CERTIFIED REASONING FOR AUTOMATED VERIFICATION

ASANKHAYA SHARMA

NATIONAL UNIVERSITY OF SINGAPORE

2014

CERTIFIED REASONING FOR AUTOMATED VERIFICATION

ASANKHAYA SHARMA

B.Tech. in Computer Science & Engineering
National Institute of Technology, Warangal

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

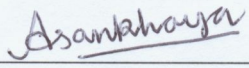
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Asankhaya Sharma

December 30, 2014

Acknowledgements

I would like to thank my adviser, Associate Professor Chin Wei Ngan for his constant support and encouragement over the last four years. He introduced me to the topic of formal verification and guided me throughout the journey of my PhD studies. He is the nicest person I have ever met and his influence in areas other than my studies has also been profound. He has taught me to build character, never give up and overcome difficulties that seem insurmountable. I am also grateful to Assistant Professor Aquinas Hobor for his help and mentorship on my research. From him, I learned the importance of certified proofs and the use of the Coq proof assistant. I appreciate the time taken by my thesis committee members Associate Professor Dong Jin Song and Associate Professor Hugh Anderson. They gave constructive feedback that helped shape the thesis.

During the past four years, I had the opportunity to work and collaborate with some of the best people I know. I appreciate my coauthors Andreea Costea, Shengyi Wang, Cristina David, Quang Loc Le and Florin Craciun for their comments and discussion on our papers which helped improve my overall understanding of the topic of automated verification. I also thank Cristian Gherghina, Shengchao Qin and Le Duy Khanh for reading and reviewing my research papers. My colleagues and friends at the Programming Language Research Lab made the journey more fun and exciting. The research lab environment was made more cheerful by the company of Ton Chanh Le, Yamilet Serrano, Quang Trung Ta, Narcisa Milea and Vijayaraghavan Murali. At NUS, I had an opportunity to meet new people, learn new things and experience new ideas. I am thankful to Shuang Liu, Kuldeep Kumar and Christopher Chak for providing a comforting friend circle on campus.

I appreciate the support of my parents who offered words of encouragement and motivation whenever I needed it. They have always been a role model for me and I hope to make them proud some day. I thank my younger brother Akash for believing in me and for the all fun times we had together. I am thankful to my in-laws for their faith in me and giving the hand of their daughter in marriage. I could not have finished this thesis without my wife Sakhee. I am grateful for her love, understanding and patience. During the last four years, I missed several weekends and skipped many vacations I could have spent with her. She provided constant support, help and advise. I promise, I will make it up to you!

Asankhaya Sharma,

Singapore, December 30, 2014

Contents

Summary

Formal methods help improve the quality and reliability of software by providing proof of correctness. However, ensuring the correctness of verification tools that apply these formal methods is itself a much harder problem. A typical way to justify the correctness is to provide soundness proofs based on semantic models. For program verifiers these soundness proofs are quite large and complex. In this thesis, we introduce *certified reasoning* to provide machine checked proofs of various components of an automated verification system. We develop new certified decision procedures and certified proofs to integrate with an existing automated verification system. Certified reasoning improves the correctness and expressivity of automated verification without sacrificing on performance.

We present a certified decision procedure (`Omega++`) for Presburger arithmetic extended with positive and negative infinity. The correctness of the procedure is established in the proof assistant Coq. Our decision procedure enables concise specifications, improves expressivity and compositionality while been efficient and scalable. We have integrated this decision procedure in `HIP/SLEEK` verification system for separation logic. This allows us to verify programs using infinite data structures and infer pure properties using `Omega++`. Next, in order to show that *certified reasoning* is not limited to the use of proof assistants, we present a certified subtype checker for the Scala

language. We reduce subtyping of traits and mixins in Scala to checking entailments in separation logic. We have extended Scala with a domain specific language SLEEK DSL, that enables checking the validity of separation logic entailments inside Scala. This shows how certified reasoning based on SLEEK can check subtyping in Scala. We have applied our technique to the Scala standard library and found that 67% of the classes do indeed conform to behavioral subtyping.

Automated verification of data structures with complex sharing is a challenging problem. Separation logic based methods have shown success in modular verification of shared mutable data structures. However, for programs using partially shared heaps, modular compositional proofs are hard to get. Verifying such programs is of practical importance as they occur in many device drivers (I/O scheduler), operating system kernels (process scheduler) and compilers (garbage collector). From a more theoretical perspective, shared usage of heap is an intrinsic property of certain data structures (such as DAGs, graphs and overlaid data structures). Existing mechanisms do not provide natural concise specifications for programs using shared heaps. We address this challenge by providing an extension of separation logic that can reason about sharing and aliasing in heaps.

We present an approach to specify and certify heap based data structures with complex sharing patterns. Our specification mechanism takes into account different sharing and aliasing scenarios. We build on the prior work of immutability specifications and support fine grained reasoning with field annotations. We introduce the notion of memory specifications to capture the precise footprint and usage of heap. In our logic we give concise and precise specifications for correctness conditions of programs using complex shared heaps. Noninterference is an important property of shared reasoning and

concurrency. We present an entailment procedure which can verify programs that use data structures with shared heaps while preserving noninterference by using a notion of *compatible sharing*. This enables us to verify practical algorithms using threaded trees and overlaid data structures. We also present a *certified proof* of correctness of compatibility checking using memory specifications. In case of interference between shared heaps, we calculate the effect of state updates using ramifications. We present a method to automatically verify ramifications in separation logic. This enables us to certify programs using data structures like DAGs and graphs.

Our verification system is totally automated and is based only on user defined predicates and user specified lemmas. We have developed a prototype for our system called `HIPComp` which is built on the `HIP/SLEEK` verification system for separation logic. `HIPComp` enables users to do automated, sound and modular reasoning about sharing in data structures. We have validated our approach on a benchmark of small but challenging programs manipulating data structures with sharing. These programs include simple examples of practical algorithms for process scheduler, disk scheduler and graph marking.

List of Tables

List of Figures

Chapter 1

Introduction

Software is increasingly playing an important role in everyday life. Many important services rely on software for proper functioning and running. With increase in use and development of software, the correct functioning of programs takes a lot of importance. Bugs or errors in the execution of programs can lead to costly mistakes (financial software) or even loss of human life (medical device software). It becomes all the more important to ensure reliability of software. Validating that programs perform the tasks they are expected to do can help in maintaining high quality and assurance levels for software.

There are two major ways to build reliable software systems. One is to do rigorous testing by executing the programs for a number of inputs. Another is to produce a formal proof of correctness of software by mathematical analysis. To quote Dijkstra [29], "Program testing can be used to show the presence of bugs, but never their absence!". While formal verification typically involves a lot of effort on part of the developer or programmer. This thesis focuses on building reliable software by formal verification. In particular, we are interested in building tools and techniques to analyze different data structures commonly

used in programs.

