket^m \; M\llbracket S_1; S_2 \rrbracket^{max(n;m)}$.

Proof. The third result depends on the second result, which depends on the first. The first two by induction on n and S, and the last by induction on m. The fourth result is by induction on n and S_2: the case analysis is delicate and non-trivial, and the other results are needed multiple times. □

It is easy to lift the first result, and have $M\llbracket S \rrbracket(X) \; M\llbracket S \rrbracket(Y)$ if $X \; Y$. The second and third results become trivial when we lift them.

Based on the properties of monotonicity we can prove the following properties of the (approximate) denotation. Recall that assert(B) abbreviates

$$\text{if } B \text{ then skip else halt} \quad :$$

We have a few intermediary technical properties, needed to establish later results.

Proposition B.3.3.

1. $M\llbracket S_1; (S_2; S_3) \rrbracket^n = M\llbracket (S_1; S_2); S_3 \rrbracket^n$.
2. $M\llbracket S \rrbracket^n = M\llbracket \text{skip}; S \rrbracket^n$.
3. fail $2 X$ implies fail $2 M\llbracket S \rrbracket^n(X)$.

These properties are easily lifted to the limit of the approximate denotation. We proceed to establish the following properties of the denotational semantics.

Lemma B.3.4.

$\hat{} \; M\llbracket S_1; S_2 \rrbracket = M\llbracket S_2 \rrbracket \; M\llbracket S_1 \rrbracket$

$\hat{} \; M\llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket = M\llbracket \text{assert}(B); S_1 \rrbracket [M\llbracket \text{assert}(\neg B); S_2 \rrbracket$

To allow for syntax-directed reasoning about while -statements, we make use of syntactic approximations. We have the following recursively defined syntactic approximation called loop unrolling:

$$(\text{while } B \text{ do } S \text{ od})^0 = \text{halt};$$
$$(\text{while } B \text{ do } S \text{ od})^{k+1} = \text{if } B \text{ then } S; (\text{while } B \text{ do } S \text{ od})^k \text{ else skip} \quad :$$

The following holds for the denotational semantics.

Lemma B.3.5 (Loop unrolling).

$$M\llbracket \text{while } B \text{ do } S \text{ od} \rrbracket = \bigsqcup_{k=0}^{1} M\llbracket (\text{while } B \text{ do } S \text{ od})^k \rrbracket$$

It is now easy to establish, for the denotational semantics, that we can replace statements by equivalent statements under any context. In other words, equivalent statements cannot be discriminated by any context. Let $S[\;]$ be any statement with a hole.

Corollary B.3.6 (Compositionality). $M\,[\![S[S_1]]\!] = M\,[\![S[S_2]]\!]$ if $M\,[\![S_1]\!] = M\,[\![S_2]\!]$

It is also possible to generalize compositionality to contexts with arbitrarily many holes, for which a similar property as above holds.

We can now establish the correspondence between the operational semantics and denotational semantics. This correspondence allows us to reason about the operational semantics of a program using the denotational semantics, and vice versa. Thus, both approaches in giving semantics coincide!

Lemma B.3.7 (Full abstraction). $M\,[\![S]\!] = M\,[S]$.

B.4 Axiomatic semantics

There is a third approach of giving semantics to statements, which is the *axiomatic semantics*. In this approach we express the behavior of programs in terms of program specifications. In contrast to the operational and denotational semantics which are relative to a given machine model and so to speak works `from inside out', the axiomatic semantics consists of a set of program specifications and works `from outside in'. In the axiomatic semantics we declare which expectations we have of the behavior of a program, without *a priori* knowing the inner workings of the primitive operations of a machine model.

The behavior of a program can be modeled by its *input/output behavior*. Operationally, one may see program behavior as a relation between input and output states, i.e. $t \in M\,[S](s)$ indicates that output state t is related to the (singleton) input state s and $\mathit{fail} \in M\,[S](s)$ indicates that the program leads to failure from the (singleton) input state s. Denotationally, we know that a statement S denotes a state transformer $M\,[\![S]\!]$, and when X is a set of (proper or improper) states that $M\,[\![S]\!](X)$ also is a set of (proper or improper) states. We have seen before that operational and denotation semantics coincide. However, this raises the question: what language can we use to describe states? We turn to that question later in this section.

First, we outline the usefulness of axiomatic semantics. Expected program behavior can be expressed by giving two descriptions: one of the set of input states, and one of the set of output states. Formally, the description of the input states is called a *precondition*, and the description of the output states is called a *postcondition*. We work directly with these descriptions by the use of *program specifications*. We introduce the notation $\{\ \}\,S\,\{\ \}$ for program specifications which consists of a precondition $\ $, the statement S, and the postcondition $\ $. Then, given a suitable interpretation $[\![\ ]\!]$ of the precondition and postcondition as sets of (proper or improper) states, we can interpret program specifications as follows (called the *partial correctness interpretation*):

$$M \models \{\ \}\,S\,\{\ \} \text{ if and only if } M\,[\![S]\!]([\![\ ]\!]) \subseteq [\![\ ]\!]$$

By $M \models \{\ \}\,S\,\{\ \}$ we then mean that the program specification $\{\ \}\,S\,\{\ \}$ is satisfied in the machine model M. Next, we consider sets of program specifications: these are called *program theories*.

Similar how theories in (first-order or higher-order) logic can be used to classify structures, we can use program theories to classify machine models. By giving a set of program specifications which we expect to be satisfied in a given machine model, we constrain the possible choices of machine models that are possible. Note that such constraints can also be expressed for complex statements, e.g. involving control structures such as loops!

Similar how each structure induces a (first-order or higher-order) theory, we also have that each machine model induces a program theory. Given a machine model M, then by $Th(M)$ we denote the set of program specifications that are satisfied in M. It is then also possible to compare machine models by their induced program theories.

Example B.4.1. Say, we work with formulas of first-order logic for the pre- and postconditions, and we want to introduce a new complex programming construct which introduces local program variables with the syntax:

$$\text{begin local } x := y;\ S \text{ end}$$

The intended meaning is that the variable x is local to the execution of the statement S: it has an initial value determined by y, but the original value of x is restored after execution of S ends. To describe the semantics of such a programming construct using the axiomatic approach would amount to saying that the program theory must be closed under the following rule:

$$\frac{\{\ \}\ x := y; S\ \{\ \}}{\{\ \}\ \text{begin local } x := y; S \text{ end}\ \{\ \}} \text{ where } x \notin FV(\ \ )$$

Thus we are able to declaratively specify properties of the semantics of this complex programming construct, without knowing how this construct decomposes into primitive operations nor saying anything about the underlying denotational or operational semantics. End of Example.

In the axiomatic approach of giving semantics, we intend to give semantics to programs in an abstract setting, without explicitly knowing the underlying (operational or denotational) semantics of the primitive operations. We thus need a language in which we can express the precondition and postcondition, to be able to describe the behavior of a program. In doing so, we have three desiderata of the language we choose:

1. The tests (and thus assertions) can be described by the language,
2. the language is expressive enough to describe the behavior of the primitive operations,
3. the language is closed under substitution, conjunction, and negation.

The first desideratum lets us speak of an assertion language. Although the second desideratum is necessary, often one is still free to choose an appropriate level of abstraction. Typically, one wishes to specify primitive operations in which the

(concrete) implementation details are hidden, i.e. at the level of abstraction of the programming language itself. The third desideratum naturally leads us to choose a logical language.

We take as assertion language the language of one of the logics that we have introduced earlier. In doing so, we restrict the class of machine models to ensure that the set of states and the tests are compatible with the chosen logic. In this section we introduce *logical machine models*, corresponding to classical first-order logic. In logical machine models we take valuations as states and the (denotation of) quantifier-free formulas which are the tests. As such, we fix a first-order program signature $FPS(\Sigma)$, for a given first-order signature Σ.

Although we focus in this section on classical first-order logic, nothing prevents us from considering other classes of machine models to be associated to different logics. In fact, in later sections we also introduce the class of machine models corresponding to separation logic. In principle, one can choose any logic and make suitable design choices to map the chosen logic to a class of machine models. For practical purposes, one makes design choices in such a way as to ensure that assertions are decidable (i.e. tests can be effectively evaluated in any state) and primitive operations are computable. This motivates our choice above to consider the quantifier-free formulas as tests. In practice, one can also restrict the first-order signature Σ to ensure only a subset of the signature of the logic can be used in tests. When doing so, one may speak of the *logical signature* and the program signature that is a subset of the logical signature. For technical simplicity, we shall speak of only one signature.

Before giving logical machine models, we introduce the concept of equal sets of valuations modulo a set of variables. Let X and Y be sets of valuations of some structure A, and $Z \subseteq V$ be a set of variables. Then by $X = Y \bmod Z$ we mean that the sets X and Y are in a correspondence such that for each valuation $\nu \in X$ that corresponds to a valuation $\nu' \in Y$ we have that $[V \setminus Z]\nu = [V \setminus Z]\nu'$. This notion is also defined when a function is applied on both sets that are in correspondence, and then takes the original correspondence. Further, by $X = f(X) \bmod Z$ we mean that for each valuation $\nu \in X$ and valuation $\nu' \in f(\nu)$ we have that $[V \setminus Z]\nu = [V \setminus Z]\nu'$. These notions can be lifted in the obvious way to result sets (being sets consisting of valuations or *fail*), or sets of proper or improper states (being the disjoint union of proper states and the *fail* marker).

Definition B.4.2. A *logical machine model* M is a pair of a structure A and an operationalization consisting of:

- for each operation O, a transition function which is a partial function O^M of valuations to a set of valuations of A,
- for every operation $x := y$, the transition function $(x := y)^M$ is defined by mapping ν to $[x := \nu(y)]$,
- for the transition function O^M we have the *change condition* that either $O^M(\nu) = \mathit{fail}$ or $\nu'[V_1 \setminus \mathit{change}(O)] = \nu[V_1 \setminus \mathit{change}(O)]$ for every $\nu' \in O^M(\nu)$,

- for the transition function O^M we have the access condition that states that $O^M(\sigma) = O^M(\sigma') \bmod var(O)$ for every σ, σ' for which $[access(O)]\sigma = [access(O)]\sigma'$ holds.

The operations $x := y$ have a fixed operationalization, namely by assigning the value of y to the program variable x. Further, the change and access conditions can be explained as follows. For every operation, we require that only the changed program variables are actually modified by the operation. We require that only the accessible program variables can have an influence on the outcome of an operation. In fact, both conditions imply that operations depend only on finitely many variables and can affect only finitely many variables.

We also write $\langle M, \mathcal{A}\rangle$ for a logical machine model to indicate its underlying structure $\mathcal{A}$. A logical machine model is a failure-sensitive machine model in the following sense: the state space of a logical machine model is the set of valuations of $\mathcal{A}$, and the given operationalization induces an operationalization for tests by associating every quantifier-free formula φ to the set $[\![\varphi]\!]^{CL}$ that denotes the valuations that satisfy φ in structure $\mathcal{A}$.

The access and change conditions can be lifted to statements S.

Lemma B.4.3 (Change Lemma). Given a set of proper states X,

$$X \subseteq \langle M, \mathcal{A}\rangle[\![S]\!](X) \bmod change(S).$$

Lemma B.4.4 (Access Lemma). Given two sets of proper states X, Y such that $X =_{(V \setminus access(S))} Y$, then

$$\langle M, \mathcal{A}\rangle[\![S]\!](X) =_{(V \setminus var(S))} \langle M, \mathcal{A}\rangle[\![S]\!](Y).$$

Intuitively, these express that a program only modifies the variables $change(S)$, and that the outcome of a program is only dependent on the variables $access(S)$.

We now formally define whether a program specification is satisfied in a logical machine model. Note that, contrary to our earlier discussion, there is a mismatch in the denotation of formulas and the sets of (proper or improper) states: formulas never denote the improper state $fail$. Thus we have a stronger interpretation for program specifications, called *strong partial correctness*, defined as such:

$$\langle M, \mathcal{A}\rangle \models^{HL} \{\varphi\}\, S\, \{\psi\} \text{ if and only if } \langle M, \mathcal{A}\rangle[\![S]\!]([\![\varphi]\!]^{CL}) \subseteq [\![\psi]\!]^{CL}.$$

Since $fail$ is never in $[\![\psi]\!]^{CL}$, this interpretation explicitly states that the machine never fails when executing program S starting from any state in $[\![\varphi]\!]^{CL}$. Note that the superscript HL (short for Hoare's Logic) is used to be able to distinguish this interpretation from the one introduced in the next chapter, but may be dropped if it is clear from context what interpretation is intended.

It is useful to restrict our attention to particular logical machine models, that are based on a structure that satisfies a particular theory. There are two levels at which we recognize theories: background theories and program theories. Let T be a set of first-order formulas, called a background theory. We write $\models^{HL}_T \{\varphi\}\, S\, \{\psi\}$ to mean $\langle M, \mathcal{A}\rangle \models^{HL} \{\varphi\}\, S\, \{\psi\}$ for every logical machine model $\langle M, \mathcal{A}\rangle$ such that

$A \models^{CL} T$. We then say that the program specification is valid relative to the given background theory. We write $\models^{HL} \{\ \}\ S\ \{\ \}$ if the program specification is valid in every logical machine model, regardless of background theory, and call it universally valid.

Let Γ be a set of program specifications, called a program theory. We write $\Gamma \models^{HL}_{T} \{\ \}\ S\ \{\ \}$ to mean $\langle M, A\rangle \models^{HL} \{\ \}\ S\ \{\ \}$ for every structure A such that $A \models^{CL} T$ and every logical machine model $\langle M, A\rangle$ such that $\langle M, A\rangle \models^{HL} \{\ ^0\}\ S^0\ \{\ ^0\}$ for each $\{\ ^0\}\ S^0\ \{\ ^0\} \in \Gamma$. We then say that the program specification $\{\ \}\ S\ \{\ \}$ is a semantic consequence of Γ with respect to the background theory T. The notion $\Gamma \models^{HL} \{\ \}\ S\ \{\ \}$, that $\{\ \}\ S\ \{\ \}$ is a semantic consequence of Γ, can be defined (regardless of the background theory) in a similar way.

It is sufficient to focus on the semantic consequence relation, regardless of the background theory, by the following argument. Observe that the program specification $\{\Gamma\}$ skip $\{\ \}$ is satisfied in a logical machine model $\langle M, A\rangle$ if and only if $\langle M, A\rangle \models \langle \text{skip}, K(A)\rangle \to K^{CL}(\ )$. $A[\![\ ]\!]K^{CL}$ if and only if $A[\![\ ]\!]K^{CL}$ is the set of all proper states if and only if $A \models^{CL}$. Hence, every formula Γ in the background theory can be represented by the program specification $\{\Gamma\}$ skip $\{\ \}$. Let T be a background theory, and T^0 be the corresponding set of program specifications in which we represent each formula $\in T$ as a program specification $\{\Gamma\}$ skip $\{\ \} \in T^0$. Then we have $\Gamma \models^{HL}_{T} \{\ \}\ S\ \{\ \}$ if and only if $[\ T^0 \models^{HL} \{\ \}\ S\ \{\ \}$. For notational convenience, we simply work with formulas instead of their representations as program specifications. Hence, we merge the notions of background theory and program theory, and simply speak of a theory, being a set of program specifications or formulas. Every theory has projections to its underlying background theory and program theory.

We introduce a proof system in which program specifications can be deduced. The purpose of using a proof system is that we can effectively check the deduction of program specifications. The proof system is set-up as a proof system with premises. Typically, we take the background theory as premises, from which we can derive program specifications. Our proof system is called Hoare's logic, or HL in short, in honor of C.A.R. Hoare (but, as mentioned in Section 1.4, the proof system given below is by K.R. Apt and F.S. de Boer).