Data structures with heap sharing are widely used in system software. Sharing enables more efficient use of memory and allows programmers to write small programs. However, it can be quite a challenge to formally reason about such programs. In addition, certain data structures like acyclic and cyclic graphs have intrinsic sharing. Sharing makes it harder to reason about different parts of the data structure in isolation. There is a need for better specification mechanism to express the sharing of heaps and enable compositional proofs about properties of programs manipulating such heaps. The success of various automated industrial strength verifiers (such as Microsoft SDV and Astrée) in improving the quality of software has prompted research into more expressive verification systems (such as KIV, Dafny, VCC, ACL2 and PVS).

Automated verification through program analysis (in case of Astrée) can be used as a push button technology by the users to check their programs. In addition, software model checking has been instrumental to reduce the bugs found in windows device drivers (Microsoft SDV). However, it is not always possible to express properties related to functional correctness in a form that can be supported by the program analyzer. Use of a more expressive logic helps in representing such properties (ACL2 and PVS). It is challenging to design complete and sound procedures for deciding properties in expressive logics.

In recent years separation logic [96, 47] has been successfully applied for automated verification of programs [23, 9, 39, 38]. Separation logic is an extension of Hoare logic [44], which enables compositional and modular reasoning of programs in the presence of heap and mutable data structures. In order to verify a program using Hoare logic we must first specify the desired correctness properties using a specification language. In Hoare logic we annotate every method with a pre and post condition. The precondition is

required to hold before call to the method and the postcondition is established at the end of the method. Thus, for each command c we consider a Hoare triple $\{P\}\ c\ \{Q\}$ with precondition P and postcondition Q. A verifier automatically checks if the given program code is correct w.r.t to its pre/post condition. Program correctness ensures safety and reliability of software. In fact, software verification has been identified as a Grand Challenge [45, 50] for computing research.

Several tools [23, 9] have been created to verify programs using separation logic. These tools can do automated verification of programs using annotations about pre and post conditions supplied by the users (along with loop invariants). Separation logic has been instrumental in increasing the popularity of formal verification as a method to ensure reliability of software.

Separation logic is particularly useful in verifying heap manipulating programs. In addition to the usual logical operators ($\neg, \wedge, \vee$), in separation logic, assertions[1] valid on disjoint portions of heap can be represented using the spatial or separating conjunction ($*$). Separating conjunction allows reasoning about different parts of a data structure in isolation with each other. This property (local reasoning) is present in reasoning with separation logic because updates on disjoint heaps do not affect each other. Local reasoning [88] is the key to scalable verification with separation logic. Local reasoning states that in order to verify a method we need to consider only the heap which is modified in that method. By the use of separation logic, the heap memory assertions can be made more precise (with the help of must-aliases implied by

[1]Our separation logic is both "Java-like" and "C-like". Our logic is "Java-like" in the sense that heap locations (pointers) contain (point to) indivisible objects rather than individual memory cells, avoiding the possibility of pointers pointing into the middle of a structure (*i.e.*, skewing). On the other hand, our logic is "C-like" because our formulae are given a classical rather than intuitionistic semantics, *i.e.*, $\mathtt{x} \mapsto \mathtt{pair}\langle \mathtt{f}, \mathtt{s} \rangle$ means that the heap contains **exactly** a single `pair` object at the location pointed to by `x` rather than **at least** a single `pair` object at `x`.

the separating conjunction) and concise (with the help of frame conditions). The [FRAME] rule of separation logic enables local reasoning.

$$\text{[FRAME]}\quad \frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}}$$

The [FRAME] rule states that if a command c can be executed in a heap satisfying precondition P and postcondition Q, then it can also be executed in a larger heap with precondition $P * F$ and postcondition $Q * F$ (with the side condition that variables modified in c cannot occur free in F). Or conversely, reading the rule bottom up, a disjoint heap assertion F can be framed on both the pre and post condition. As an example consider the following assignment, where x and y are two pointers which are known to be not aliased. The [FRAME] rule allows us to remove (frame away) the $\mathtt{y} \mapsto \_$ assertion in order to verify the assignment.

$$\{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}$$
$$\{\mathtt{P} \equiv \mathtt{x} \mapsto \_\}$$
$$\mathtt{x.val} = \mathtt{1};$$
$$\{\mathtt{Q} \equiv \mathtt{x} \mapsto \mathtt{1}\}$$
$$\{\mathtt{x} \mapsto \mathtt{1} * \mathtt{y} \mapsto \_\}$$

This means that we can verify the code for a smaller heap and the proof carries over to the larger heap. In other words, our specifications for methods can be small in the sense that we need to specify only the heap that is actually accessed in the method. The [FRAME] rule ensures that the we can execute the method in a larger heap and need not redo the proof again. Hence the above example is also valid for a larger heap as shown below.

$$\{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_ * \mathtt{z} \mapsto \_\}$$

$$\mathtt{x.val} = 1;$$

$$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto \_ * \mathtt{z} \mapsto \_\}$$

[FRAME] rule is the key to achieve scalability in modular verification with separation logic. Hence it is desirable to support this rule (framing) even in cases where the heaps cannot be made disjoint in the pre or post condition. Many common data structures can actually be represented using the separating conjunction. For example, in a linked list the head of the list is separated from the rest of the list and in a binary tree the left and right children are separated. However, there are certain data structures (like overlaid data structures and graphs) where it is not possible to isolate them using separating conjunction. Sharing in data structures leads to following challenges for verification:

- Parts of data structures cannot be isolated.
- Reasoning about data structures requires use of global invariants about heap.
- In the absence of a [FRAME] rule and presence of global invariants it becomes difficult to make the verification method scale to larger programs.
- The invariants about data structures are difficult to state (unnatural) which makes it harder to express interesting properties.

In these cases we can still get compositional proofs and enable local reasoning if we can show that the shared heaps of the data structure do not interfere. This gives us the following [NONINTER] rule.

[NONINTER]

$$\frac{\{P\}\ c\ \{Q\}\ \ noninter(R,P) \wedge noninter(R',Q)}{\{R\}\ c\ \{R'\}}$$

The [NONINTER] rule generalizes the [FRAME] rule by allowing us to conclude R and R' from P and Q as long as the precondition P and postcondition Q does not interfere with R and R' respectively. For our running example of two pointers `x` and `y`, consider the following case where the two pointers are aliased. We cannot represent them using the separation conjunction ($*$) and hence we cannot use the [FRAME] rule directly. However, we can still verify the assignment if we can check that the heap specified in the precondition `P` (postcondition `Q`) does not interfere with the larger heap given by `R` (`R`$'$).

$$\{\mathtt{R} \equiv (\mathtt{x} \mapsto 1 \wedge \mathtt{y} \mapsto 1)\}$$
$$\{\mathtt{P} \equiv \mathtt{x} \mapsto 1\}$$
$$\mathtt{x.val} = 1;$$
$$\{\mathtt{Q} \equiv \mathtt{x} \mapsto 1\}$$
$$\{\mathtt{R} \equiv (\mathtt{x} \mapsto 1 \wedge \mathtt{y} \mapsto 1)\}$$

The [NONINTER] rule allows us to handle cases where the shared heaps do not affect each other. As a more realistic example, consider the IO scheduler in Linux which maintains an overlaid structure of doubly-linked list and a red-black tree. The linked list is used to record the order in which nodes are inserted in the queue, and the red-black tree provides an efficient indexing structure on sector fields of nodes. In such an overlaid data structure, provided the fields are not updated, the linked list and the tree do not interfere with each other even though they share the entire heap. More formally, such a data

structure can be specified using a conjunction [68], e.g. given a predicate representing a linked list : $\texttt{ll}\langle\texttt{x}\rangle$, and a predicate representing a tree : $\texttt{tree}\langle\texttt{t}\rangle$, the following predicate $\texttt{ll}\langle\texttt{x}\rangle \wedge \texttt{tree}\langle\texttt{t}\rangle$ captures an overlaid data structure of a list and tree. In this thesis we present an automated procedure to detect non-interference and verify programs using such data structures. Data structures like graphs involve unrestricted sharing, the left and right children of a binary DAG may point into each other. Data structures with unrestricted sharing lead to the following challenges for verification:

- Due to unrestricted sharing it is harder to specify which parts of the data structure are shared.