Definition B.4.5. The proof system HL consists of:

- program specifications or formulas of first-order logic as objects,
- the smallest deduction relation $\vdash^{HL}$ satisfying the conditions:

  (skip) $\vdash^{HL} \{\ \}$ skip $\{\ \}$,

  (halt) $\vdash^{HL} \{\ \}$ halt $\{$false$\}$,

  (assign) $\vdash^{HL} \{\ [x := y]\}\ x := y\ \{\ \}$,

  (comp) $\{\ \}\ S_1\ \{\ \}; \{\ \}\ S_2\ \{\ \} \vdash^{HL} \{\ \}\ S_1; S_2\ \{\ \}$,

  (if) $\{\ \wedge\ \}\ S_1\ \{\ \}; \{\ \wedge\ \neg\ \}\ S_2\ \{\ \} \vdash^{HL} \{\ \}$ if then S_1 else S_2 $\{\ \}$,

  (while) $\{\ \wedge\ \}\ S\ \{\ \} \vdash^{HL} \{\ \}$ while do S od $\{\ \wedge\ \neg\ \}$,

(conseq) $(\ ' \Rightarrow\ );\ f\ g\ S\ f\ g;(\ \Rightarrow\ ') \vdash_{HL} f\ '\ g\ S\ f\ '\ g$,

(subst) $f\ g\ S\ f\ g \vdash_{HL} f\ [x := y]g\ S\ f\ [x := y]g$ for $x \notin var(S); y \notin change(S)$,

(invar) $f\ g\ S\ f\ g \vdash_{HL} f\ \wedge\ g\ S\ f\ \wedge\ g$ if $FV(\ ) \cap change(S) = ;$,

(9-intro) $f\ g\ S\ f\ g \vdash_{HL} f\exists x\ g\ S\ f\ g$ for $x \notin var(S)\ [\ FV(\ )$.

Note how only the consequence proof rule (conseq) uses formulas as premises. Every deduction is a proof tree constructed in the usual way. Hence, a deduction has only finitely many premises, being either formulas or program specifications.

Let $\ $ be a given theory. We can now formulate that the proof system satisfies the following meta-theoretical property, relating the proof system to the semantic consequence relation on program specifications.

Lemma B.4.6 (Soundness)

$$\vdash_{HL} f\ g\ S\ f\ g \text{ implies } \models_{HL} f\ g\ S\ f\ g.$$

Proof. Generalized reflexivity and generalized transitivity holds for the strong partial correctness interpretation too, by induction on the structure of the deduction. We then verify the axioms: For (skip), we have $\models f\ g$ skip $f\ g$ regardless of $\ $, and this easily follows from the denotational semantics. Also for (halt), we have $\models f\ g$ halt $f$false$g$ regardless of $\ $, similar to skip. For (assign), every logical machine model has a fixed interpretation of $x := y$, and thus we have the result by the substitution lemma. We have that (comp) follows directly from the denotational semantics. For the proof rule (if) we can perform a case distinction on whether $\ $ holds or not, and from $f\ \wedge\ g\ S\ f\ g$ we can obtain $f\ g$ assert$(\ ); S\ f\ g$ and similar for the other case. For the proof rule (while) we can do loop unrolling, and then observe that

$$\bigcup_{k=0}^{1} \langle M; A\rangle\langle(\text{while}\ \ \text{do}\ S\ \text{od})^k\rangle(A \rightarrow K^{CL}) \cap A \rightarrow\ \wedge : \ \subseteq K^{CL}$$

holds by considering that the following holds

$$\langle M; A\rangle\langle(\text{while}\ \ \text{do}\ S\ \text{od})^k\rangle(A \rightarrow K^{CL}) \cap A \subseteq K^{CL}$$

for every k; the latter can be shown by induction on k and the premise. For the proof rule (conseq) the result holds for logical machine models in which the premises are satisfied.

The remaining rules can be proven sound, given the following intuition:

- In the substitution rule (subst) we make use of the access lemma to take any computation from $f\ g\ S\ f\ g$ and change the initial state with respect to variable x that is not occurring in S to obtain another computation (the variable x can then not be overwritten by S). The value to assign to x is the value of y, which must have the same value in the initial and final state due to the change lemma. The specification then is satisfied by applying the substitution lemma on the initial and final state.

- The invariance rule (invar) follows from the change lemma, where the denotation of depends entirely on its free variables, which values cannot change.
- The 9-introduction rule (9-intro) follows from the access lemma, since the value of x cannot have any e ect on the computation of S nor determine the denotation of . □

In fact, under suitable assumptions of the expressivity of the assertion language, the converse can be stated as well. To do so, we introduce the notions of *weakest precondition* and *strongest postcondition*, relative to a given theory .

Remark B.4.7. Since we have limited the variables that are changed, and every program depends only on nitely many variables, it is possible to express the weakest precondition (strongest postcondition) by a formula. These conditions cannot be described by a (nite) formula if, for example, a primitive operation would a ect the value of in nitely many variables, or would depend on the value of in nitely many variables. End of Remark.

Given a program S and formula , let WP (S;) denote the weakest (liberal) precondition, a formula with the following properties:

- $\models^{HL}$ f WP (S;)g S f g,
- $\models^{HL}$ f g S f g implies $\models$! WP (S;),
- $\models^{HL}$ WP (skip ;) ! ,
- $\models^{HL}$ WP (halt ;) ! ? ,
- $\models^{HL}$ WP (x := y;) ! [x := y],
- $\models^{HL}$ WP (S_1; S_2;) ! WP (S_1; WP (S_2;)),
- $\models^{HL}$ WP (if then S_1 else S_2 ;) !
  (WP (S_1;) ^) _ (WP (S_2;) ^ :),
- $\models^{HL}$ WP (while do S od;) !
  (! WP (S; WP (while do S od;))) ^ (: !).

Note that in some of these conditions we use formulas , such that $\models$ means that for every logical machine model hM ; Ai that satis es the theory , we must have that the formula is valid, that is, A $\models$. Whether a formula exists, that can express the weakest precondition, is a property of the assertion language and the given theory: not all choices of signatures and theories allow us to express such weakest precondition as a formula.

There are some general properties that hold of the weakest precondition, as de ned above, that show that the weakest precondition is closely related to our denotational semantics:

Proposition B.4.8. $\models^{HL}$ WP (S_1; WP (S_2;)) ! WP (S_1; S_2;).

Proof. By de nition of the weakest precondition, we have

$$\models^{HL} \{WP(S_2, \Psi)\}\, S_2\, \{\Psi\}$$

and

$$\models^{HL} \{WP(S_1; WP(S_2, \Psi))\}\, S_1\, \{WP(S_2, \Psi)\}.$$

By the soundness of the composition rule, we thus have

$$\models^{HL} \{WP(S_1; WP(S_2, \Psi))\}\, S_1; S_2\, \{\Psi\}.$$

But from this, it follows that $\models^{HL} WP(S_1; WP(S_2, \Psi)) \rightarrow WP(S_1; S_2, \Psi)$ also from the de nition of weakest precondition. □

Other converses of the other properties of the weakest precondition given above can be shown too: this establishes that we deal with equivalence and not merely logical implications.

In fact, for a given logical machine model $\langle M, A\rangle$ we can precisely specify what the weakest precondition denotes. A logical machine model fixes the background theory $Th_1(A)$ by the choice of the underlying structure A, and furthermore fixes the program theory $Th(\langle M, A\rangle)$ by the operationalization of M. So we can take as theory $\Theta = Th_1(A) \cup Th(\langle M, A\rangle)$. In this case we can simply speak of $WP(S, \Psi)$, dropping the subscript and instead take the theory induced by the given model. Then, the weakest precondition is understood, semantically, to denote

$$\|WP(S, \Psi)\|^{CL}_{\sigma} = \{\sigma' \mid \langle M, A\rangle, \sigma \Rightarrow_S \sigma' \text{ implies } \sigma' \in \|\Psi\|^{CL}\}$$

where σ ranges over proper states (the valuations of A). Since our semantics denotes the empty set for diverging programs, and the empty set is always included in any set of proper states, we thus have a weakest liberal precondition in the sense that we do not care about diverging computations.[1] Note that in this setting, we are able to distinguish a primitive operation leading to failure from an indeterminate primitive operation: in the former case the weakest precondition is empty (since fail is never contained in any set of proper states), whereas in the latter case the weakest precondition is the set of all proper states (since the empty set is always contained in any set of proper states).

Next, let $NF(S)$ denote a formula expressing the precondition so that computations of S do not lead to failure, and let $SP(\Phi, S)$ denote a formula expressing the strongest postcondition, with the following properties:

- $\models^{HL} \{NF(S) \wedge \Phi\}\, S\, \{SP(\Phi, S)\}$,
- $\models^{HL} \{\Phi\}\, S\, \{\Psi\}$ implies $\models SP(\Phi, S) \rightarrow \Psi$.

Note the asymmetry between the weakest precondition and the strongest postcondition due to the failure-sensitive semantics: the strongest postcondition is only

[1] If liberal is not caring about getting stuck without making any progress, then progressive is caring about making progress. But `weakest progressive precondition' is terminology I invented.

given in the case the precondition excludes the possibility any computation leads to failure.

Again, for a given logical machine model $\langle M, A\rangle$ we can precisely specify what the non-failing formula and the strongest postcondition denotes. Since a particular logical machine model induces a particular theory, we simply write $NF(S)$ and $SP(\varphi, S)$, dropping the subscript. We can take

$$\llbracket NF(S) \rrbracket^{CL} = \{\sigma \mid \text{fail} \notin \langle M, A\rangle(S)(\sigma)\}$$

and note that, for deterministic machine models, $NF(S)$ and $WP(S, \text{true})$ denote the same set of proper states. We also can take

$$\llbracket SP(\varphi, S) \rrbracket^{CL} = \{\tau \mid \exists \sigma \in \llbracket \varphi \rrbracket^{CL} : \tau \in \langle M, A\rangle(S)(\sigma)\}$$

where we take all proper final states starting from a state in the given precondition. Note that, for a non-deterministic program S, we may have that a particular given σ possibly leads to failure but in a non-deterministic way. In that case, $NF(S)$ is empty, since the failure cannot be avoided. However, $SP(\sigma, S)$ then could still be non-empty for the computations that do not lead to failure, whereas $SP(\text{false}, S)$ is empty. Hence there is a difference between $SP(\sigma, S)$ and $SP(NF(S) \wedge \sigma, S)$.

We now show the relative completeness result, by using the weakest (liberal) precondition. This completeness result is called *relative* since we have two assumptions: we assume the expressivity of the weakest precondition, and we assume that every logical truth is contained in the theory. The latter assumption is quite strong and may go beyond what is computable or even recursively enumerable, and hence we refer to that latter assumption as if we have access to an *oracle*.

Lemma B.4.9 (Relative completeness) Given a theory Γ where the weakest liberal precondition $WP(S, \psi)$ is expressible and Γ is maximally consistent (with respect to the background theory), then

$$\Gamma \models^{HL} \{\varphi\}\, S\, \{\psi\} \text{ implies } \Gamma \vdash^{HL} \{\varphi\}\, S\, \{\psi\}$$

for all formulas φ, ψ.

Proof. We assume $\Gamma \models^{HL} \{\varphi\}\, S\, \{\psi\}$. The proof goes as follows: it suffices to show that $\Gamma \vdash^{HL} \{WP(S, \psi)\}\, S\, \{\psi\}$, since we obtain the desired result by an application of the consequence rule and the property that $\Gamma \models^{HL} \varphi \rightarrow WP(S, \psi)$. Since Γ is maximally consistent we can actually apply the consequence rule.

The proof is by induction on S. For primitive operations, the specification must follow from Γ (otherwise it contradicts our assumption). The skip, halt, and assignment operations follow from the properties of the weakest precondition.

For sequential composition, we apply the consequence rule (together with the property of the weakest precondition that distributes over composition) and need to show $\Gamma \vdash^{HL} \{WP(S_1, WP(S_2, \psi))\}\, S_1; S_2\, \{\psi\}$. This can be done by an application of the composition rule, with $WP(S_2, \psi)$ as intermediary formula, and the induction hypotheses. The other complex statements are similar. □

Summarizing, the completeness result depends essentially on the expressivity of the weakest precondition. That this is crucial boils down to the following observation: if we know that $\models^{HL} \{\}\ S_1; S_2\ \{\}$, how can we find a description of the possible intermediate states? The weakest precondition offers such a description. Similarly, the weakest precondition describes the loop invariant in the case of the while-statement. The fact that we need an oracle is secondary. In the case we deal with finite structures, for which the background theory is complete, we also have completeness of Hoare's logic. For example, in the case of 32-bit signed integers, the oracle can be effectively implemented by means of a decision procedure, and the question whether a program specification is deducible is decidable as well. However, there are background theories such as the theory of stacks, for which one can give concrete programs where the loop invariant is non-expressible [133]. In that case, having an oracle does not help in overcoming an inexpressive background theory.

From a proof-theoretic and model-theoretic point of view, the discussion of completeness becomes more interesting. We know that there is a sound and complete, finitary proof system for first-order logic. We can combine that proof system with the proof system for Hoare's logic: any proof needed in the consequence rule can be provided by a deduction in the proof system for first-order logic. The relative completeness result now no longer needs a background theory that is maximally consistent (the oracle), since by the completeness of first-order logic we already know that every semantic consequence (of the background theory) can be deduced. In this case, it is important to keep in mind that program theories are interpreted with a semantics of programs with respect to *arbitrary* structures that satisfies the background theory. This, in fact, further shows that expressivity of the weakest precondition is essential to the completeness result.

Nothing prevents us from taking one of the axiomatic set theories (such as Zermelo-Fraenkel set theory [49], Quine's New Foundations [85], Von Neumann Bernays Gödel set theory [153], among others) as a background theory. The resulting program logic is very expressive: we can specify very rich specifications of programs. However, in that case, the relation with practical computing becomes less clear, although even Dijkstra did not mind speaking about programs that work on sets or real numbers[1]. It is an interesting avenue to see what assumptions (such as encodability, recursive enumerability, and decidability) are needed for modeling the primitive operations and tests out of which a program is constructed. One assumption, for example, could be that the value of every accessible or changed program variable must have a digital encoding, so it can actually be represented in the memory of a classical digital computer. However, alternative assumptions may be needed for programs intended to be executed by quantum computers.

Note that we can also prove relative completeness using the strongest postcondition, but we leave that as an exercise for the reader. Also note that we did not need all proof rules in the (relative) completeness proof, but this changes after introducing recursively defined procedures with parameters. For while-programs it is in fact the case that the invariance rule, substitution rule, and 9-introduction rule are admissible. However, these rules are not derivable from the other rules.

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

B.5 Recursive procedures

This section sketches how to extend our approach to recursive procedures with parameters. The purpose is to demonstrate that the set-up of the axiomatic semantics above naturally extends to giving a proof system for programs with recursive procedures. We shall limit our formal development to outline the main ideas, and instead refer readers to the journal article *Completeness and Complexity of Reasoning about Call-by-Value in Hoare Logic* for all technical details [29]. However, the presentation given here is slightly more elegant than that of [29], due to the use of program signatures.

The axiomatic semantics above naturally leads to a programming methodology called *design by contract* [154, 18]. In essence, every operation of the program can be assigned a contract that declares a precondition (which the caller of the procedure needs to guarantee) and a postcondition (which the caller of the procedure may assume to hold after the procedure terminates). We follow this methodology in the design of a proof system that supports verifying programs with recursive procedures.

Given a program signature. A recursive program $(D \mid S)$ consists of a main statement S and a set of declarations D. A declaration declares the meaning of an operation of the program signature, by defining it in terms of a procedure body which is a statement of our language. Operations that lack such a declaration are so-called *native* operations. A native operation thus lacks a procedure body. The set D associates to each operation at most one declaration. Declarations are denoted as follows

$$O\langle x_1, \ldots, x_n, \overline{y_1}, \ldots, \overline{y_m}\rangle :: S.$$

The operation is annotated with a set of polarized variables, that indicate that execution of the operation may access variables $x_1, \ldots, x_n$ and may change variables $y_1, \ldots, y_m$. In the context of a set D, we call an operation O for which there is a declaration in D a procedure, and say it has procedure body S. Since the operations are already fixed by the program signature, they can occur in statements, including the main statement.

Intuitively, a procedure body should only access and change the variables as declared. A recursive program is *well-formed* if for all procedures O with body S, we have $\mathrm{access}(S) \subseteq \mathrm{access}(O)$ and $\mathrm{change}(S) \subseteq \mathrm{change}(O)$.

Example B.5.1. Given a program signature which has the operations Z, S, P, $+$, and the test $Z?$ that accesses variable x. The following procedure declarations D:

$$\begin{aligned}
Z\langle \overline{z}\rangle &:: \text{native}\\
S\langle x, \overline{z}\rangle &:: \text{native}\\
P\langle x, \overline{z}\rangle &:: \text{native}\\
+\langle x, y, z, w, \overline{x}, \overline{y}, \overline{z}, \overline{w}\rangle &:: \text{if } Z?(x) \text{ then } z := y \text{ else}\\
&\qquad P; w := z; x := y; S; y := z; x := w; +
\end{aligned}$$

and main statement + forms the recursive program $(D \mid +)$. The intended data structure is that of the natural numbers. The intuition is that Z resets the variable x to the value 0, S computes the successor of the value in x and stores that in z, P computes the predecessor of the value in x and stores it in z (and if x is 0 it does not terminate). We are then able to give the procedure body of + that computes the addition of the values in x and y and stores the result in z. End of Example.

For notational convenience, one may leave out the access and change variables in procedure declarations, since the smallest sets of polarized variables can be computed for a given set of declarations. Hence, we shall not write these variable annotations anymore.

Up to now all variables are *global*, in the sense that the same variable in every context refers to the same `storage location'. We extend the programming language with a block statement for introducing *local* variables, which allow us to temporarily change the value of a variable within the scope of the block. As such, the statements are extended to include the complex block statement

$$S ::= \ldots \mid \textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}$$

where $\vec{x}$ and $\vec{y}$ are sequences of variables of the same length, and $\vec{x}$ consists of unique variables. The variables of $\vec{x}$ are local variables within the scope of the block. We also extend the definitions of access and change as follows:

$$\begin{aligned} access(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}) &= (access(S) \setminus \vec{x}) \cup \vec{y}; \\ change(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}) &= change(S) \setminus \vec{x}. \end{aligned}$$

From the perspective of operational semantics, there is a problem with giving the block statement a small-step semantics. The small-step semantics is in a sense a `local' semantics, which transforms the statement and state one step at a time. However, the intended semantics is that after the block is exited, the values of the local variables have to be restored to their original value, that is, at the time before entering the block statement. A possible solution is to keep track of the original values in the state by means of a stack, to which values can be pushed, and from which original values can be popped. In that way, the block construct can be seen as a structured short-hand of a program in a block program signature, where the original value of each local variable is first pushed, then the parallel assignment is performed, and after the block ends the original values are popped again in reverse order.

For logical machine models, the big-step semantics of the block statement can be given directly, i.e. without pushing and popping, by the following transition:

$$\begin{aligned} &(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}; s) \quad (X; s'[\vec{x} := s(\vec{x})]) \text{ if } (S; s[\vec{x} := \vec{y}]) \quad (X; s') \\ &(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}; s) \quad \text{fail if } (S; s[\vec{x} := \vec{y}]) \quad \text{fail} \end{aligned}$$

where we have parallel update of, and parallel access from, proper states s (being valuations of the underlying structure). These are denoted $s[\vec{x} := \vec{v}]$ for the updated state where $\vec{v}$ is a sequence of new values (of the same length as $\vec{x}$), and $s(\vec{x})$ for a

sequence of values that the variables of $\vec{x}$ have in state s, respectively. Without much difficulty it is also possible to extend the denotational semantics in a similar way.

Having local variables and block statements allows us to introduce procedures with parameters. In the signature, for each operation O we also record its arity (being a set of variables), denoted $arity(O)$. Consequently, we extend the notion of declarations to display the arity of operations as follows:

$$O(x_1, \ldots, x_n) :: S,$$

where $x_1, \ldots, x_n$ are distinct variables, exactly covering $arity(O)$, called the formal parameters of the operation O. The difference between the arity of an operation, and the formal parameters of an operation is that in the latter the order of variables is significant. (Note that the signature still assigns polarized variables to each operation, indicating the potentially accessible and changed variables, but they are left implicit in declarations.) The formal parameters are local to the procedure body, and thus are never included in the variables that are potentially accessed or changed: the latter variables are global variables that are not among the formal parameters. As such, we require that $arity(O)$ and $access(O)$ are disjoint, as well as $arity(O)$ and $change(O)$.

Further, we have the following call statement:

$$S ::= \ldots \mid O(y_1, \ldots, y_n)$$

where $y_1, \ldots, y_n$ are the actual parameters supplied as part of the call, where we assume O has the arity $\vec{x}$, and $\vec{y}$ and $\vec{x}$ of equal length. It is permissible that the actual parameters $\vec{y}$ have duplicate variables, while this is not the case for the formal parameters $\vec{x}$. We speak of a *procedure call* in the context of a set of declarations if the corresponding operation is a procedure, and otherwise speak of a native call. We also extend the definitions of $access$ and $change$ as follows:

$$access(O(\vec{y})) = access(O) [\vec{y}/\vec{x}],$$
$$change(O(\vec{y})) = change(O)[\vec{y}/\vec{x}].$$

Intuitively, a procedure body should only access and change the variables as declared, but it is permissible to access and change the formal parameters. A recursive program is *well-formed* if for all procedures O with body S, we have $(access(S) \setminus arity(O)) \subseteq access(O)$ and $(change(S) \setminus arity(O)) \subseteq change(O)$.

In fact, procedures without parameters can be regarded as procedures with zero parameters, that have an empty arity. The notions defined with parameters thus are a refinement of the former notions without parameters.

To give semantics to recursive programs, we lift the semantics of statements to a semantics of (well-formed) recursive programs by also considering the set of declarations in each configuration. The big-step semantics of the procedure call then is defined by the transition

$$(D \mid O(\vec{y}); s) \quad C \quad \text{if } (D \mid \textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}; s) \quad C$$

where $O(\vec{x}) :: S$ is in the set of declarations D. We have that $\vec{x}$ and $\vec{y}$ match in length, because the actual parameters and arity of an operation match in length and also that the formal parameters gives an order of the variables of the arity. This style of operational semantics is also called *body replacement*, since intuitively we replace the procedure call by the body of the procedure wrapped in a block statement. We here restrict ourselves to the call-by-value parameter passing mechanism, meaning that the values of actual parameters are computed before starting to execute the procedure body. This is in contrast to call-by-name where we would have to perform substitution of formal parameters by actual parameters in the procedure body, resulting in a different replacement for each procedure call with different actual parameters, and requires handling variable capturing by block statements. Note that the behavior of a native call is left unspecified, and as such can be interpreted by the operationalization of a machine model.

The denotational semantics can be given iteratively. This is similar to how we previously gave a denotational semantics to *while* -statements. We leave the details out and instead refer the reader to [29]. What is more important concerning our discussion is the proof system for recursive programs.

Procedures return values by the use of a global variables (that can be changed). It is possible to designate a special variable, called *result* , that cannot occur as a formal parameter of any procedure declaration, and to which the procedure may assign an output value in the procedure body. For procedure declarations that have the *result* variable listed among its changed variables, we can introduce the following abbreviation:

$$x := O(\vec{y})$$

which denotes the statement

$$O(\vec{y}); x := \mathit{result}$$

Since we have introduced blocks that allow us to introduce local variables, we can also introduce the following syntactic sugar. An expression e is constructed out of either an individual variable, or a procedure call with as formal parameters other expressions (matching in length the number of formal parameters of the corresponding operation). For example, $O(P(x, y), z, Q(w))$ is an expression given operations O, P, Q and variables x, y, z, w. Each expression $O(e_1, \ldots, e_n)$ abbreviates a statement. If all expressions $e_1, \ldots, e_n$ are variables then $O(e_1, \ldots, e_n)$ is simply a procedure call with the corresponding variables as actual parameters. Otherwise, let e_i be the first non-variable expression from the left, then $O(e_1, \ldots, e_n)$ abbreviates the following statement:

$$e_i ; \mathbf{begin\ local}\ z_i := \mathit{result} ; O(e_1, \ldots, z_i, \ldots, e_n)\ \mathbf{end}$$

and the statement $x := O(e_1, \ldots, e_n)$ abbreviates

$$O(e_1, \ldots, e_n); x := \mathit{result}$$

where we take $z_1, \ldots, z_n$ to be fresh variables (i.e. not occurring in any of the sub-expressions, nor in accessible or changed variables of the operation). Note

that this e ectively evaluates the expressions from the left to the right, storing the result of each sub-expression in a local variable. Also note that the abbreviation is recursively de ned.

Note the resemblance between terms (in the assertion language) and expressions (in the programming language). Terms can be added to a logic without terms, by the introduction of an existential quanti er that binds the output value that are assigned to inputs by a functional relation. In that way, terms can be used in the place of variables. Similarly, we use local variables to capture the output of an expression so that, as well, expressions can be used in the place of variables. However, note that with expressions we also de ned a so-called*order of evaluation*, by evaluating the expressions from left to right. The reason to do so, and why this is not needed in the assertion language, is because expressions can have*side*-e ects on global variables, which consequently a ect the evaluation of subsequent expressions.

Also note that the programming language as described above employs*dynamic* scoping This means that a global variable can be captured by the use of a block statement, to isolate the e ects of procedures. We say that a program is*statically scoped*if the local variables and global variables are separated, thus disallowing capturing of global variables and ensuring that the referent of a global variable remains the same in every context. This may remind the reader of Barendregt's variable convention, in which bound and free variables are separate. See also the discussion of local variables by K.R. Apt and E.-R. Olderog [11, Section 5.2].

Finally, we consider extending the proof system HL with rules for proving properties about block statements and procedure calls. The objects of the proof system remain Hoare triples $\{\ \}\ S\ \{\ \}$ in which we are oblivious of the set of declarations in the proof system. We furthermore add the Hoare triples $\{\ \}\ D \mid S\ \{\ \}$ for specifying (well-formed) recursive programs $(D \mid S)$.

We have the following additional proof rules:

(block) $\{\ [\vec{x} := \vec{y}]\}\ S\ \{\ \} \vdash_{HL} \{\ \}$ begin local $\vec{x} := \vec{y}$; S end $\{\ \}$ if $FV(\ ) \setminus \vec{x} = \ ;$,

(inst) $\{\ \}\ O(\vec{x})\ \{\ \} \vdash_{HL} \{\ [\vec{x} := \vec{y}]\}\ O(\vec{y})\ \{\ \}$ if $FV(\ ) \setminus \vec{x} = \ ;$,

(rec) If $\ ; \ \vdash_{HL} \{\ _i\}\ S_i\ \{\ _i\}$ for every $1 \quad i \quad n$ and $\ ; \ \vdash_{HL} \{\ \}\ S\ \{\ \}$, then
$\vdash_{HL} \{\ \}\ D \mid S\ \{\ \}$ where $D = \{O_1(\vec{x}_1) :: S; \ldots; O_n(\vec{x}_n) :: S_n\}$
and $\ = \{\{\ _1\}\ O_1(\vec{x}_1)\ \{\ _1\}; \ldots; \{\ _n\}\ O_n(\vec{x}_n)\ \{\ _n\}\}$
and $FV(\ _i) \setminus \vec{x}_i = \ ;$ for every $1 \quad i \quad n$.

Recall that $[\vec{x} := \vec{y}]$ is the substitution of the variables $\vec{x}$ by the corresponding variables $\vec{y}$. The speci cations in are called contracts, and we require that the formal parameters of each procedure do not occur in the postcondition of the contract. The di erence between and is that consists of the program theory and background theory, used for axiomatizing the native operations and the underlying structure, whereas introduces contracts for the procedures which have a procedure body.

It is now possible to formulate both soundness and (relative) completeness for recursive programs too. The essence of the relative completeness proof is to introduce most general contracts for each procedure, making use of a strongest

postcondition axiomatization in the line of Gorelick [96]. We refer the reader to [29] for more details.

Remark B.5.2. These rules are admissible in Hoare's logic: every local block can be eliminated by introducing fresh variables (that do not occur in any surrounding program), and recursive procedures can be eliminated and reduced to simple while loops. However, showing why this is the case in detail is out of scope of this thesis.

Appendix C

Intuitionistic separation logic

C.1 Standard semantics

De nition C.1.1 (Satisfaction relation). f Given a structure $A = (A; I)$, a valuation of A, a nite heap h of A, and a separation logic formula . The satisfaction relation $A; h; \models^{SISL}$ is de ned inductively on the structure of :

- $A; h; \models^{SISL} ?$ never holds,
- $A; h; \models^{SISL} (x \doteq y)$ i $(x) = (y)$,
- $A; h; \models^{SISL} (x \mapsto y)$ i $h(\ (x))$ is de ned and $h(\ (x)) = (y)$,
- $A; h; \models^{SISL} C(x_1; : : : ; x_n)$ i $(\ (x_1); : : : ; \ (x_n)) \ 2 \ C^I$,
- $A; h; \models^{SISL} \ !$ i $A; h^0; \models^{SISL}$ implies $A; h^0; \models^{SISL}$ for every h^0 h,
- $A; h; \models^{SISL} 8x$ i $A; h; [x := a] \models^{SISL}$ for every $a \ 2 \ A$,
- $A; h; \models^{SISL}$ i $A; h_1; \models^{SISL}$ and $A; h_2; \models^{SISL}$ for some $h_1; h_2$ such that $h \quad h_1] h_2$,
- $A; h; \models^{SISL}$ i $A; h^0; \models^{SSL}$ implies $A; h^{00}; \models^{SISL}$ for every $h^0; h^{00}$ such that $h^{00} \quad h] h^0$.

This de nition is based on nite heaps. SISL stands for Standard Intuitionistic Separation Logic. It crucially di ers from SSL in the clause for logical implication. Note that the pure fragment of separation logic still is interpreted classically, since pure formulas do not depend on the heap. The de nition above can be adapted to obtain full intuitionistic separation logic, and general intuitionistic separation logic, in a similar manner as before.

C.2 Intuitionistic Reynolds' logic

In the intuitionistic version of separation logic we cannot express directly anymore that a location x is not allocated. The definition of the substitution $p[\langle x \rangle := e]$ and $p[\langle x \rangle := ?]$, and the above alt-backwards axiomatization of mutation, allocation and dispose instructions therefore breaks down. We can use new modalities $[[x] := e]$ and $[[x] := ?]$ corresponding to the mutation and the dispose instruction. Differently from the heap update operation and the heap clear operation, correctness of the modalities $[[x] := e]$ and $[[x] := ?]$ require that x is allocated. In ISL we then can define the heap update substitution $p[\langle x \rangle := e]$ by $(x \not\mapsto) \to p[[x] := e]$, as explained in details below.

Further note, that we indeed can use in the allocation axiom disjunction (instead of the intuitionistic implication) because of its classic interpretation (this is explained in the soundness and completeness proof below).

Definition C.2.1 (Substitution for mutation). We define $p[[x] := e]$ recursively on p (assuming the variables of e and x do not occur bound in p).