- Updates made to different shared parts of the data structure need to preserve the shape properties.

- Local changes made to a part of data structure may have the unintended consequence of changing other shared parts.

Recently, Hobor and Villard [46] have introduced the [RAMIFY] rule to deal with such cases.

$$[\textbf{RAMIFY}]$$

$$\frac{\{P\}\ c\ \{Q\} \quad ramify(R, P, Q, R')}{\{R\}\ c\ \{R'\}}$$

The key idea being that if we cannot prove noninterference, we need to show that the result of replacing $\texttt{P}$ in $\texttt{R}$ by $\texttt{Q}$ implies $\texttt{R}'$. The [RAMIFY] rule supports this kind of reasoning. For the running example with two pointers it is clear that removing $\texttt{x} \mapsto \_$ from $\texttt{R}$ and replacing with $\texttt{x} \mapsto \texttt{1}$ will give us $\texttt{R}'$.

$$\{\mathtt{R} \equiv (\mathtt{x} = \mathtt{y} \wedge \mathtt{x} \mapsto \_\,) \vee (\mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\,)\}$$

$$\{\mathtt{P} \equiv \mathtt{x} \mapsto \_\,\}$$

$$\mathtt{x.val} = 1;$$

$$\{\mathtt{Q} \equiv \mathtt{x} \mapsto 1\}$$

$$\{\mathtt{R}' \equiv (\mathtt{x} = \mathtt{y} \wedge \mathtt{x} \mapsto 1) \vee (\mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \mapsto 1 * \mathtt{y} \mapsto \_\,)\}$$

This example shows that [RAMIFY] is more general than [NONINTER], however it is also more expensive to calculate ramifications so we try to use [NONINTER] for cases with compatible sharing and [RAMIFY] for cases where the heaps may interfere.

Automated verification tools help reduce the burden of correctness on the user by generating proofs. However, these tools themselves may be source of soundness bugs. A bug in a program verifier may be responsible for wrongly verifying several other user programs. Thus it is important that the correctness of these verifiers itself be subject to formal methods. It is extremely hard and challenging to take a large piece of software like a program verifier and certify its correctness. Instead, in this thesis, we propose a novel but practical solution to this problem by certifying the reasoning that is built into the program verifier. Our thesis is that *certified reasoning* helps improve the correctness and expressivity of a program verifier without sacrificing on the efficiency of the verification process.

As an application of *certified reasoning* we present a new decision procedure (`Omega++`) for Presburger arithmetic extended with positive and negative infinity. The correctness proof of `Omega++` is mechanized and verified in the proof assistant Coq. The certified proof is also used to generate OCaml code by extraction from Coq and that code is integrated into a program verifier. By isolating and verifying a key decision procedure that is integrated into a

program verifier, we increase confidence in the correctness of the overall automated verification process. This also reduces the trusted computing base of the overall system. We call this process *certified reasoning* as the reasoning capability of the verifier about a particular domain of interest (Presburger arithmetic extended with positive and negative infinity) has been certified to be correct.

Certified reasoning is not limited to the use of proof assistants only. It is possible to use other more mature tools which can be trusted to certify software built on top of them. As an example of this we show how we can use `SLEEK` entailment checker for separation logic to decide subtyping between traits and mixins in Scala. We extend Scala with a domain specific language for this task and reduce subtyping to checking validity of entailments. This check for validity can be carried out using a trusted component (`SLEEK`), thus giving a higher degree of assurance.

1.1 Thesis Objectives

The key goal of this thesis is to certify programs and proofs that are used in automated verification using separation logic. The overall theme of the effort is to reduce the trusted computing base of a program verifier so that we can check the proof of correctness of its components. This also enables more end to end proofs as we can check not only the program but also the verifier (that is used to certify the program). To lay the foundations for *certified reasoning* we identify three different kinds of components that are part of automated reasoning as typically implemented in a program verifier - certified decision procedures, certified programs and certified proofs.

Firstly we tackle the problem of certified decision procedure. A program

verifier applies Hoare rules and reduces the verification of program to solving some decision problems in a particular domain. Our goal in this respect, is to design and certify the correctness of a decision procedure which is useful for verifying programs of interest. We present a certified decision procedure (`Omega++`) for Presburger arithmetic extended with positive and negative infinities. This domain enables verification of programs with size properties and is found to be quite useful for allowing concise, composable and expressive specifications. The source code is extracted from the certified proof by reflection and the decision procedure is also integrated into a program verifier (`HIP`).

Secondly we present a certified program which enables subtyping between traits and mixins in Scala. Unlike Java, the Scala type system does not respect subtyping between mixin classes. We extend Scala with a domain specific language (`SLEEK DSL`) that enables subtyping with traits and mixins. This check for subtyping is semantic and itself based on deciding entailments in separation logic and is thus called *verified subtyping*.

Thirdly we present a certified proof of compatibility checking in data structures. Separation logic is quite useful for expressing data structures that are based on disjoint heaps, but it is poor at capturing sharing in data structures. We present a mechanism to specify and verify sharing in data structures based on a notion of compatibility. The compatibility checking proof is certified in Coq and that is the key soundness guarantee of our entailment procedure. This procedure is also implemented in a verifier (`HIPComp`) and the certified proof increases our confidence in the semantic formulation of the separation logic extension that requires compatibility checking.

Using *certified reasoning* during automated verification is bound to incur some performance penalty due to the fact that the extracted code may be

inefficient. Another objective of the thesis is to ensure that *certified reasoning* is efficient. To this end we benchmark our approach and present optimizations that ensure that *certified reasoning* is efficient for the verification of programs of interest.

1.2 Contributions of the Thesis

The contributions of the thesis can be divided across the following three vectors:

Certified reasoning with infinity

(Chapter 3, first presented in [108])

- We start from the well-established domains of separation logic [96] and Presburger arithmetic [93] and add two abstract/fictitious/ghost symbols ∞ and $-\infty$, for which we provide a precise, well-defined semantics. Although a seeming-minor addition, these symbols add significantly to the expressivity and power of our logic.