- $b[[x] := e] = b$,
- $(e' \mapsto e'')[[x] := e] = (x \doteq e' \wedge e' \mapsto e'') \vee (x = e' \wedge e'' = e)$,
- $(p \wedge q)[[x] := e] = p[[x] := e] \wedge q[[x] := e]$, and similar for $\vee$ and $\to$,
- $(\exists y p)[[x] := e] = \exists y(p[[x] := e])$ and similar for $\forall$,
- $(p \mathbin{-\!\!*} q)[[x] := e] = ((p[[x] := e] \wedge x \not\mapsto) \mathbin{-\!\!*} q) \vee (p \mathbin{-\!\!*} (q[[x] := e] \wedge x \not\mapsto))$
- $(p \mathbin{*\!\!-} q)[[x] := e] = p \mathbin{*\!\!-} (q[[x] := e])$

Lemma C.2.2 (Correctness mutation substitution). Let $s(x) \in \mathrm{dom}(h)$. We then have $h, s \models p[[x] := e]$ iff $h[s(x) := s(e)], s \models p$.

Proof. The proof proceeds by induction on the structure of p. We treat the following main cases.

- $h, s \models (p \to q)[[x] := e]$ iff (definition substitution)
  $h, s \models p[[x] := e] \to q[[x] := e]$ iff (semantics implication)
  $h', s \models p[[x] := e]$ implies $h', s \models q[[x] := e]$, for all h' such that $h \perp h'$
  iff (induction hypothesis)
  $h'[s(x) := s(e)], s \models p$ implies $h'[s(x) := s(e)], s \models q$, for all h' such that $h \perp h'$
  iff (see below)
  $h', s \models p$ implies $h', s \models q$, for all h' such that $h[s(x) := s(e)] \perp h'$
  iff (semantics implication)
  $h[s(x) := s(e)], s \models p \to q$
  Note that $h[s(x) := s(e)] \perp h'$ implies $h'[s(x) := h(s(x))] = h'$, and $h \perp h'$ implies $h[s(x) := s(e)] \perp h'[s(x) := s(e)]$.

^ $h; s \models (p \quad q)[[x] := e]$
i (de nition substitution)
$h; s \models (x \mapsto \quad) \Rightarrow ((p[[x] := e] \wedge x \mapsto \quad) \quad q) \_ (p \quad (q[[x] := e] \wedge x \mapsto \quad))$
i (see below)
$h[s(x) := s(e)]; s \models p \quad q$.
+: First, let $s(x) \ 2 \ dom(h)$. W.l.o.g. we may assume that $h = h_1 \ ] \ h_2$, $h_1; s \models p[[x] := e]$, and $h_2; s \models q$, for some $h_1; h_2$ such that $s(x) \ 2 \ dom(h_1)$. Induction hypothesis: $h_1[s(x) := s(e)]; s \models p$. Further: $h[s(x) := s(e)] = h_1[s(x) := s(e)] \ ] \ h_2$. Next, let $s(x) \ 62 dom(h)$. Let $h^0 = h[s(x) := n]$, for some arbitrary n. Again, w.l.o.g. we may assume that $h^0 = h_1 \ ] \ h_2$, $h_1; s \models p[[x] := e]$, and $h_2; s \models q$, for some $h_1; h_2$ such that $s(x) \ 2 \ dom(h_1)$. Induction hypothesis: $h_1[s(x) := s(e)]; s \models p$. Further: $h^0[s(x) := s(e)] = h[s(x) := s(e)] = h_1[s(x) := s(e)] \ ] \ h_2$.
*: Let $h[s(x) := s(e)] = h_1 \ ] \ h_2$ such that $h_1; s \models p$ and $h_2; s \models q$. W.l.o.g., assume that $s(x) \ 2 \ dom(h_1)$. Further , let $h \vee h^0$. Let $h_1^0 = h^0 \, n \, h_2$. It follows that $h^0 = h_1^0 \ ] \ h_2$. Monotonicity: $h_1^0; s \models p$. Induction hypothesis: $h_1^0; s \models p[[x] := e]$ (note that $h_1^0[s(x) := s(e)] = h_1^0$).

^ $h; s \models (p \quad q)[[x] := e]$
i (de nition of substitution)
$h; s \models (x \mapsto \quad) \Rightarrow (p \quad (q[[x] := e]))$ i (semantics intuitionistic and separating implication)
$h^0; s \models p$ implies $h \ ] \ h^0; s \models q[[x] := e]$, for every $h \quad h^0$, with $s(x) \ 2 \ dom(h^0)$, and h^{00} disjoint from h^0
i (induction hypothesis)
$h^{00}; s \models p$ implies $(h^0 \ ] \ h^{00})[s(x) := s(e)]; s \models q$, for every $h \quad h^0$, with $s(x) \ 2 \ dom(h^0)$, and h^{00} disjoint from h^0
i (see below)
$h^0; s \models p$ implies $h[s(x) := s(e)] \ ] \ h^0; s \models q$, for every h^0 disjoint from $h[s(x) := s(e)]$
i (semantics of separating implication)
$h[s(x) := s(e)]; s \models p \quad q$.
+: First, let $s(x) \ 62 dom(h)$. So $h \vee h[s(x) := s(e)]$, and we can take $h[s(x) := s(e)]$ for h^0. Next, let $s(x) \ 2 \ dom(h)$. So it su ces to observe that $h^{00} \# h$ implies $h^{00} \# h[s(x) := s(e)]$.
*: let $h \quad h^0$, with $s(x) \ 2 \ dom(h^0)$, and h^{00} disjoint from h^0 such that $h^{00}; s \models p$. Clearly, h^{00} is disjoint from $h[s(x) := s(e)]$, and thus we have that $h[s(x) := s(e)] \ ] \ h^{00}; s \models q$, that is, $(h \ ] \ h^{00})[s(x) := s(e)]; s \models q$. We further have that $(h \ ] \ h^{00})[s(x) := s(e)] \vee (h; \ ] \ h^{00})[s(x) := s(e)]$, and so by monotonicity, we conclude that $(h^0 \ ] \ h^{00})[s(x) := s(e)]; s \models q$. □

Corollary C.2.3 (Correctness intuitionistic heap update).
We have $h; s \models p[hxi := e]$ i $h[s(x) := s(e)]; s \models p$.

Proof. First let $h; s \models p[hxi := e]$, that is (by de nition), $h; s \models (x \mapsto \quad) \Rightarrow p[[x] := e]$. Let $h^0 = h$, if $s(x) \ 2 \ dom(h)$, and $h^0 = h[s(x) := n]$, for some arbitrary n, otherwise. So $h \vee h^0$, and thus we infer from $h; s \models (x \mapsto \quad) \Rightarrow p[[x] := e]$ that

$h'; s \models p[[x] := e]$, and so by the correctness of the mutation substitution, we have $h'[s(x) := s(e)]; s \models p$, that is, $h[s(x) := s(e)]; s \models p$.

On the other hand, assuming $h[s(x) := s(e)]; s \models p$, let $h \# h'$ such that $s(x) \in \mathrm{dom}(h')$. We show that $h'; s \models p[[x] := e]$: By the monotonicity property of ISL we have that $h[s(x) := s(e)]; s \models p$ implies $h'[s(x) := s(e)]; s \models p$. By the correctness of the mutation substitution, it then suffices to observe that $h'; s \models p[[x] := e]$ if and only if $h'[s(x) := s(e)]; s \models p$. □

For the intutitionistic axiomatization of the dispose instruction we introduce the following substitution.

Definition C.2.4 (Substitution for dispose). We define $p[[x] := ?]$ recursively on p (assuming that x does not occur bound in p).

- $b[[x] := ?] = b$
- $(e \mapsto e')[[x] := ?] = x \doteq e \wedge e \mapsto e'$
- $(p \wedge q)[[x] := ?] = p[[x] := ?] \wedge q[[x] := ?]$, and similar for $\vee$
- $(p \to q)[[x] := ?] = (p[[x] := ?] \to q[[x] := ?]) \wedge \forall y (p[[x] := y] \to q[[x] := y])$ where y is a fresh variable
- $(\exists y p)[[x] := ?] = \exists y (p[[x] := ?])$
- $(p \mathbin{-\!\!*} q)[[x] := ?] = (p[[x] := ?] \wedge (x \not\mapsto)) \mathbin{-\!\!*} q$
- $(p \mathbin{*\!\!-} q)[[x] := ?] = (p \mathbin{*\!\!-} q[[x] := ?]) \wedge \forall y (p[\langle x \rangle := y] \mathbin{*\!\!-} q[\langle x \rangle := y])$ where y is a fresh variable.

Determining whether $p \mathbin{-\!\!*} q$ holds after disposing x, we predict whether p or q holds for the sub-heap that contained the disposed x. Since the dispose instruction $[x] := ?$ requires that x is allocated, we distinguish between between these two cases by checking in which part of the heap x is allocated. But since after the dispose instruction both p and q are evaluated in sub-heaps which do not contain the location x, we can choose between where to allocate x initially. Formally, the assertions $(p[[x] := ?] \wedge (x \not\mapsto)) \mathbin{-\!\!*} q$ and $(q[[x] := ?] \wedge (x \not\mapsto)) \mathbin{-\!\!*} p$ are equivalent. For example, we that $\mathrm{true} \mathbin{-\!\!*} (x \not\mapsto y)$ does not hold after execution of $[x] := ?$. By definition $(\mathrm{true} \mathbin{-\!\!*} (x \not\mapsto y))[[x] := ?]$ reduces to $(\mathrm{true} \wedge x \not\mapsto y) \mathbin{-\!\!*} (x \not\mapsto y)$, which further reduces to false. On the other hand, $((x \not\mapsto y) \mathbin{-\!\!*} \mathrm{true})[[x] := ?]$ reduces to $(x \doteq x \wedge (x \not\mapsto)) \mathbin{-\!\!*} \mathrm{true}$, which further reduces to $\mathrm{false} \mathbin{-\!\!*} \mathrm{true}$, which also is equivalent to false.

Lemma C.2.5 (Correctness dispose substitution). Let $s(x) \in \mathrm{dom}(h)$. We then have $h; s \models p[[x] := ?]$ if and only if $h[s(x) := ?]; s \models p$.

Proof. The proof proceeds by induction on the structure of p. We treat the following main cases.

^ $h; s \models (p \to q)[[x] := ?] \iff$ (definition of substitution)
$h; s \models (p[[x] := ?] \to q[[x] := ?]) \wedge \forall y(p[[x] := y] \to q[[x] := y])$
$\iff$ (see below)
$h[s(x) := ?]; s \models p \to q$.
First we show that $h[s(x) := ?] \circ h^0$ and $h^0; s \models p$ implies $h^0; s \models q$. We distinguish the following two cases. First let $s(x) \notin dom(h^0)$. It follows that $h^0[s(x) := h(s(x))]; s \models p[[x] := ?]$ (by the induction hypothesis we have $h^0[s(x) := h(s(x))]; s \models p[[x] := ?]$ if and only if $h^0; s \models p$). Since $h \circ h^0[s(x) := h(s(x))]$ we thus derive from $h; s \models (p[[x] := ?] \to q[[x] := ?])$ that $h^0[s(x) := h(s(x))]; s \models q[[x] := ?]$, and so by the induction hypothesis again we obtain $h^0; s \models q$.

Next let $s(x) \in dom(h^0)$. From $h^0; s \models p$, it follows from the correctness of the mutation substitution that $h^0[s(x) := h(s(x))]; s^0 \models p[[x] := y]$, where $s^0 = s[y := h^0(s(x))]$ (since y does not appear in p and $x \neq y$). Since $h \circ h^0[s(x) := h(s(x))]$ we thus derive from $h; s^0 \models p[[x] := y] \to q[[x] := y]$ that $h^0[s(x) := h(s(x))]; s^0 \models q[[x] := y]$, and so by the correctness of the mutation substitution again we obtain $h^0; s^0 \models q$, that is, $h^0; s \models q$ (since y does not appear in q).

Conversely, let $h[s(x) := ?]; s \models p \to q$. First we show that $h; s \models p[[x] := ?] \to q[[x] := ?]$. Let $h \circ h^0$ and $h^0; s \models p[[x] := ?]$, and so by the induction hypothesis $h^0[s(x) := ?]; s \models p$. We have to show that $h^0; s \models q[[x] := ?]$. By the induction hypothesis ($s(x) \in dom(h) \setminus dom(h^0)$) it suffices to show that $h^0[s(x) := ?]; s \models q$. Since $h[s(x) := ?] \circ h^0[s(x) := ?]$ and $h^0[s(x) := ?]; s \models p$ we thus derive from $h[s(x) := ?]; s \models p \to q$ that $h^0[s(x) := ?]; s \models q$.

Next we show that $h; s \models \forall y(p[[x] := y] \to q[[x] := y])$. Let $h \circ h^0$ and $h^0; s^0 \models p[[x] := y]$, where $s^0 = s[y := n]$, for some arbitrary n. We have to show that $h^0; s^0 \models q[[x] := y]$ which by the correctness of the mutation substitution boils down to $h^0[s(x) := n]; s \models q$. By the correctness of the mutation substitution again we have $h^0; s^0 \models p[[x] := y]$ if and only if $h^0[s(x) := n]; s^0 \models p$. Since $h[s(x) := ?] \circ h^0[s(x) := n]$ we thus derive from $h[s(x) := ?]; s \models p \to q$ and $h^0[s(x) := n]; s^0 \models p$ that $h^0[s(x) := n]; s \models q$ (since y does not occur in p and q).

^ $h; s \models (p \mathbin{*} q)[[x] := ?] \iff$ (definition substitution)
$h; s \models (p[[x] := ?] \mathbin{\wedge} x \not\mapsto) \mathbin{*} q \iff$ (semantics separating conjunction)
$h_1; s \models p[[x] := ?] \mathbin{\wedge} x \not\mapsto$ and $h_2; s \models q$, for some $h_1; h_2$ such that $h = h_1 \circ h_2$
$\iff$ (semantics of points-to) $h_1; s \models p[[x] := ?]$ and $h_2; s \models q$, for some $h_1; h_2$ such that $s(x) \in dom(h_1)$ and $h = h_1 \circ h_2$
$\iff$ (induction hypothesis)
$h_1[s(x) := ?]; s \models p$ and $h_2; s \models q$, for some $h_1; h_2$ such that $s(x) \in dom(h_1)$ and $h = h_1 \circ h_2$
$\iff$ (see below)
$h_1; s \models p$ and $h_2; s \models q$, for some $h_1; h_2$ such that $h = h_1 \circ h_2$
$\iff$ (semantics separating conjunction)
$h[s(x) := ?]; s \models p \mathbin{*} q$. Note that $s(x) \in dom(h_1)$ and $h = h_1 \circ h_2$ implies

$h[[x] := ?] = h_1[s(x) := ?] \rceil h_2$, and $h[[x] := ?] = h_1 \rceil h_2$ implies that $h = h_1[s(x) := h(s(x))] \rceil h_2$.

$\hat{}$ $h; s \models ((p \sqsubset q)[[x] := ?]$
i (definition substitution)
$h; s \models (p \sqsubset q[[x] := ?]) \wedge \forall y(p[\langle x \rangle := y] \sqsubset q[\langle x \rangle := y])$
i (see below)
$h[s(x) := ?]; s \models p \sqsubset q$.
First let h^0 be disjoint from $h[s(x) := ?]$ and $h^0; s \models p$. We have to show that $h[s(x) := ?] \rceil h^0; s \models q$. We distinguish the following two cases.