- We use infinities to increase the compositionality of our logic by showing that "lists" and "bounded lists" are equivalent when the bound is ∞. Infinities also add to our specification framework's readability and conciseness. For example, we show that ∞ allows us to drop disjuncts in the specification for code that manipulates a sorted linked list.

- Finally, infinities enable some interesting applications. We apply the notion of quantifier elimination in Presburger arithmetic with infinities to infer pure (non-heap) properties of programs.

- Our major technical advance is the development of `Omega++`, a sound and complete decision procedure for Presburger arithmetic with infinities

(including arbitrary quantifier use). In other words, we do not sacrifice any of the computational advantage normally gained by restricting ourselves to Presburger arithmetic, despite the addition of infinities.

- `Omega++` is written in the Coq theorem prover [1], allowing us to formally certify it (modulo the correctness of Omega itself, which we utilize as our backend). We extract our performance-tuned Coq implementation into OCaml and package it as a library, which we benchmarked using the `HIP/SLEEK` verification toolset [23].

Verified subtyping with traits and mixins

(Chapter 4, first presented in [106])

- We formalize the traits and mixins hierarchies in Scala as required for checking subtyping. We present an approach based on entailment in separation logic to verify subtyping.

- We present a domain specific language (`SLEEK DSL`) which is embedded in Scala and can support verified subtyping with traits and mixins. The `SLEEK DSL` extends the Scala language and allows programmers to insert separation logic entailments in their code.

- We apply our technique to the Scala standard library and verify subtyping in 67% of mixins. To the best of our knowledge this is the first such study of use of traits and mixins with respect to behavior subtyping in Scala. This shows that even though mixins do not enforce subtyping, 67% of usage of mixins is in conformance with behavior subtyping.

Specifying compatible sharing in data structures

(Chapter 5, first presented in [107])

- We provide a specification mechanism to express different kinds of sharing and aliasing scenarios. We enhance automated reasoning in separation logic with new operators ($\curlywedge$, $\wedge$ and $\curlywedge\!\!\!\!\!\!\wedge$). These operators can capture may, must and partial aliasing scenarios.

- We show, how to check for non interference for data structures with sharing. By ensuring that shared parts of data structures are accessed only in a immutable fashion, we can show that the update to the shared parts will not interfere. This enables us to treat them locally and reason with the [NONINTER] rule.

- For interfering data structures we provide a method to do automated ramifications. Ramifications are used to calculate the effect of updating to one part of a shared structure on the other. Automated ramifications allow us to use [RAMIFY] rule during entailment and support unrestricted sharing in data structures.

- Our entailment procedure preserves the principle of local reasoning which ensures scalability during modular verification.

- We have implemented our procedure in a prototype and applied to a small benchmark of programs using data structures with complex sharing.

1.3 Outline

The rest of the thesis is structured as follows. In chapter 2, we survey some related work in the area. In particular, we compare and place our work with respect to related work in certified programs and proofs, program verification, program analysis and code synthesis.

Chapter 3 presents the first instance of *certified reasoning* in the form of a decision procedure for Presburger arithmetic extended with positive and negative infinities. We discuss the challenges in certifying such a decision procedure in Coq and also the performance implications. Our implementation provides an experience report for others interested in enabling *certified reasoning* in their program verifier.

Chapter 4 presents an instance of certified program, a type checker which provides verified subtyping with traits and mixins. We describe our reduction from subtyping to entailment proving and the use of `SLEEK DSL` to support separation logic formulas in Scala. The next two chapters present an instance of certified proof, where the soundness of the compatible sharing technique is established in Coq.

Chapter 5 discusses verification of sharing without interference. We introduce the notion of memory specification with field annotations to handle verification of sharing without interference. The field annotations help in deciding compatible sharing between data structures while the memory specification enables us to support various aliasing and sharing scenarios. Our implementation can do automated verification of several challenging programs with sharing.

Chapter 6 considers the case with interference. To support verification of algorithms with unrestricted updates on shared parts, we use the [RAMIFY] rule. We present a method to automate the verification of ramifications in separation logic using lemmas. Our implementation can do automated verification of several challenging programs including DAG and graph marking algorithms.

Finally, we conclude in chapter 7, with some pointers for future work. In particular, for future work we are interested in reducing the annotation burden on the user for the current technique. Our extended logic captures various

sharing and aliasing scenarios that can be used to infer shape predicates for programs using such structures. Aliasing and deep sharing are challenging problems for current shape analysis tools. We can capture more precise shape predicates using the various sharing operators ($\curlywedge$, $\wedge$ and $\curlywedge\!\!\!\!\curlywedge$) described in this thesis. In addition, this thesis lays the foundations for *certified reasoning*. The eventual goal of *certified reasoning* is to build more trust in formal tools by doing machine checked proofs of their correctness. Other potential application targets for *certified reasoning* include symbolic execution engines [101, 105, 102], program analyzers and model checkers.

Chapter 2

Related Work

There is a rich body of research focused on designing specification languages for verification of object oriented and imperative systems. We first mention some general verification related work and then survey more deeply, research in the area of certified proofs and programs.

The Java Modeling Language (JML) [18, 15] and related tools allow Hoare style pre and post conditions specification for verification of Java programs. In JML, users can annotate classes and methods with specifications that are checked automatically by the verifier. Specifications are written as Java annotations and can be compiled by any Java compiler. The limitation when using the same language for specification and development is that certain properties are hard to specify. In particular, the relationship between object references and aliasing of fields are harder to capture in JML. In comparison our framework is based on separation logic which allows to express properties about shared mutable data structures in a concise and precise manner.

Dafny [97] is recent verifier based on implicit dynamic frames which can be used to check functional properties of .NET programs. Dynamic frames also use the concept of framing which is essential in separation logic. However, in

dynamic frames, there is additional annotation burden on the user to specify the frame for each method. So, in addition to the pre and post conditions the user also needs to provide a modifies condition which captures the parts of heap that are used and changed by the method. Dynamic frames inherit the benefit of local reasoning as the parts unchanged by the method can be framed out. Separation logic based entailment checkers typically have the ability to infer the frame during the entailment. Our proposal builds on the frame inference capability of such an entailment checker. We enhance the frame inference to allow us to do framing in the presence of sharing in heap. Sharing with noninterference can be supported by checking for compatibility prior to framing and then allowing shared parts to be framed as well. Chalice [69] extends Dafny to verify concurrent programs. The benefits of framing carry over to the concurrent case as well. Thus, it is possible to allow different threads to modify same global state as long as they are not interfering.

On the other hand, for separation logic, we have VeriFast [48], a verifier for C and Java programs which supports multiple threads and lock based reasoning. In VeriFast, users provide pre and post specifications, loop invariants, lemmas and proof guidance. The tool then checks automatically if the program can be validated with the given specifications. In contrast, we do not require users to give proof guidance for automated verification. In fact, our entailment procedure is lifted to a set of states to do proof search while checking an entailment. We exploit this capability of our entailment system to implement automated ramifications and compatible sharing without asking the user for guidance during proofs. We support user defined predicates and user specified lemmas that allow flexibility in expressing different shape predicates. Lemmas are also applied automatically in a goal directed fashion to support verification.

Our work builds on the work of verification by separation logic based approaches. The general framework of separation logic is undecidable and Berdine et al. proposed a decidable fragment in [7]. They also developed the first verification tool Smallfoot [9] which is based on symbolic execution [8] of separation logic formulas. Shape predicates supported by Smallfoot are limited to list segments. List segments are not sufficient for verifying properties of programs manipulating more complex shapes like binary search trees and AVL trees. Their work was extended to work with user defined predicates [23] and user specified lemmas [83] and implemented in the HIP/SLEEK [21] verification system.

In HIP/SLEEK, the user needs to only provide annotations for pre and post conditions (along with loop invariants). The tool can do automated verification of programs using complex shapes like binary search trees, AVL trees and red black trees. HIP/SLEEK verifier also supports verification of numerical and set properties. In order to handle sharing in data structures we need to reason over the set of addresses that the predicates corresponding to the data structure hold over. We use the set of addresses to capture the memory region represented by the predicate and check for noninterference over memory regions. Doing so lets us avoid unnecessary unfolding of predicate definitions and reduces the test for compatibility to be a simple syntactic check.

Several further optimizations [24, 25, 65, 100] have been implemented by verifiers using separation logic. The specialization calculus (described in [24]) allows a more efficient way to handle disjunctive predicates that arise while verifying programs with inductive shape definitions. The specialization was essential to verify certain programs and algorithms with large disjuncts in their predicate definition. Our entailment procedure is implemented inside such a system and benefits from all existing and future advancements done for

efficient processing of formulas. These issues are orthogonal to our work which handles sharing and aliasing inside an entailment procedure for separation logic formulas.

Separation logic has also been successfully applied to verify object oriented programs [22, 32] where the principle of abstraction and separation [89] can be used to model inheritance and information hiding [90]. Separation logic based techniques provide a natural way to express abstraction and information hiding found in object oriented programs. These verifiers allow supporting behavior subtyping and other object oriented patterns in the programs.

The success of separation logic for verification of heap manipulating programs is due to the ability to specify disjoint regions of memory and enable local reasoning with the frame rule. In addition, many other formalisms have been proposed for shape analysis of data structures. This includes the work of Moeller and Schwartzbach [79] in Pointer Assertion Logic (PAL) which uses second-order monadic logic for specification. In PAL shape invariants for loop and function calls must be supplied by the programmer and checked by MONA tool. Sagiv et al. [98] presented a parameterized framework for shape analysis using 3-valued logic (TVLA) . In TVLA, based on the properties expected of data structures, programmers must supply a set of predicates to the framework which are then used to analyze that certain shape invariants are maintained.

Kuncak et al. [61] used role type system to specify legal aliasing relationships. Their role system allows the programmer to specify as role constraints, the legal aliasing relationships that define the roles for objects, fields and parameters. An inter-procedural and context sensitive role analysis algorithm can then verify that a program maintains user-supplied role constraints. Hackett and Rugina [40] proposed a region-based shape analysis where shape abstraction is built on region abstraction. Their region analysis

identifies points-to relation between memory regions, while their shape analysis abstracts the state of each individual heap location by keeping track of the reference counts from each region to the tracked location. Most of the above techniques only focus on analysis shape invariants and do not attempt to track the size properties of data structures. In addition, they do not work well with aliasing and sharing in heap. By developing a richer specification mechanism that can capture the different aliasing and sharing situations we allow programmers to verify data structures with complex sharing.

Another related methodology for automated verification is that of model checking. It is a push button technique based on state space search that does not impose a large annotation burden on the user. There are several popular model checkers like PAT [111] and SPIN [103] that have been applied for verification of real time systems (e.g. cardiac pacemaker [99]). The specification language used in model checkers (such as PROMELA for SPIN) is usually not executable although it is possible to extract an implementation [104] from the model after verification. In this thesis, we focus on a specification and verification based system instead as it allows us to capture more expressive properties.

This covers the brief overview of the work related to general verification of programs and shape analysis. In the next four sections, we review and discuss in more detail, closely related research to certification, verification, analysis and synthesis.

2.1 Certified Programs and Proofs

In recent years, there have been several projects that have looked at certifying large software like operating systems, compilers and databases. The CompCert

[70] project has produced a certified compiler for a large subset of the ANSI C language. The correctness of the compiler has been mechanized in the proof assistant Coq. In a recent study of bugs in C compilers [19], CompCert was the only one which did not have any bugs in its implementation. This reinforces the importance of certification and mechanized proofs in programming.

Considerable progress has also been made towards certified operating systems, the seL4 OS kernel [57] was certified using the Isabelle/HOL proof assistant. The entire certification process took several man years to complete. Stewart et. al. [110] have built a certified heap theorem prover based on separation logic. The theorem prover is based on a decision procedure for the list segment fragment and cannot handle other kinds of data structures. Certified programs have also found applications in program analysis [3]. Our proposal of *certified reasoning* is similar in spirit to existing works on certified programs and proofs. However, instead of trying to verify the entire system we take a pragmatic approach and certify different components that form the core of the reasoning required for automated verification. This way we build on top of the success of existing work and extend it to support newer domains (Presburger arithmetic with positive and negative infinity) and applications (sharing in data structures).

2.2 Logics and Verification

In order to understand better the existing work on logics related to sharing let us define some preliminary notions. Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ. Where we define stacks (total mapping between variables and values) and heaps (partial map between locations and values) in the usual way. With this we can define the separation

conjunction $*$ as follows.

$$s, h \models \kappa_1 * \kappa_2 \quad \texttt{iff} \quad \exists h_1, h_2\ h_1 \perp h_2 \text{ and } h = h_1 \cdot h_2 \text{ and } s, h_1 \models \kappa_1 \text{ and } s, h_2 \models \kappa_2$$

Here $h_1 \perp h_2$ represents disjoint heaps h_1 and h_2, while $h_1 \cdot h_2$ is the disjoint union. The $*$ helps in capturing heaps that are not aliased, as an example consider the following pointers x and y which are known to be not aliased.

$$x \mapsto \_ * y \mapsto \_$$

Our sharing and aliasing logic is most closely related to Hobor and Villard [46]. They present the [RAMIFY] rule of separation logic and show how to reason (paper and pen) with graphs, DAGs and overlaid structures using their ramification library. Our work can be seen as a specific instance where we seek to automatically verify programs with sharing that lead to ramifications in the proof. Hobor and Villard use the operator of overlapping conjunction ($\circledast$) to specify shared heaps between two predicates. The $\circledast$ operator can be defined as the follows:

$$s, h \models \kappa_1 \circledast \kappa_2 \quad \texttt{iff} \quad \exists h_1, h_2, h_3\ h_1 \perp h_2 \perp h_3 \text{ and } h = h_1 \cdot h_2 \cdot h_3 \text{ and } s, h_1 \cdot h_3 \models \kappa_1 \text{ and } s, h_2 \cdot h_3 \models \kappa_2$$