First, let $s(x) \notin dom(h^0)$. So h^0 and h are disjoint, and thus (since $h; s \models p \sqsubset q[[x] := ?]$) we have $h \rceil h^0; s \models q[[x] := ?]$. From which we derive $(h \rceil h^0)[s(x) := ?]; s \models q$ by the induction hypothesis (note that $s(x) \in dom(h) \cap dom(h \rceil h^0)$). We then can conclude this case by the observation that $h[s(x) := ?] \rceil h^0 = (h \rceil h^0)[s(x) := ?]$.

Next, let $s(x) \in dom(h^0)$. We then introduce $s^0 = s[y := h^0(s(x))]$. Since $h^0; s^0 \models p$ (y does not appear in p), it follows by the correctness of the intuitionistic heap update (see above corollary) that $h^0[s(x) := ?]; s^0 \models p[\langle x \rangle := y]$. Since $h^0[s(x) := ?]$ and h are disjoint (which clearly follows from that h^0 and $h[s(x) := ?]$ are disjoint), and so (since $h; s^0 \models p[\langle x \rangle := y] \sqsubset q[\langle x \rangle := y]$) we have that $h \rceil (h^0[s(x) := ?]); s^0 \models q[\langle x \rangle := y]$. Applying again the correctness of the intuitionistic heap update we obtain $(h \rceil (h^0[s(x) := ?]))[s(x) := s^0(y)]; s^0 \models q$. We then can conclude this case by the assumption that y does not appear in q and the observation that $h[s(x) := ?] \rceil h^0 = (h \rceil (h^0[s(x) := ?]))[s(x) := s^0(y)]$.

Conversely, let $h[s(x) := ?]; s \models p \sqsubset q$. We first show that $h; s \models p \sqsubset q[[x] := ?]$: Let h^0 be disjoint from h and $h^0; s \models p$. We have to show that $h \rceil h^0; s \models q[[x] := ?]$. Clearly, h^0 and $h[s(x) := ?]$ are disjoint, and so (since $h[s(x) := ?]; s \models p \sqsubset q$) $h[s(x) := ?] \rceil h^0; s \models q$. By the induction hypothesis (note that $s(x) \in dom(h) \cap dom(h \rceil h^0)$) we have $h \rceil h^0; s \models q[[x] := ?]$ i $(h \rceil h^0)[s(x) := ?]; s \models q$. We then can conclude this case by the observation that $(h \rceil h^0)[s(x) := ?] = h[s(x) := ?] \rceil h^0$, because $s(x) \in dom(h) \cap dom(h^0)$.

Next we show that $h; s \models \forall y(p[\langle x \rangle := y] \sqsubset q[\langle x \rangle := y])$: Let h^0 be disjoint from h and $s^0 = s[y := n]$, for some n, such that $h^0; s^0 \models p[\langle x \rangle := y]$. We have to show that $h \rceil h^0; s^0 \models q[\langle x \rangle := y]$. By the correctness of the intuitionistic heap update it then follows that $h^0[s(x) := n]; s^0 \models p$, that is, $h^0[s(x) := n]; s \models p$ (since y does not appear in p). Since $h^0[s(x) := n]$ and $h[s(x) := ?]$ are disjoint, we derive from the assumption $h[s(x) := ?]; s \models p \sqsubset q$ that $h[s(x) := ?] \rceil h^0[s(x) := n]; s \models q$. Again by the correctness of the intuitionistic heap update, we have that $h \rceil h^0; s^0 \models q[\langle x \rangle := y]$ i $(h \rceil h^0)[s(x) := n]; s^0 \models q$ (that is, $(h \rceil h^0)[s(x) := n]; s \models q$, because y does not appear in q). We then can conclude this case by the observation that $(h \rceil h^0)[s(x) := n] = h[s(x) := ?] \rceil h^0[s(x) := n]$. □

The following axiomatization is by Reynolds [187].

De nition C.2.6 (Weakest precondition axiomatization).

$$f\, p[x := e]\, g\ x := e\ f\, p\, g$$

$$f\, \exists y((e \mapsto y) \wedge p[x := y])\, g\ x := [e]\ f\, p\, g$$

$$f\, (x \not\mapsto)\ ((x \not\mapsto e) \multimap p)\, g\ [x] := e\ f\, p\, g$$

$$f\, \forall y((y \not\mapsto e) \multimap p[x := y])\, g\ x := new\, (e)\ f\, p\, g$$

$$f\, (x \not\mapsto)\ p\, g\ delete\ (x)\ f\, p\, g$$

The next two axiomatizations are novel.

De nition C.2.7 (Alternative weakest precondition axiomatization).

$$f\, p[x := e]\, g\ x := e\ f\, p\, g$$

$$f\, \exists y((e \mapsto y) \wedge p[x := y])\, g\ x := [e]\ f\, p\, g$$

where y is fresh

$$f\, (x \not\mapsto) \wedge p[[x] := e]\, g\ [x] := e\ f\, p\, g$$

$$f\, \forall x((x \not\mapsto) \vee p[\langle x \rangle := e])\, g\ x := new\, (e)\ f\, p\, g$$

where $x \notin fv(e)$

$$f\, (x \not\mapsto) \wedge p[[x] := ?]\, g\ delete\ (x)\ f\, p\, g$$

De nition C.2.8 (Strongest postcondition axiomatization).

$$f\, p\, g\ x := e\ f\, (\exists y)(p[x := y] \wedge e[x := y] = x)\, g$$

$$f\, (e \mapsto) \wedge p\, g\ x := [e]\ f\, \exists y(p[x := y] \wedge e[x := y] \not\mapsto x)\, g$$

$$f\, (x \not\mapsto) \wedge p\, g\ [x] := e\ f\, \exists y(p[[x] := y]) \wedge (x \not\mapsto e)\, g$$

where $x \notin fv(e)$

$$f\, p\, g\ x := new\, (e)\ f\, (\exists y(p[x := y]))[[x] := ?] \wedge (x \not\mapsto e)\, g$$

$$f\, (x \not\mapsto) \wedge p\, g\ delete\ (x)\ f\, (x \not\mapsto) \Rightarrow \exists y(p[[x] := y])\, g$$

where y is fresh everywhere

We showcase the soundness and completeness of the strongest postcondition axiomatization of dispose (soundness and completeness of the above axiomatization of the other instructions follow in a straightforward manner from the corresponding substitution lemmas).

- $\models \{p \wedge (x \mapsto \_)\}\ \mathtt{delete}(x)\ \{(x \mapsto \_) \mathrel{-\!\!*} \exists y(p[[x] := y])\}$:
  Let $h; s \models r \wedge x \mapsto \_$. We have to show that $h[s(x) := ?]; s \models x \mapsto \_ \mathrel{-\!\!*} \exists y(p[[x] := y])$. That is, for $h[s(x) := ?] \bullet h'$ such that $s(x) \in \mathrm{dom}(h')$ we have to show that $h'; s \models \exists y(p[[x] := y])$: We show $h'[s(x) := n]; s[y := n] \models p$ for $n = h(s(x))$: By Lemma C.2.2, we have $h'; s[y := n] \models p[[x] := y]$ if and only if $h'[s(x) := n]; s[y := n] \models p$. By monotonicity ($h; s \models p$ and $h \bullet h'[s(x) := n]$) it follows that $h'[s(x) := n]; s \models p$. Since y does not appear in p, we thus have that $h'[s(x) := n]; s[y := n] \models p$.

- $\models \{p \wedge (x \mapsto \_)\}\ \mathtt{delete}(x)\ \{q\}$ implies $\models ((x \mapsto \_) \mathrel{-\!\!*} \exists y(p[[x] := y])) \mathrel{-\!\!*} q$:
  This boils does to showing that $h; s \models x \mapsto \_ \mathrel{-\!\!*} \exists y(p[[x] := y])$ implies $h; s \models q$, for any heap h and store s. So, let $h; s \models x \mapsto \_ \mathrel{-\!\!*} \exists y(p[[x] := y])$, that is, $h'; s \models x \mapsto \_$ implies $h'; s \models \exists y(p[[x] := y])$, for any $h \bullet h'$. Let $h' = h$, in cases $(x) \in \mathrm{dom}(h)$, and $h' = h[s(x) := n]$, for some arbitrary n, otherwise. Clearly, $h \bullet h'$ and $h'; s \models x \mapsto \_$. So $h'; s \models \exists y(p[[x] := y])$. Let $h'; s[y := k] \models p[[x] := y]$, for some k. By Lemma C.2.2 again, it follows that $h'[s(x) := k]; s[y := k] \models p$. From our assumption $\models \{p \wedge x \mapsto \_\}\ [x] := ?\ \{q\}$ we then derive that $h'[s(x) := ?]; s[y := k] \models q$. By definition of h' we have that $h'[s(x) := ?] \bullet h$, and so by monotonicity we infer $h; s[y := k] \models q$, and so $h; s \models q$, assuming w.l.o.g. that y does not appear (free) in q.

Appendix D

Formalization in Coq

The main motivation behind formalizing results in a proof assistant is to rigorously check hand-written proofs. For our formalization we used the dependently-typed calculus of inductive constructions as implemented by the Coq proof assistant.

This thesis is accompanied by an artifact [112]. In this appendix, we discuss two parts of the artifact: one corresponding to the alternative axiomatization of Reynolds' logic (see Section 4.5 and Appendix C) based on the insights from dynamic separation logic (see Section 4.4), and one corresponding to the natural deduction proof system for separation logic (see Section 3.3).

D.1 Alternative axiomatization

In this part of the artifact, we have used no axioms other than the axiom of function extensionality (for every two functions f, g we have that $f = g$ if $f(x) = g(x)$ for all x) and propositional extensionality (equivalent propositions are equal). This means that we work with an underlying intuitionistic logic: we have not used the axiom of excluded middle for reasoning classically about propositions. However, the decidable propositions (propositions P for which the excluded middle $P \lor \neg P$ can be proven) allow for a limited form of classical reasoning.

We formalize the basic instructions of our programming language (assignment, look-up, mutation, allocation, and deallocation) and the semantics of basic instructions. For Boolean and arithmetic expressions we use a shallow embedding, so that those expressions can be directly given as a Coq term of the appropriate type (with a coincidence condition assumed, i.e. that values of expressions depend only on finitely many variables of the store).

There are two approaches in formalizing the semantics of assertions: shallow and deep embedding. We have taken both approaches. In the first approach, the shallow embedding of assertions, we define assertions of DSL by their extension of satisfiability (i.e. the set of heap and store pairs in which the assertion is satisfied), that must satisfy a coincidence condition (assertions depend only on finitely many variables of the store) and a stability condition (see below). The definition of

the modality operator follows from the semantics of programs, which includes basic control structures such as the while -loop. In the second approach, the deep embedding of assertions, assertions are modeled using an inductive type and we explicitly introduce two meta-operations on assertions that capture the heap update and heap clear modality. We have omitted the clauses for emp and $(e \mapsto e')$, since these could be defined as abbreviations, and we restrict to the basic instructions.

In the deep embedding we have no constructor corresponding to the program modality $[S]p$. Instead, two meta-operations denoted $p[\langle x \rangle := e]$ and $p[\langle x \rangle := ?]$ are defined recursively on the structure of p. Crucially, we formalized and proven the following lemmas (the details are almost the same as showing the equivalences hold in the shallow embedding, Lemmas 4.4.3 and 4.4.4):

Lemma D.1.1 (Heap update substitution lemma).
$h; s \models p[\langle x \rangle := e] \Leftrightarrow h[s(x) := s(e)]; s \models p$.

Lemma D.1.2 (Heap clear substitution lemma).
$h; s \models p[\langle x \rangle := ?] \Leftrightarrow h[s(x) := ?]; s \models p$.

By also formalizing a deep embedding, we show that the modality operator can be defined entirely on the meta-level by introducing meta-operations on formulas that are recursively defined by the structure of assertions: this captures Theorem 4.4.5. For technical simplicity we restrict ourselves to the basic instructions, but it should be natural to extend the formalization of the completeness result to languages with while -statements, e.g. following [31]. On the other hand, in the shallow embedding it is easier to show that our approach can be readily extended to complex programs including while -loops.

In both approaches, the semantics of assertions is classical, although we work in an intuitionistic meta-logic. We do this by employing a double negation translation, following the set-up by R. O'Connor [162]. In particular, we have that our satisfaction relation $h; s \models p$ is stable, i.e. $\neg\neg(h; s \models p)$ implies $h; s \models p$. This allows us to do classical reasoning on the image of the higher-order semantics of our assertions.

The source code of our formalization is accompanied with this thesis as a digital artifact. The artifact consists of the following files:

- shallow/Language.v : Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a shallow embedding of our assertion language, as presented in the prequel.
- shallow/Proof.v : Provides proof of the equivalences (E1-16), and additionally standard equivalences for modalities involving complex programs.
- deep/Heap.v: Provides an axiomatization of heaps as partial functions.
- deep/Language.v: Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a deep embedding of our assertion language, on which we inductively define the meta operations of heap update and heap clear. We finally formalize Hoare triples and proof systems using weakest precondition and strongest postcondition axioms for the basic instructions.

- deep/Classical.v : Provides the classical semantics of assertions, and the strong partial correctness semantics of Hoare triples. Further it provides proofs of substitution lemmas corresponding to our meta-operators. Finally, it provides proofs of the soundness and completeness of the aforementioned proof systems.

D.2 Natural deduction

In this part of the artifact we use the classical axiom of excluded middle.

The proof system based on natural deduction is embedded in the Coq proof assistant using an axiomatic approach. It is known that adding axioms to Coq may a ect soundness: it is future work to show that the axioms can be consistently used. Nonetheless, using the axiomatic approach allows us to stay close to the natural deduction proof system as introduced in this thesis.

```
Axiom D: Type.
Axiom hasval: D -> D -> Prop.
```

The type D is used as domain, over which we let the relation hasval range. This relation is the weak points to relation, so the strong points to relation can be de ned in terms of this relation.

```
Axiom sep: Prop -> Prop -> Prop.
Axiom sepimp: Prop -> Prop -> Prop.
```

We also introduce axiomatically new connectives for use in propositions. Coq is also extended with syntax for constructing separation logic formulas.

```
Axiom rooted: Prop -> (D -> D -> Prop) -> Prop.
Axiom root_equiv: forall (A: Prop), A <-> rooted A hasval.
Axiom root_above: forall (A: Prop) h h' ,
  (forall x y, h x y <-> h' x y) -> rooted A h -> rooted A h'.
Axiom root_assoc: forall (A: Prop) h h' ,
  rooted (rooted A h) h' <-> rooted A (fun x y => rooted (h x y) h').
```

A rooted assertion consists of an assertion and a (rst-order) description of the heap with respect to which it is evaluated. Note that in our axiomatization we do not limit that the given function actually is a rst-order description. We also axiomatize that any assertion is equivalent to the rooted assertion (@¦), that we can replace equivalent descriptions of the heap, and extensionality of the description of the heap.

```
Axiom root_True: forall (h: D -> D -> Prop), rooted True h.
Axiom root_False: forall (h: D -> D -> Prop), rooted False h -> False.
Axiom root_hasval: forall (h: D -> D -> Prop) x y,
  rooted (hasval x y) h <-> h x y.
```

```
Axiom root_eq: forall (h: D -> D -> Prop) (T: Type) (x y: T),
  rooted (x = y) h <-> x = y.
Axiom root_split ' : forall (A B: Prop) h,
  (rooted A h /\ rooted B h) -> rooted (A /\ B) h.
Axiom root_join ' : forall (A B: Prop) h,
  (rooted A h \/ rooted B h) -> rooted (A \/ B) h.
Axiom root_and_elim: forall (A B: Prop) h,
  rooted (A /\ B) h -> rooted A h /\ rooted B h.
Axiom root_or_elim: forall (A B: Prop) h,
  rooted (A \/ B) h -> rooted A h \/ rooted B h.
Axiom root_imp' : forall (A B: Prop) h,
  (rooted A h -> rooted B h) -> rooted (A -> B) h.
Axiom root_imp_elim ' : forall (A B: Prop) h,
  rooted (A -> B) h -> rooted A h -> rooted B h.
Axiom root_forall ' : forall (T: Type) (A: T -> Prop) h,
  (forall (x: T), rooted (A x) h) -> rooted (forall (x: T), A x) h.
Axiom root_forall_elim ' : forall (T: Type) (A: T -> Prop) h,
  rooted (forall (x: T), A x) h -> forall (x: T), rooted (A x) h.
Axiom root_exists ' : forall (T: Type) (A: T -> Prop) h,
  (exists (x: T), rooted (A x) h) -> rooted (exists (x: T), A x) h.
Axiom root_exists_elim ' : forall (T: Type) (A: T -> Prop) h,
  rooted (exists (x: T), A x) h -> exists (x: T), rooted (A x) h.
```