The $\circledast$ operator allows us to represent aliased (or overlapping) heap (h_2). This is useful to capture may aliasing, as an example if the pointers x and y may be aliased we can express them as the following formula.

$$x \mapsto \_ \circledast y \mapsto \_$$

On the other hand the $\wedge$ operator is defined as follows.

$$s, h \models \kappa_1 \wedge \kappa_2 \quad \texttt{iff} \quad s, h \models \kappa_1 \text{ and } s, h \models \kappa_2$$

The $\wedge$ operator is helpful in expressing must aliasing in heap. If we know that pointers x and y are aliased, then we can represent them using the following formula.

$$x \mapsto \_ \wedge y \mapsto \_$$

Together these operators can handle a variety of sharing and aliasing scenarios (we defer the discussion of $\mathbb{A}$ operator to chapter 5 as it requires some additional notions). In addition it is useful to define the $-\!\circledast$ operator which can help in capturing state which may be missing some heap. It is defined as follows.

$$\begin{aligned} s, h \models \kappa_1 -\!\circledast \kappa_2 \quad \texttt{iff} \quad & \exists h_1, h_2 \; h_2 = h_1 \cdot h \\ & s, h_1 \models \kappa_1 \text{ and } s, h_2 \models \kappa_2 \end{aligned}$$

As an example consider the following formulas.

$$x \mapsto \_ -\!\circledast (x \mapsto \_ * y \mapsto \_) \equiv y \mapsto \_$$

$$x \mapsto \_ -\!\circledast (x \mapsto \_ \uplus y \mapsto \_) * x \mapsto \_ \equiv (x \mapsto \_ \uplus y \mapsto \_)$$

Table 2.1 provides a list of the various operators used in this thesis. The use of new operators for handling sharing is further motivated by recent discovery of sepish operator by Gardner et al. [36] in the context of verification of JavaScript programs. However, they present only the logic and do not provide an automated system for reasoning. The operator which supports overlapping heaps is notoriously hard to reason with in an automated fashion and thus most tools do not support it.

Table 2.1: List of Operators based on Separation Logic

Name	*Symbol*	*Description*
Separating Conjunction	$*$	disjoint objects
Conjunction	$\wedge$	same objects
Overlaid Conjunction	$\barwedge$	same object disjoint fields
Overlapping Conjunction	$\curlywedge$	overlapping objects
Septraction	$-\!\circledast$	there exists an object inside another

The concept of ramification was introduced by Krishnaswami et al. [60] for verifying event-driven programs. They show how to calculate ramified frames in a domain specific logic with particular semantics. The frame captured in their logic has a ramification operator defined on it. The ramification operator helps to calculate the changes to other parts of structure in presence of sharing.

Other formalisms to reason with shared structures include logics for reasoning with graphs [17] and views [49]. Prior logics for graphs do not focus on spatial aspects of how the graph may be represented in a program. However, they provide mechanisms to express different mathematical graph transformations in a specification language. In contrast, we are focused on automated verification of shared structures as they are represented in program by programmers. This enables us to directly verify code that may be written by users without transforming it to some mathematical model.

The problem of sharing has also been explored in the context of concurrent data structures and objects [31, 112]. The concurrent abstract predicates of Young et al. use the $-\!\circledast$ operator and shared memory regions for verifying concurrent data structures. Our work is influenced by them but for a sequential setting, indeed the notion of self-stable concurrent abstract predicates is analogous to our condition for noninterference. Concurrent abstract predicates

implement control using resource permissions, with the property that the permissions must ensure that a predicate is self-stable: that is, immune from interference from the surrounding environment. Predicates are thus able to specify independent properties about the data, even though the data are shared. The check for noninterference in a sequential setting is much simpler. We use the memory specifications to syntactically check predicates with compatible sharing. Even though, we are reasoning in a sequential setting the use of heap by different predicate may correspond to different access patterns. These access patterns need to be verified to ensure that they do not lead to interference.

Regional logic [5] also uses a notion of set of addresses as footprint of formulas. These regions are used with dynamic frames to enable local reasoning of programs. The frames are captured using read and write effects in regional logic. As an example consider the following assignment.

$$\{\mathtt{x} \neq \mathtt{null}\}$$
$$\mathtt{y} = \mathtt{x.left};$$
$$\{\mathtt{y} = \mathtt{x.left}\}[\mathtt{wr}\ \mathtt{y}]$$

Here $[\mathtt{wr}\ \mathtt{y}]$ captures the write effect of the assignment command. In regional logic one can conjoin an invariant I on both sides if the write effect of the command doesn't affect it. For this example we can use the standard rule of consequence and get the following.

$$\{\mathtt{x} \neq \mathtt{null} \wedge \mathtt{I}\}$$
$$\mathtt{y} = \mathtt{x.left};$$
$$\{\mathtt{y} = \mathtt{x.left} \wedge \mathtt{I}\}[\mathtt{wr}\ \mathtt{y}]$$

In contrast, we allow users to specify predicates with set of addresses. The

set of addresses are checked by the tool to ensure that they cover the footprint of the formulas. Our entailment procedure can do frame inference and we do not need to specify framing conditions with methods. Memory layouts [37] were used by Gast, as a way to formally specify the structure of individual memory blocks. A grammar of memory layouts enable distinguishing between variable, array, or other data structures. When dealing with shared regions of memory, knowing the layout of memory can be quite helpful for reasoning. Our notion of memory specification is not general enough to reason with layouts but we use a Java like model with objects and fields with fixed layout. The overlapping between heap nodes is restricted by this model to be only between corresponding fields that respect the fixed object and field layout.

In the next section, we will discuss some work related to type systems and program analysis which aims to tackle sharing, with special emphasis on heaps and aliasing.

2.3 Program Analysis and Type Systems

In the area of program analysis, the work most closely related to ours is by Lee et al. [68] on overlaid data structures. They show, how to use two complementary static analysis over different shapes and combine them to reason about overlaid structures. Their shape analysis uses the $\wedge$ operator in the abstract state to capture the sharing of heaps in overlaid structures (see fig. 2.1), but they do not provide a general way to reason with shared heaps. They also require the use of ghost instructions and ghost state to transfer information between the sub-analyses. These instructions are used to control communication among the sub-analyses. The purpose of this control is to achieve a high level of efficiency by allowing only necessary information to be

transferred among sub-analyses at as few program points as possible. Their analysis has been used to prove memory safety of the Linux deadline IO scheduler and AFS server.

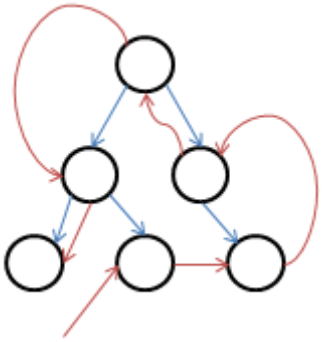


Figure 2.1: Overlaid Data Structure

A separation logic based program analysis has been used to handle non-linear data structures like trees and graphs [20]. In order to handle cycles they keep track of the nodes which are already visited using multi-sets. In our work, with graphs, we do not need to explicitly track nodes or do any other global analysis. This is the key to compositional and modular proof which leads to a more natural and easy specification. Shape analysis for other composite and complex structures has been done through the use of higher-order predicates [6] and abstract modeling of containers [30]. These approaches cannot handle unrestricted sharing and aliasing across containers.

The importance of noninterference was described by Boyland in his work on fractional permissions [14]. Usually noninterference is used in the context of concurrency, we define a similar notion for shared heaps. Fractional permissions can be used to check for noninterference. However, in our work, we use a simpler construct using annotations on fields and memory regions.

The field annotations help in specifying the cases when the memory cells are aliased (overlapped) and the fields are disjoint (overlapping). It is useful to

specify multiple views over the same set of memory cells (e.g. overlaid data structure). The following table shows a comparison between field annotations and fractional permissions.

Table 2.2: Comparing Field Annotations with Fractional Permissions

Annotation	Permission	Fractional Permission
@M	Mutable	1
@I	Immutable	$0 \cdots 1$
@A	Absent	existential

The immutability annotation when applied to field, differs from fractional permission [14], in that it retains full ownership of field, while the read-only property is being imposed. They are helpful for automatic verification of predicates with compatible sharing without the need for solvers with fractional reasoning capability. Reasoning with fractions requires the use of special provers that can handle fractional constraints and many other tools like Chalice [69] and VeriFast [48] provide abstractions to hide fractions from users. Recently a type system for borrowing permissions [81] was proposed by Naden et al. which avoids the use of fractions and relies on access permission based annotations. In the next section, we review some work related to program synthesis.

2.4 Data and Code Synthesis

The problem of developing programs which use sharing in data structures has been considered challenging enough that there has been some work to automatically synthesize correct code for such programs. In [41] Hawkins et al. describe a high level relational algebra based specification mechanism to specify complex sharing, which is then used to generate the physical data

structure that has sharing. They allow users to choose from several base data structures like lists, containers and hash tables that are later fused (composed) together to create composite structures with sharing to enable better and more efficient retrieval.

They extend their approach in [42] to generate data representation as well as the code to query the data structure in the form of relational queries. In their paper they identify the challenge of specifying invariants on multiple overlapping data structures and mention that existing verification techniques are insufficient to reason about them. Our work can be seen as an attempt to provide a specification and verification mechanism for such shared structures.

Synthesis based approaches [41, 42] restrict the potential use of the generated data structure. The only way to use the structure is by querying over a fix set of generated access functions. In contrast, verification of such complex structures allows us to reason with arbitrary code that may manipulate these structures. In addition synthesis can help in generating new code and data structures but is not useful for verifying the existing code which uses non-trivial sharing. Moreover, the use of relational operators for querying though intuitive and declarative is already difficult to reason with when used along with concurrent structures [43] and requires separate lock placements.

In this thesis, we take a verification based perspective on the problem and provide mechanisms to specify precisely the sharing in data structures. In comparison to prior work we present the first automated verification method to handle different kinds (may, must and partial) of sharing in data structures. We have implemented our approach in a prototype and can verify several challenging new and existing examples of programs using such sharing in data structures.

Chapter 3

Certified Reasoning with Infinity

We demonstrate how infinities improve the expressivity, power, readability, conciseness, and compositionality of a program logic. We prove that adding infinities to Presburger arithmetic enables these improvements without sacrificing decidability. We develop Omega++, a Coq-certified decision procedure for Presburger arithmetic with infinity and benchmark its performance. Both the program and proof of Omega++ are parameterized over user-selected semantics for the indeterminate terms (such as $0 * \infty$).

3.1 Introduction

Formal software analysis and verification frameworks benefit from expressive, compositional, decidable, and readable specification mechanisms. Of course, these goals often conflict with each other: for example, it is easy to add expressivity if one is willing to give up decidability! Happily, we have found a free lunch: by adding the notion of "infinity" to the specification language we can usefully add to the expressivity, readability, and compositionality of our specifications while maintaining their decidability.

Specifically, we start from the well-established domains of separation logic [96] and Presburger arithmetic [93] and add two abstract/fictitious/ghost symbols ∞ and $-\infty$, for which we support a precise, well-defined semantics. Although a seeming-minor addition, these symbols add significantly to the expressivity and power of our logic.

In section 3.2.3 we use infinities to increase the compositionality of our logic by showing that "lists" and "bounded lists" are equivalent when the bound is ∞. Moreover, in section 3.2.4, we use ∞ to mix notions of partial and total correctness within a logic.

Infinities also add to our specification framework's readability and conciseness. For example, we will see in section 3.2.2 that ∞ allows us to drop disjuncts in the specification for code that manipulates a sorted linked list.

Finally, infinities enable some interesting applications. In section 3.2.5 we apply the notion of quantifier elimination in Presburger arithmetic with infinities to infer pure (non-heap) properties of programs.

All of the previous gains are worthy in their own right, but our major technical advance is the development of Omega++, a sound and complete decision procedure for Presburger arithmetic with infinities (including arbitrary quantifier use). In other words, we do not sacrifice any of the computational advantage normally gained by restricting ourselves to Presburger arithmetic, despite the addition of infinities. We call our tool "Omega++" both to acknowledge the importance of the underlying Presburger solver Omega [55] and because we believe we have modestly incremented its utility.

Omega++ is written in Gallina, the specification language of Coq [1], allowing us to formally certify it (modulo the correctness of Omega itself, which we utilize as our backend). We extract our performance-tuned Gallina into OCaml and package it as a library, which we have benchmarked using the

HIP/SLEEK verification toolset [23].

One notable technical feature of Omega++ is that it can handle several semantic variants of Presburger arithmetic with infinity. For example, Presburger arithmetic usually admits multiplication by a constant as a notational convenience, *e.g.* $3 \cdot x \stackrel{\text{def}}{=} x + x + x$. This obvious-seeming convenience becomes a little less obvious when one adds infinities: what is $0 \cdot \infty$? Mathematical sophisticates can—*and do*—disagree: some prefer 0 as a convention in certain contexts (including, reasonably, ours) [78], while others prefer the result to be undefined due to the indeterminate status of the corresponding limit forms [59]. When possible, Omega++ takes an agnostic approach to such disagreements by allowing the user to specify the semantics of some subtle cases. Omega++ is thus a certified compiler from a *set* of related source languages (Presburger arithmetics with infinities) to a fixed, well-understood target (vanilla Presburger).

The rest of the chapter is structured as follows. In section 3.2 we demonstrate the value of adding infinities to Presburger arithmetic. In section 3.3 we develop a precise formal semantics for Presburger arithmetic with infinities. In section 3.4 we describe the algorithm for the decision procedure and state its key properties, all of which have been verified in Coq for our optimized implementation. In section 3.5 we discuss the implementation itself and in section 3.6 we benchmark its performance. Finally, in section 3.7 we describe some related work and conclude.

Omega++ is available for download and experimentation here:
`http://loris-7.ddns.comp.nus.edu.sg/~project/SLPAInf/`

3.2 Motivation

In this section, we highlight the benefits of augmenting a specification logic with infinities. For consistency we focus on separation logic [47, 96] but other specification mechanisms which rely on Presburger arithmetic can enjoy similar benefits.