We also axiomatize reasoning about the classical connectives under a rooted assertion.

```
Definition Par (h1 h2: D -> D -> Prop) :=
  (forall x y, h1 x y -> forall z, ~h2 x z) /\
  (forall x y, h2 x y -> forall z, ~h1 x z).
Definition Split (h h1 h2: D -> D -> Prop) :=
  (forall x y, h x y <-> h1 x y \/ h2 x y) /\ Par h1 h2.
Axiom sep_elim' : forall (A B C: Prop) (h: D -> D -> Prop),
  rooted (A ** B) h ->
  (forall h1 h2, Split h h1 h2 -> rooted A h1 -> rooted B h2 -> C) -> C.
Axiom sep_intro ' : forall (A B: Prop) (h: D -> D -> Prop),
  (exists h1 h2, Split h h1 h2 /\ rooted A h1 /\ rooted B h2) ->
  rooted (A ** B) h.
Axiom sepimp_elim' : forall (A B C: Prop) (h h ' : D -> D -> Prop),
  rooted (A -** B) h ->
  Par h h' /\ rooted A h ' /\
    (rooted B (fun x y => h x y \/ h ' x y) -> C) -> C.
Axiom sepimp_intro ' : forall (A B: Prop) (h: D -> D -> Prop),
  (forall h ', Par h h ' -> rooted A h ' ->
    rooted B (fun x y => h x y \/ h ' x y)) ->
  rooted (A -** B) h.
```

Finally, we introduce axioms for reasoning about the new separating connectives. We use the syntax** for separating conjunction and -** for separating implication.

From these axioms, it becomes possible to prove the following lemmas:

```
Lemma sep_assoc (A B C: Prop): (A ** B) ** C <-> A ** B ** C.
Lemma sep_Empty (A: Prop): A ** emp <-> A.
Lemma sep_or (A B C: Prop): (A \/ B) ** C <-> A ** C \/ B ** C.
Lemma sep_and (A B C: Prop): (A /\ B) ** C -> A ** C /\ B ** C.
Lemma adjoint (A B: Prop): A ** (A -** B) -> B.
```

Also the modalities can be de ned and properties proven:

```
Definition box (A: Prop) := True ** (emp /\ (True -** A)).
Lemma box_elim (A: Prop): box A -> A.
Lemma box_indep (A: Prop): box A -> forall h, rooted (box A) h.
Lemma root_under: forall (A B: Prop),
  box (A -> B) -> forall h, rooted A h -> rooted B h.
Lemma box_rooted (A: Prop): (forall h, rooted A h) -> box A.
Lemma sep_mono (A' AB B : Prop):
  box (A -> A' ) -> box (B -> B ' ) -> A ** B -> A '  ** B' .
```

Finally, we can prove the equivalence:

```
Definition F1: Prop :=
  alloc x /\ ((x = y /\ z = w) \/ (x <> y /\ hasval y z)).
Definition F2: Prop :=
  pointsToDash x ** (pointsTo x w -** hasval y z).
Proposition F12 ' : F1 -> F2.
Proposition F21: F2 -> F1.
```

The artifact consists of the following le:

- proof/Language.v : Provides an axiomatization of natural deduction for separation logic, and proves a number of lemmas based on theses axioms.

Bibliography

[1] Andrew Aberdein. Mathematical wit and mathematical cognition. Topics in Cognitive Science, 5(2):231 250, 2013.

[2] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In International Workshop on Current Trends in Applied Formal Methods, pages 168 183. Springer, 1998.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. Deductive Software Veri cation - The KeY Book - From Theory to Practice, volume 10001 of Lecture Notes in Computer Science. Springer, 2016.

[4] Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic: To be or not to be created. In 2nd World Congress on Formal Methods (FM), volume 5850 of Lecture Notes in Computer Science, pages 612 627. Springer, 2009.

[5] Mikbs Ajtai. Isomorphism and higher order equivalence. Annals of Mathematical Logic, 16(3):181 203, 1979.

[6] Mahmudul Faisal Al Ameen. Completeness of Veri cation System with Separation Logic for Recursive Procedures. PhD thesis, Tokyo, 2016.

[7] Paul Ammann and Je O utt. Introduction to software testing. Cambridge University Press, 2016.

[8] Marc Andreessen. Why software is eating the world. Wall Street Journal, 2011.

[9] Peter B. Andrews. An introduction to mathematical logic and type theory: to truth through proof, volume 27 of Applied Logic Series. Springer, 2013.

[10] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rudiger Olderog. Veri cation of sequential and concurrent programs. Springer, 3rd edition, 2009.

[11] Krzysztof R. Apt and Ernst-Rudiger Olderog. Fifty years of Hoare's logic. Formal Aspects of Computing 31(6):751 807, 2019.

[12] Lukas Armborst and Marieke Huisman. Permission-based verification of red-black trees and their merging. In 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE), pages 111 123. IEEE, 2021.

[13] Mario Rodriguez Artalejo. Some questions about expressiveness and relative completeness in Hoare's logic. Theoretical computer science, 39:189 206, 1985.

[14] Rob Arthan, Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. A general framework for sound and complete Floyd-Hoare logics. ACM Transactions on Computational Logic (TOCL) , 11(1):1 31, 2009.

[15] Callum Bannister, Peter Höfner, and Gerwin Klein. Backwards and forwards with separation logic. In Jeremy Avigad and Assia Mahboubi, editors, 9th International Conference on Interactive Theorem Proving (ITP) , volume 10895 of Lecture Notes in Computer Science, pages 68 87. Springer, 2018.

[16] Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. Proceedings of the ACM on Programming Languages, 4(POPL):1 30, 2019.

[17] Jon Barwise. Handbook of mathematical logic. Elsevier, 1982.

[18] Davide Basile. Specification and Verification of Contract-Based Applications . PhD thesis, University of Pisa, 2016.

[19] Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. Foundations for entailment checking in quantitative separation logic. In 31st European Symposium on Programming (ESOP), volume 13240 of Lecture Notes in Computer Science, pages 57 84. Springer, 2022.

[20] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), volume 3328 of Lecture Notes in Computer Science, pages 97 109. Springer, 2005.

[21] Jan A. Bergstra and John V. Tucker. Expressiveness and the completeness of Hoare's logic. Journal of computer and system sciences, 25(3):267 284, 1982.

[22] Jan A. Bergstra and John V. Tucker. Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs. Theoretical Computer Science, 17(3):303 315, 1982.

[23] Guram Bezhanishvili and Lawrence S. Moss. Undecidability of first-order logic, 2009. Educational module for the NSF-sponsored project on Learning Discrete Mathematics and Computer Science via Primary Historical Sources.

[24] Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, and Stijn de Gouw. Integrating ADTs in KeY and their application to history-based reasoning. In 24th International Symposium on Formal Methods (FM), volume 13047 of Lecture Notes in Computer Science. Springer, 2021.

[25] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In ACM Conference on Computer and Communications Security, pages 833 844, 2012.

[26] David Binder, Thomas Piecha, and Peter Schroeder-Heister. The Logical Writings of Karl Popper . Springer, 2022.

[27] Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In 10th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS), volume 4423 of Lecture Notes in Computer Science, pages 93 107. Springer, 2007.

[28] Ales Bizjak and Lars Birkedal. On models of higher-order separation logic. Electronic Notes in Theoretical Computer Science, 336:57 78, 2018.

[29] Frank S. de Boer and Hans-Dieter A. Hiep. Completeness and complexity of reasoning about call-by-value in Hoare logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 43(4), October 2021.

[30] M. Boogaard and E. Spoor. The software crisis in the Netherlands. In Serie Research Memoranda, No. 1994-21. Vrije Universiteit Amsterdam, 1994.

[31] George S. Boolos, John P. Burgess, and Richard C. Je rey. Computability and logic. Cambridge University Press, 2002.

[32] Richard Bornat. Proving pointer programs in Hoare logic. In 5th International Conference on Mathematics of Program Construction (MPC), pages 102 126. Springer, 2000.

[33] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In Proceedings of the 32nd ACM Symposium on Principles of Programming Languages, pages 259 270, 2005.

[34] William D. Brewer. Godel's doctoral thesis, 1928 30: The completeness of rst-order logic. In Kurt Godel: The Genius of Metamathematics, pages 101 129. Springer, 2022.

[35] Remi Brochenin, Stephane Demri, and Etienne Lozes. On the almighty wand. Information and Computation , 211:106 137, 2012.

[36] Stephen Brookes. A semantics for concurrent separation logic. Theoretical Computer Science, 375(1-3):227 270, 2007.

[37] Stephen Brookes. A revisionist history of concurrent separation logic. Electronic Notes in Theoretical Computer Science, 276:5 28, 2011.

[38] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. ACM SIGLOG News, 3(3):47 65, 2016.

[39] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. ACM SIGPLAN Notices , 43(1):101 112, 2008.

[40] James Brotherston and Max Kanovich. Undecidability of propositional separation logic and its neighbours. In 25th IEEE Symposium on Logic in Computer Science (LICS), pages 130 139. IEEE, 2010.

[41] Luitzen Egbertus Jan Brouwer. Intuitionism and formalism. In A. Heyting, editor, Philosophy and Foundations of Mathematics, pages 123 138. Elsevier, 1975.

[42] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. Machine intelligence, 7(23-50):3, 1972.

[43] Cristiano Calcagno. Semantic and Logical Properties of Stateful Programming PhD thesis, Universita di Genova, 2002.

[44] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to rst-order logic. In 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS), volume 3441 of Lecture Notes in Computer Science, pages 395 409. Springer, 2005.

[45] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. Electronic Notes in Theoretical Computer Science, 172:133 175, 2007.

[46] Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. Bringing order to the separation logic jungle. In 15th Asian Symposium on Programming Languages and Systems (APLAS), pages 190 211. Springer, 2017.

[47] Chen Chung Chang and H. Jerome Keisler. Model theory. Elsevier, 1990.

[48] Adam Chlipala. Mostly-automated veri cation of low-level programs in computational separation logic. In 32nd ACM Conference on Programming Language Design and Implementation (PLDI), pages 234 245, 2011.

[49] Krzysztof Ciesielski. Set theory for the working mathematician. Cambridge University Press, 1997.

[50] Edmund M. Clarke Jr. Completeness and incompleteness theorems for Hoare-like axiom systems PhD thesis, Cornell University, 1976.

[51] Edmund M. Clarke Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. Journal of the ACM, 26(1):129 147, 1979.

[52] Edmund M. Clarke Jr. The characterization problem for Hoare logics. Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences, 312(1522):423 440, 1984.

[53] Edmund M. Clarke Jr., Steven M. German, and Joseph Y. Halpern. E ective axiomatizations of Hoare logics. Journal of the ACM, 30(3):612 636, 1983.

[54] Timothy T.R. Colburn, James H. Fetzer, and R.L. Rankin. Program verication: Fundamental issues in computer science, volume 14 of Studies in Cognitive Systems Springer, 2012.

[55] Stephen A. Cook. Soundness and completeness of an axiom system for program veri cation. SIAM Journal on Computing, 7(1):70 90, 1978.

[56] Stephen A. Cook and Derek C. Oppen. An assertion language for data structures. In 2nd ACM Symposium on Principles of Programming Languages (POPL) , pages 160 166, 1975.

[57] Laura Crosilla. Predicativity and Feferman. In Gerhard Jäger and Wilfried Sieg, editors, Feferman on Foundations: Logic, mathematics, philosophy, pages 423 447. Springer, 2017.

[58] H.-H. Dang, Peter Höfner, and Bernhard Möller. Algebraic separation logic. Journal of Logic and Algebraic Programming, 80(6):221 247, 2011.

[59] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound automation of magic wands. In 34th International Conference on Computer Aided Veri cation (CAV) , volume 13372 of Lecture Notes in Computer Science, pages 130 151. Springer, 2022.

[60] Clayton Allen Davis, Onur Varol, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Botornot: A system to evaluate social bots. In 25th International Conference Companion on World Wide Web, pages 273 274. ACM, 2016.

[61] Jacobus W. de Bakker. Mathematical theory of program correctness. Prentice-Hall, 1980.

[62] Jacobus W. de Bakker and Lambert G.L.T. Meertens. On the completeness of the inductive assertion method. Journal of Computer and System Sciences, 11(3):323 357, 1975.

[63] Frank S. de Boer. Reasoning about Dynamically Evolving Process Structures; a proof theory for the parallel object-oriented language pool. PhD thesis, Vrije Universiteit Amsterdam, April 1991.

[64] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDK's sort method for generic collections. Journal of Automated Reasoning 62(1):93 126, 2019.

[65] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof Pearl: The KeY to correct and stable sorting. Journal of Automated Reasoning, 53(2):129 139, 2014.

[66] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. In 27th International Conference on Computer Aided Veri cation (CAV) , volume 9206 of Lecture Notes in Computer Science, pages 273 289. Springer, 2015.

[67] Stephane Demri and Morgan Deters. Logical investigations on separation logics. European Summer School on Logic, Language and Information (ESSLLI), 2015.

[68] Stephane Demri and Morgan Deters. Separation logics and modalities: a survey. Journal of Applied Non-Classical Logics, 25(1):50 99, 2015.

[69] Stephane Demri, Etienne Lozes, and Alessio Mansutti. The e ects of adding reachability predicates in propositional separation logic. In 21st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS), volume 10803 of Lecture Notes in Computer Science, pages 476 493. Springer, 2018.

[70] Stephane Demri, Etienne Lozes, and Alessio Mansutti. A complete axiomatisation for quanti er-free separation logic. Logical Methods in Computer Science, 17(3), 2021.

[71] Edsger W. Dijkstra. A discipline of programming. Prentice-Hall, 1976.

[72] Yifan Ding, Nicholas Botzer, and Tim Weninger. Posthoc veri cation and the fallibility of the ground truth. arXiv preprint arXiv:2106.07353 , 2021.

[73] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In 40th ACM Symposium on Principles of Programming Languages (POPL) , pages 287 300. ACM, 2013.

[74] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 3920 of Lecture Notes in Computer Science, pages 287 302. Springer, 2006.

[75] Brijesh Dongol, Victor B.F. Gomes, and Georg Struth. A program construction and veri cation tool for separation logic. In 12th International Conference on Mathematics of Program Construction (MPC), volume 9129 of Lecture Notes in Computer Science, pages 137 158. Springer, 2015.

[76] Frederick Douzet, Louis Pétiniaud, Loqman Salamatian, Kave Salamatian, and Thibaut Alchus. Measuring the fragmentation of the internet: the case of the Border Gateway Protocol (BGP) during the Ukrainian crisis. In 12th International Conference on Cyber Con ict (CyCon), volume 1300, pages 157 182. IEEE, 2020.

[77] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. On the expressive completeness of Bernays-Schon nkel-Ramsey separation logic. arXiv preprint arXiv:1802.00195, 2018.

[78] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. The Bernays-Schon nkel-Ramsey class of separation logic with uninterpreted predicates. ACM Transactions on Computational Logic (TOCL), 21(3):1 46, 2020.

[79] Herbert B. Enderton. A mathematical introduction to logic. Elsevier, 2001.

[80] Gergo Erdi. Compositional Type Checking. Master thesis, Eotvos Lorand University, 2011.

[81] Mahmudul Faisal Al Ameen and Makoto Tatsuta. Completeness for recursive procedures in separation logic. Theoretical Computer Science, 631:73 96, 2016.

[82] William M. Farmer. Simple Type Theory: A Practical Logic for Expressing and Reasoning About Mathematical Ideas. Springer, 2023.

[83] Melvin Fitting. Proof methods for modal and intuitionistic logics, volume 169 of Synthese Library. Springer, 1983.

[84] Robert W. Floyd. Assigning meanings to programs. In Program Veri cation: Fundamental Issues in Computer Science, pages 65 81. Springer, 1993.

[85] Thomas Forster. Quine's New Foundations. In The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, 2019.

[86] Nissim Francez. Program veri cation. Addison-Wesley, 1992.

[87] Thomas Frayne, Anne Morel, and Dana Scott. Reduced direct products. Fundamenta mathematicae, 51(3):195 228, 1962.

[88] Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A Perez. A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency. Proceedings of the ACM on Programming Languages, 6(OOPSLA2):841 869, 2022.

[89] Didier Galmiche and Dominique Larchey-Wendling. Expressivity properties of Boolean BI through relational models. In 26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), volume 4337 of Lecture Notes in Computer Science, pages 357 368. Springer, 2006.

[90] Didier Galmiche and Daniel Méry. Tableaux and resource graphs for separation logic. Journal of Logic and Computation, 20(1):189 231, 2010.

[91] Mohan Ganesalingam. The language of mathematics. Springer, 2013.

[92] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In 25th International Conference on Formal Methods for Industrial Critical Systems (FMICS), volume 12327 of Lecture Notes in Computer Science, pages 3 69. Springer, 2020.

[93] Kurt Godel. Uber die vollstandigkeit des logikkalkuls PhD thesis, University of Vienna, 1929.

[94] Michael D. Godfrey and David F. Hendry. The computer as Von Neumann planned it. IEEE Annals of the History of Computing, 15(1):11 21, 1993.

[95] Joseph A. Goguen et al. Formal methods: Promises and problems. IEEE Software, 14(1):73 85, 1997.

[96] Gerald Arthur Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Master thesis, University of Toronto, 1975.

[97] Clemens Grabmayer. Relating proof systems for recursive types. PhD thesis, Vrije Universiteit Amsterdam, 2005.

[98] Clemens Grabmayer. From abstract rewriting systems to abstract proof systems. arXiv preprint arXiv:0911.1412, 2009.

[99] Michal Grabowski. On relative completeness of Hoare logics. Information and control, 66(1-2):29 44, 1985.

[100] David Gries. The science of programming Springer, 2012.

[101] Jan Friso Groote, Ammar Osaiweran, and Jacco H. Wesselius. Analyzing the e ects of formal methods on the development of industrial control software. In 27th IEEE International Conference on Software Maintenance (ICSM), pages 467 472. IEEE, 2011.

[102] Reiner Hahnle. Dijkstra's legacy on program veri cation. In C.A.R. Hoare Krzysztof R. Apt, editor, Edsger Wybe Dijkstra: His Life, Work, and Legacy, pages 105 140. ACM, 2022.

[103] Anthony Hall. Seven myths of formal methods. IEEE software, 7(5):11 19, 1990.

[104] Anthony Hall. Realising the bene ts of formal methods. In 7th International Conference on Formal Engineering Methods (ICFEM), volume 3785 of Lecture Notes in Computer Science. Springer, 2005.

[105] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual e ectiveness of logic in computer science. Bulletin of Symbolic Logic, 7(2):213 236, 2001.

[106] Richard Wesley Hamming. The unreasonable e ectiveness of mathematics. The American Mathematical Monthly, 87(2):81 90, 1980.

[107] David Harel. First-Order Dynamic Logic , volume 68 of Lecture Notes in Computer Science Springer, 1979.

[108] Leon Henkin. The completeness of the rst-order functional calculus. Journal of Symbolic Logic, 14(3):159 166, 1949.

[109] Leon Henkin. Completeness in the theory of types. Journal of Symbolic Logic, 15(2), 1950.

[110] Andreas Herzig. A simple separation logic. In 20th International Workshop on Logic, Language, Information, and Computation (WoLLIC) , volume 8071 of Lecture Notes in Computer Science, pages 168 178. Springer, 2013.

[111] Joel Hestness, Stephen W. Keckler, and David A. Wood. A comparative analysis of microarchitecture e ects on CPU and GPU memory system behavior. In 2014 IEEE International Symposium on Workload Characterization (IISWC) , pages 150 160. IEEE, 2014.

[112] Hans-Dieter A. Hiep. New Foundations for Separation Logic (Coq artifact), 2024. https://dx.doi.org/10.5281/zenodo.10558424 .

[113] Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. History-based speci cation and veri cation of Java collections in KeY. In 16th International Conference on Integrated Formal Methods (iFM), volume 12546 of Lecture Notes in Computer Science, pages 199 217. Springer, 2020.

[114] Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. Verifying OpenJDK's LinkedList using KeY (extended paper). International Journal on Software Tools for Technology Transfer, 24(5):783 802, 2022.

[115] Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, and Stijn de Gouw. Verifying OpenJDK's LinkedList using KeY. In 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 12079 of Lecture Notes in Computer Science, pages 217 234. Springer, 2020.

[116] Scott A. Hissam, Daniel Plakosh, and C. Weinstock. Trust and vulnerability in open source software. IEEE Proceedings-Software, 149(1):47 51, 2002.

[117] Charles Anthony Richard Hoare. How did software get so reliable without proof? In 3rd International Symposium of Formal Methods Europe (FME), volume 1051 of Lecture Notes in Computer Science, pages 1 17. Springer, 1996.

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

[119] Wilfrid Hodges. A shorter model theory. Cambridge University Press, 1997.

[120] Johannes Hostert, Andrej Dudenhefner, and Dominik Kirst. Undecidability of dyadic rst-order logic in Coq. In 13th International Conference on Interactive Theorem Proving (ITP) . Schloss Dagstuhl-Leibniz-Zentrum fur Informatik, 2022.

[121] Zhe Hou and Alwen Tiu. Completeness for a rst-order abstract separation logic. In 14th Asian Symposium on Programming Languages and Systems (APLAS) , volume 10017 of Lecture Notes in Computer Science, pages 444 463. Springer, 2016.

[122] Zhe Hou and Alwen Tiu. Completeness for a rst-order abstract separation logic. In Atsushi Igarashi, editor, Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings, volume 10017 of Lecture Notes in Computer Science, pages 444 463, 2016.

[123] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal methods: From academia to industrial practice. arXiv preprint arXiv:2002.07279 , 2020.

[124] Roberto Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. Computer languages, 19(1):19 40, 1993.

[125] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In 28th ACM Symposium on Principles of Programming Languages (POPL), pages 14 26, 2001.

[126] Sushil Jajodia, Paulo Shakarian, V.S. Subrahmanian, Vipin Swarup, and Cli Wang. Cyber Warfare: Building the Scienti c Foundation , volume 56 of Advances in Information Security. Springer, 2015.

[127] Jakob L. Jensen, Michael E. J rgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic veri cation of pointer programs using monadic second-order logic. In 18th ACM Conference on Programming Language Design and Implementation (PLDI) , pages 226 234, 1997.

[128] Capers Jones. Measuring defect potentials and defect removal e ciency. Journal of Defense Software Engineering, 21(6):11 13, 2008.

[129] Capers Jones and Olivier Bonsignour. The economics of software quality. Addison-Wesley Professional, 2011.

[130] Paul C. Jorgensen and Byron DeVries. Software testing: a craftsman's approach. CRC Press, 5th edition, 2021.

[131] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In Proceedings of the ACM on Programming Languages, volume 2 of POPL, pages 1 34. ACM, 2017.

[132] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, 28:e20, 2018.

[133] Samuel Kamin. The expressive theory of stacks. Acta informatica , 24:695 709, 1987.

[134] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Muck, Benjamin Peters, Gert Smolka, et al. A Coq library for mechanised rst-order logic. In The Coq Workshop 2022. hal.science, 2022.

[135] Stephen Cole Kleene. Introduction to metamathematics. Wolters-Noordho , 1971.

[136] Stephen Cole Kleene. The work of Kurt Godel. Journal of Symbolic Logic, 41(4):761 778, 1976.

[137] Ralf Kneuper. Limits of formal methods. Formal Aspects of Computing 9:379 394, 1997.

[138] Tomasz Kowaltowski. Correctness of programs manipulating data structures PhD thesis, University of California, Berkeley, 1973.

[139] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In 44th ACM Symposium on Principles of Programming Languages (POPL), pages 205 217, 2017.

[140] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In 4th International Workshop on Types in Language Design and Implementation, pages 105 116. ACM, 2009.

[141] Krishan Kumar and Sonal Dahiya. Programming languages: A survey. International Journal on Recent and Innovation Trends in Computing and Communication, 5(5):307 313, 2017.

[142] Mark Steven Laventhal. Veri cation of programs operating on structured data. Bachelor and master thesis, MIT, 1974.

[143] Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. ACM SIGPLAN Notices , 49(1):477 490, 2014.

[144] Martti Lehto and Pekka Neittaanmaki. Cyber security: Critical infrastructure protection, volume 56 of Computational Methods in Applied Sciences. Springer, 2022.

[145] Ewen Maclean, Andrew Ireland, and Gudmund Grov. Proof automation for functional correctness in separation logic. Journal of Logic and Computation, 26(2):641 675, 2016.

[146] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP miscon guration. ACM SIGCOMM Computer Communication Review, 32(4):3 16, 2002.

[147] Makarov Evgeny Maratovich. Dynamic separation logic and its use in education. Ñîâðåìåííûå èíôîðìàöèîííûå òåõíîëîãèè è ÈÒ-îáðàçîâàíèå , 16(3):543 550, 2020.

[148] Nicolas Marti and Reynald A eldt. A certi ed veri er for a fragment of separation logic. Information and Media Technologies, 4(2):304 316, 2009.

[149] Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. Hoare logic in the abstract. In 20th International Workshop on Computer Science Logic (CSL) volume 4207 of Lecture Notes in Computer Science, pages 501 515. Springer, 2006.

[150] Jeferson Martnez and Javier M. Duran. Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study. International Journal of Safety and Security Engineering, 11(5):537 545, 2021.

[151] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Veri cation condition generation via theorem proving. In 13th International Conference on Logic for Programming, Arti cial Intelligence, and Reasoning (LPAR) , volume 4246 of Lecture Notes in Computer Science, pages 362 376. Springer, 2006.

[152] Tom F. Melham. Higher order logic and hardware veri cation. Cambridge University Press, 2009.

[153] Elliott Mendelson. Introduction to Mathematical Logic. CRC Press, 6th edition, 2015.

[154] Bertrand Meyer. Design by contract and the component revolution. In 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS). IEEE Computer Society, 2000.

[155] Ronald Middelkoop. A proof system for object oriented programming using separation logic. Master thesis, Technische Universiteit Eindhoven, 2003.

[156] Raul E. Monti, Robert Rubbens, and Marieke Huisman. On deductive veri cation of an industrial concurrent software component with VerCors. In International Symposium on Leveraging Applications of Formal Methods, pages 517 534. Springer, 2022.

[157] J.H. Morris. Veri cation oriented language design. Technical report, University of California, Berkeley, 1972.

[158] Joseph M. Morris. A general axiom of assignment.Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by F.L. Bauer, E.W. Dijkstra and C.A.R. Hoare, pages 25 34, 1982.

[159] Peter Mu ller, Malte Schwerho , and Alexander J Summers. Viper: A veri cation infrastructure for permission-based reasoning. In Veri cation, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17, pages 41 62. Springer, 2016.

[160] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. The art of software testing. Wiley Online Library, 2nd edition, 2004.

[161] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and German Andres Delbianco. Communicating state transition systems for ne-grained concurrent resources. In23rd European Symposium on Programming Languages and Systems (ESOP) volume 8410 ofLecture Notes in Computer Science, pages 290 310. Springer, 2014.

[162] Russell O'Connor. Classical mathematics for a constructive world.Mathematical Structures in Computer Science, 21(4):861 882, 2011.

[163] Peter W. O'Hearn. Resources, concurrency, and local reasoning.Theoretical computer science, 375(1):271 307, 2007.

[164] Peter W. O'Hearn. A primer on separation logic (and automatic program veri cation and analysis). Software safety and security 33:286 318, 2012.

[165] Peter W. O'Hearn. Incorrectness logic. In Proceedings of the ACM on Programming Languages, volume 4 of POPL, pages 1 32. ACM, 2019.

[166] Peter W. O'Hearn. Separation logic. Communications of the ACM, 62(2):86 95, 2019.

[167] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. Bull. Symb. Log., 5(2):215 244, 1999.

[168] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, 15th International Workshop on Computer Science Logic (CSL) volume 2142 of Lecture Notes in Computer Science, pages 1 19. Springer, 2001.

[169] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In 7th ACM Symposium on Theory of Computing, pages 107 116, 1975.

[170] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. Acta informatica, 6(4):319 340, 1976.

[171] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM, 19(5):279 285, 1976.

[172] Jens Pagel and Florian Zuleger. Strong-separation logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 44(3):1 40, 2022.

[173] Matthew J. Parkinson. Local reasoning for Java. Technical report, University of Cambridge, Computer Laboratory, 2005.

[174] David Lorge Parnas. Really rethinking `formal methods'. Computer, 43(1):28 34, 2010.

[175] David Parsons. Foundational Java: Key Elements and Practical Programming. Springer, 2020.

[176] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), volume 2884 of Lecture Notes in Computer Science, pages 64 78. Springer, 2003.

[177] Ruzica Piskac, Thomas Wies, and Damien Zu erey. Automating separation logic using SMT. In 25th International Conference on Computer Aided Veri cation (CAV) , volume 8044 of Lecture Notes in Computer Science, pages 773 789. Springer, 2013.

[178] Ruzica Piskac, Thomas Wies, and Damien Zu erey. Automating separation logic with trees and data. In 26th International Conference on Computer Aided Veri cation (CAV) , volume 8559 of Lecture Notes in Computer Science, pages 711 728. Springer, 2014.

[179] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In 17th Annual Symposium on Foundations of Computer Science, pages 109 121. IEEE, 1976.

[180] David Pym, Jonathan M. Spring, and Peter W. O'Hearn. Why separation logic works. Philosophy & Technology, 32:483 516, 2019.

[181] David J. Pym. The semantics and proof theory of the logic of bunched implications. In Applied Logic Series, 2002.

[182] Panu Raatikainen. Godel's incompleteness theorems. In The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, 2020.

[183] Brian Randell. The 1968/69 NATO software engineering reports, 1996.

[184] Habib ur Rehman, Eiad Ya , Mohammed Nazir, and Khurram Mustafa. Security assurance against cybercrime ransomware. In International Conference on Intelligent Computing & Optimization (ICO) , pages 21 34. Springer, 2018.

[185] Andrew Reynolds, Radu Iosif, and Cristina Serban. Reasoning in the Bernays-Schon nkel-Ramsey fragment of separation logic. In 18th International Conference on Veri cation, Model Checking, and Abstract Interpretation (VMCAI) , volume 10145 of Lecture Notes in Computer Science, pages 462 482. Springer, 2017.

[186] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in smt. In International Symposium on Automated Technology for Veri cation and Analysis, pages 244 261. Springer, 2016.

[187] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, Millennial Perspectives in Computer Science, Cornerstones of Computing, pages 303 321. Macmillan Education UK, 2000.

[188] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In 17th IEEE Symposium on Logic in Computer Science (LICS) pages 55 74. IEEE Computer Society, 2002.

[189] John C. Reynolds. An overview of separation logic. In First Conference on Veri ed Software: Theories, Tools, Experiments (VSTTE) , volume 4171 of Lecture Notes in Computer Science, pages 460 469. Springer, 2005.

[190] John C. Reynolds. An introduction to separation logic. In Engineering Methods and Tools for Software Safety and Security, pages 285 310. IOS Press, 2009.

[191] Tom Ridge and James Margetson. A mechanically veri ed, sound and complete theorem prover for rst order logic. In 18th International Conference on Theorem Proving in Higher Order Logics, volume 3603 of Lecture Notes in Computer Science, pages 294 309. Springer, 2005.

[192] Dennis M. Ritchie. The development of the C programming language. In Richard G. Gibson Thomas J. Bergin, editor, History of Programming languages, pages 671 698. ACM, 1996.

[193] David S. Rosenblum. Formal methods and testing: why the state-of-the art is not the state-of-the practice. ACM SIGSOFT Software Engineering Notes, 21(4):64 66, 1996.

[194] Yaman Roumani. Patching zero-day vulnerabilities: an empirical analysis. Journal of Cybersecurity, 7(1), 2021.

[195] Mohammad Salahuddin, Khorshed Alam, and Ilhan Ozturk. The effects of Internet usage and economic growth on CO_2 emissions in OECD countries: A panel investigation. Renewable and Sustainable Energy Reviews, 62:1226–1235, 2016.

[196] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In International Conference on Formal Verification of Object-Oriented Software (FoVeOOS), volume 6528 of Lecture Notes in Computer Science, pages 138–152. Springer, 2011.

[197] Ravi Sen. Challenges to cybersecurity: Current state of affairs. Communications of the Association for Information Systems, 43(1):2, 2018.

[198] Syed Muhammad Ali Shah, Maurizio Morisio, and Marco Torchiano. An overview of software defect density: A scoping study. In 19th Asia-Pacific Software Engineering Conference, volume 1, pages 406–415. IEEE, 2012.

[199] Stewart Shapiro. Foundations without foundationalism: A case for second-order logic. Clarendon Press, 1991.

[200] Sajjan G. Shiva. Advanced computer architectures. CRC Press, 2018.

[201] Mihaela Sighireanu, Juan A Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, et al. Sl-comp: competition of solvers for separation logic. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 116–132. Springer, 2019.

[202] Raymond M. Smullyan. Gödel's incompleteness theorems. Oxford University Press, 1992.

[203] Raymond M. Smullyan. First-Order Logic. Dover, 1995.

[204] James Somers. The coming software apocalypse. The Atlantic, 26:1, 2017.

[205] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. Acta Informatica, 27:505–517, 1990.

[206] Bjarne Stroustrup. A history of C++: 1979–1991. In Richard G. Gibson Thomas J. Bergin, editor, History of Programming languages, pages 671–698. ACM, 1996.

[207] Johanna Stuber. Verification of Red-Black Trees in KeY: A Case Study in Deductive Java Verification. Bachelor thesis, Karlsruher Institut für Technologie (KIT), 2023.

[208] Andy S. Tatman. Analysis and Formal Specification of OpenJDK's BitSet. Bachelors thesis, Leiden University, 2023.

[209] Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. Completeness and expressiveness of pointer program verification by separation logic. Information and Computation, 267:1–27, 2019.

[210] Balder ten Cate, Johan van Benthem, and Jouko Vaananen. Lindström theorems for fragments of first-order logic. In 22nd Annual IEEE Symposium on Logic in Computer Science (LICS), pages 280–292. IEEE, 2007.

[211] Aditya Thakur, Jason Breck, and Thomas Reps. Satisfiability modulo abstraction for separation logic with linked lists. In International SPIN Symposium on Model Checking of Software, pages 58–67. ACM, 2014.

[212] Anne Sjerp Troelstra and Helmut Schwichtenberg. Basic Proof Theory. Cambridge University Press, 2nd edition, 2000.

[213] Anne Sjerp Troelstra and Dirk Van Dalen. Constructivism in Mathematics, Vol 1. Elsevier, 1988.

[214] Anne Sjerp Troelstra and Dirk Van Dalen. Constructivism in Mathematics, Vol 2. Elsevier, 1988.

[215] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In 34th ACM Symposium on Principles of Programming Languages (POPL), pages 97–108, 2007.

[216] Jouko Vaananen. Second order logic or set theory? Bulletin of Symbolic Logic, 18(1):91–121, 2012.

[217] Johan van Benthem and Kees Doets. Higher-order logic. In Handbook of philosophical logic, pages 189–243. Springer, 1983.

[218] Guido van Rossum and Fred L. Drake Jr. Python tutorial. Technical report, Centrum voor Wiskunde en Informatica, 1995.

[219] Moshe Y. Vardi. Move fast and break things. Communications of the ACM, 61(9):7–7, 2018.

[220] Philip Wadler. Propositions as types. Communications of the ACM, 58(12):75–84, 2015.

[221] Benjamin Wagner. Understanding internet shutdowns: A case study from Pakistan. International Journal of Communication, 12(1):22, 2018.

[222] Tjark Weber. Towards mechanized program verification with separation logic. In 18th International Workshop on Computer Science Logic (CSL), volume 3210 of Lecture Notes in Computer Science, pages 250–264. Springer, 2004.

[223] Mark Allen Weiss. Data structures and algorithm analysis in Java. Pearson Education, 2012.

[224] Stephen B. Wicker. The ethics of zero-day exploits the NSA meets the trolley car. Communications of the ACM, 64(1):97 103, 2020.

[225] Eugene P. Wigner. The unreasonable e ectiveness of mathematics in the natural sciences.Communications on Pure and Applied Mathematics, 13:1 14, 1960.

[226] Jeannette M. Wing. A speci er's introduction to formal methods. Computer, 23(9):8 22, 1990.

[227] Glynn Winskel. The formal semantics of programming languages: an introduction. MIT Press, 1993.

[228] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, Joao C Pereira, and Peter Muller. Gobra: Modular speci cation and veri cation of go programs. In 33rd International Conference on Computer Aided Veri cation (CAV) , volume 12759 ofLecture Notes in Computer Science, pages 367 379. Springer, 2021.

[229] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In 6th International ACM Symposium on Machine Programming pages 1 10. ACM, 2022.

[230] Hongseok Yang.Local reasoning for stateful programs PhD thesis, University of Illinois, 2001.

[231] Hongseok Yang. Relational separation logic.Theoretical Computer Science, 375(1-3):308 334, 2007.

[232] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and U e Engberg, editors,5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) volume 2303 ofLecture Notes in Computer Science, pages 402 416. Springer, 2002.

[233] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In 2009 IEEE International Conference on Software Maintenance, pages 274 283. IEEE, 2009.

[234] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. IEEE Security & Privacy , 7(2):87 90, 2009.

[235] Job Zwiers, Ulrich Hannemann, Yassine Lakhnech, Willem P. de Roever, and Frank A. Stomp. Modular completeness: Integrating the reuse of speci ed software in top-down program development. In Marie-Claude Gaudel and Jim Woodcock, editors, FME'96: Industrial Bene t and Advances in Formal Methods, volume 1051 ofLecture Notes in Computer Science, pages 595 608. Springer, 1996.

List of Publications

1. 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

2. Formal Speci cation and Analysis of OpenJDK's BitSet Class
   Andy S. Tatman, Hans-Dieter A. Hiep, Stijn de Gouw
   In: iFM 2023: 18th International Conference, iFM 2023, Proceedings
   Lecture Notes in Computer Science, volume 14300
   Springer, 2023

3. 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 2023, Proceedings
   Lecture Notes in Computer Science, volume 14278
   Springer, 2023

4. Integrating ADTs in KeY and their application to History-Based Reasoning about Collection
   Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, Stijn de Gouw
   Formal Methods in System Design, volume 61
   Springer, 2022

5. 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

6. 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

7. Integrating ADTs in KeY and their application to History-Based Reasoning
   Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, Stijn de Gouw
   In: Formal Methods, 24th International Symposium, FM 2021, Proceedings
   Lecture Notes in Computer Science, volume 13047
   Springer, 2021

8. 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

9. History-Based Speci cation and Veri cation of Java Collections in KeY
   Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, Stijn de Gouw
   In: Integrated Formal Methods, 16th International Conference, IFM 2020, Proceedings
   Lecture Notes in Computer Science, volume 12546
   Springer, 2020

10. A Tutorial on Verifying LinkedList Using KeY
    Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, Stijn de Gouw
    In: Deductive Software Veri cation: Future Perspectives, Re ections on the Occasion of 20 Years of KeY
    Lecture Notes in Computer Science, volume 12345
    Springer, 2020

11. History-based speci cation and veri cation of Java collections in KeY
    Frank S. de Boer, Hans-Dieter A. Hiep
    In: Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs
    Association for Computing Machinery, 2020

12. Reowolf: Synchronous Multi-party Communication over the Internet
    Christopher Esterhuyse, Hans-Dieter A. Hiep
    In: Formal Aspects of Component Software, 16th International Conference, FACS 2019, Proceedings
    Lecture Notes in Computer Science, volume 12018
    Springer, 2019

13. 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

14. Reowolf 1.0: Project Documentation
    Christopher A. Esterhuyse, Hans-Dieter A. Hiep
    Technical Report
    CWI, 2020

15. Axiomatic Characterization of Trace Reachability for Concurrent Objects
    Frank S. de Boer, Hans-Dieter A. Hiep
    In: Integrated Formal Methods, 15th International Conference, IFM 2019, Proceedings
    Lecture Notes in Computer Science, volume 11918
    Springer, 2019

See also the ORCiD page (0000-0001-9677-6644) for the current list of publications.

Summary

The research presented in this thesis concerns one of the most important questions in software engineering of our time: how can we make sure that software is free from memory safety bugs? Memory safety bugs are the major cause of common vulnerabilities and exposures, and their presence threatens the stability and security of our digital world. This question is so important that it has escalated to the highest level. In a recent White House press release (February 26, 2024), the National Cyber Director of the United States of America calls on the academic community to help solve this hard problem:[1] addressing [this challenge] is imperative to ensuring we can secure our digital ecosystem long-term and protect the security of our Nation. The accompanying technical report advises on the use of memory safe programming languages, and prominently mentions formal methods as one way to achieve the highly desired freedom from bugs, including memory safety bugs.

In this thesis, formal methods are studied that are used to analyze software for its correctness, where correctness means that software satisfies its specification and incorrectness means the presence of a bug. The focus is on separation logic, a formal method designed as a scalable technique in ensuring freedom from memory safety bugs. Nowadays, separation logic is a well-established field of research: it has been widely studied academically in the past twenty years, and is successfully applied on an industry-wide scale to ensure memory safety. For example, separation logic is the technique used to prove, with mathematical certainty, that memory safe programming languages (such as Rust and Go) indeed live up to the promise that they offer a way to eliminate, not just mitigate, entire bug classes.

In two parts, this thesis presents important scientific contributions that fill a gap in the academic literature. The first part contains the missing completeness theorem for separation logic, that is on par with the fundamental result by Gödel for first-order logic. Completeness is important for any formal method as it shows that the formal method can be adequately used for demonstrating every validity. The second part finally introduces dynamic separation logic that gives an alternative way to analyze memory safety problems, such that now it is possible to prove elementary specifications without needing extra logical techniques. This is important because it ensures `backwards compatibility' with automated reasoning techniques that are optimized for first-order logic.

[1] https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

Samenvatting

Titel:

Nieuwe fundamenten voor separatielogica

Het onderzoek dat in dit proefschrift wordt gepresenteerd betreft één van de meest belangrijke vragen in programmatuurkunde op dit moment: hoe zorgen we ervoor dat software geen geheugenbeveiligingsgaten bevat? Geheugenbeveiligingsgaten zijn de grootste veroorzaker van veelvoorkomende kwetsbaarheden en lekken, en zijn een ernstige bedreiging voor de stabiliteit en veiligheid van onze digitale wereld. Deze vraag is dermate belangrijk dat het is geëscaleerd tot het hoogste niveau. In een recent persbericht van het Witte Huis (26 februari 2024) vraagt de National Cyber Director van de Verenigde Staten de academische gemeenschap om hulp om dit hardnekkige probleem op te lossen[1]: het aanpakken van [deze uitdaging] is noodzakelijk om te zorgen voor de lange-termijn beveiliging van ons digitale ecosysteem en om onze nationale veiligheid te beschermen. Het bijbehorende rapport adviseert over het gebruik van programmeertalen die geheugenveilig zijn, en geeft nadrukkelijk aan dat gebruik van formele methoden leidt naar de zeer gewenste vrijheid van bugs, waaronder de vrijheid van geheugenbeveiligingsgaten.

In dit proefschrift bestuderen we formele methoden voor het analyseren van software op correctheid, waarbij correctheid betekent dat software voldoet aan diens specificatie en incorrectheid betekent dat er een bug schuilgaat. De focus ligt op separatielogica, een formele methode ontworpen als een schaalbare techniek voor het garanderen van vrijheid van geheugenveiligheidsgaten. Vandaag de dag is separatielogica een bewezen wetenschapsgebied: de afgelopen twintig jaar is het uitgebreid bestudeerd binnen de academie, en zijn er tal van succesvolle toepassingen in de industrie waarbij geheugenbeveiligingsgaten worden bestreden. Zo wordt separatielogica als techniek toegepast om met wiskundige zekerheid te bewijzen dat geheugenveiligetalen (zoals Rust en Go) daadwerkelijk de belofte nakomen om volledige categorieën bugs, niet alleen te mitigeren, maar te vermijden.

In twee delen presenteert dit proefschrift belangrijke wetenschappelijke bijdra-

[1] https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

gen die een kloof in de academische literatuur dicht. Het eerste deel bevat de ontbrekende volledigheidsstelling voor separatielogica, dat gelijk staat aan het fundamentele resultaat van Gödel voor de predicatenlogica. Volledigheid is belangrijk voor elke formele methode omdat het laat zien dat de formele methode adequaat gebruikt kan worden, om alles wat valide is te demonstreren. Eindelijk introduceert het tweede deel dynamische separatielogica, dat een alternatieve manier geeft voor het analyseren van geheugenbeveiligingsproblemen zodat het nu mogelijk is om basale specificaties te bewijzen zonder extra logische technieken. Dit is belangrijk omdat het 'achterwaartse compatibiliteit' geeft met technieken voor geautomatiseerd redeneren die optimaal werken voor predicatenlogica.

Curriculum Vitae

Hans-Dieter Anton Hiep was born on the 21st of March, 1991 in Hoorn, North-Holland, the Netherlands. In 2010, he passed the matriculation examinations in preparatory scientific education (VWO) at Werenfridus, Tabor College, in Hoorn. In 2016, he completed a Bachelor of Science (BSc) degree *cum laude* in Computer Science, at the Vrije Universiteit (VU), in Amsterdam. In 2018, he completed a Master of Science (MSc) joint degree *cum laude* in Computer Science, at both the Vrije Universiteit (VU) and the University of Amsterdam (UvA).

From November 2017 until June 2024, Hiep worked at the Dutch national research laboratory for mathematics and computer science Centrum Wiskunde & Informatica (CWI) in Amsterdam, and from November 2020 until June 2024 at the Leiden Institute of Advanced Computer Science (LIACS) of Leiden University.

From July 2024, Hiep will start working as an Applied Scientist in the Automated Reasoning Group at Amazon Web Services, in Cambridge, United Kingdom.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *E cient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *E cient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Bu er Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *E cient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Gri oen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be a ected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Cosimulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More E ective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04