3.2.1 Orientation

Our flavor of separation logic is based on the HIP/SLEEK system [23], letting us run tests and benchmarks with a state-of-the-art verification toolchain. Methods are specified with a pair of pre- and postcondition (Φ_{pr}, Φ_{po}), with the keyword `res` allowed in the Φ_{po} to refer to the return value. We have enhanced the logic to allow the symbols ∞ and $-\infty$ where it would normally require integers; we also allow quantification over infinities.

From a systems perspective, our setup is sketched in figure 3.1. First, entailment between separation logic formulae with infinities in HIP/SLEEK is reduced (*à la* Chin *et al.* [23]) to entailment between numeric formulae in Presburger arithmetic with infinities (PAInf). Our main contribution is the next phase, detailed in section 3.4, in which we translate PAInf to vanilla Presburger arithmetic (PA). Finally, we discharge PA proof obligations with Omega. There are other combinations of separation logic with extensions of PA (such as sets/multisets) that can be used to enhance the specification and verification process. We discuss them in section 3.7.2 with related work.

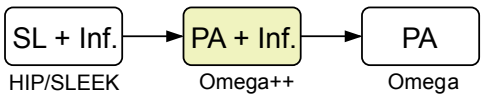


Figure 3.1: Our setup: SL + Inf to PA

3.2.2 Infinities enable Concise Specifications

Let's start to see what infinities can buy us! Consider a simple program that inserts a new node into a sorted linked list, whose nodes are defined as follows:

$$\texttt{data node}\,\{\texttt{int val; node next;}\}$$

The data field `val` stores numerical information and the pointer field `next` points to the subsequent node in the structure. Consider two alternative inductive predicates that characterize sortedness using only a single numeric parameter[1] that describes the list's minimum value:

Scenario 1 - no infinity enhancement:

$$\texttt{sorted\_ll}\langle\texttt{root},\texttt{min}\rangle \equiv \texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{null}\rangle$$
$$\vee\ \exists\,\texttt{q},\ \texttt{mtail}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{q}\rangle * \texttt{sorted\_ll}\langle\texttt{q},\texttt{mtail}\rangle \wedge \texttt{min}{\leq}\texttt{mtail})$$

Scenario 2 - with infinity enhancement:

$$\texttt{sorted\_ll}\langle\texttt{root},\texttt{min}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{min}{=}\infty)$$
$$\vee\ \exists\,\texttt{q},\ \texttt{mtail}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\texttt{min},\texttt{q}\rangle * \texttt{sorted\_ll}\langle\texttt{q},\texttt{mtail}\rangle \wedge \texttt{min}{\leq}\texttt{mtail})$$

The base case of *Scenario 1* denotes a singleton, while its inductive case describes a linked list of length at least two. Though usable, this definition has a frustrating shortcoming: it cannot handle empty linked lists, since such lists do not have a finite minimum value. In contrast, *Scenario 2* handles the empty list gracefully since the minimum of an empty list can be defined to be just ∞!

The code for `insert` is in figure 3.2. Parameter `x` points to a sorted linked list, while `y` is the data node we wish to insert (preserving sortedness). Notice

[1] Note that there are other ways of specifying sortedness, such as through the use of multi-set, that may also capture stronger properties, like content preservation. However, they may require more complex provers in their reasoning.

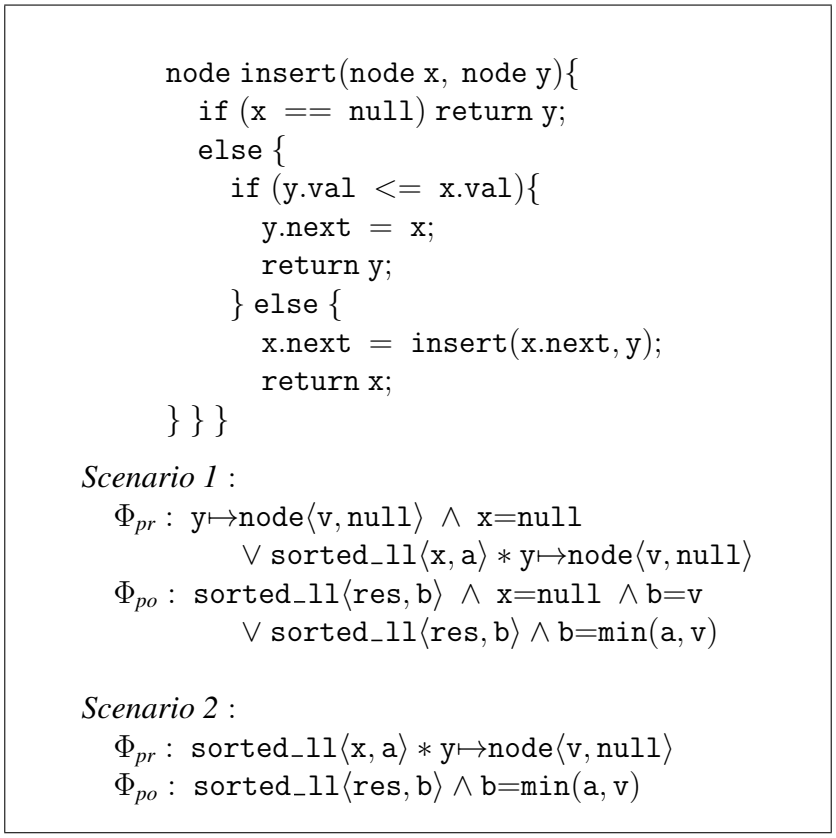


Figure 3.2: Two pre-/postconditions for insertion into a sorted linked list

that the pre/post specifications in Scenario 1 require disjunctions to separate the cases when x is empty and nonempty, whereas Scenario 2 handles both cases uniformly. Infinities thus enable more *concise* and *readable* (easy to maintain) specifications.

3.2.3 Infinities increase Compositionality

Consider this definition for an n-node linked list whose values are bounded by b:

$$\begin{aligned}\texttt{llB}\langle\texttt{root},\texttt{n},\texttt{b}\rangle \equiv\ & (\texttt{root}{=}\texttt{null}\ \wedge\ \texttt{n}=0)\\ & \vee(\,\exists\,\texttt{q},\texttt{v}\cdot\texttt{root}{\mapsto}\texttt{node}\langle\texttt{v},\texttt{q}\rangle * \texttt{llB}\langle\texttt{q},\texttt{n}-1,\texttt{b}\rangle\ \wedge\ \texttt{v}\leq\texttt{b})\end{aligned}$$

Suppose we have a function f which uses this definition in its precondition:

$$\Phi_{pr}:\ \texttt{llB}\langle\texttt{x},\texttt{n},\texttt{m}\rangle * \ldots$$

where x points to a linked list bounded by m. Next, suppose we call f from a program point where the only available information involves the shape and length of a linked list x (that is, we have no information about its bound), *e.g.* we satisfy the predicate $\texttt{ll}\langle\texttt{x},\texttt{n}\rangle$ as defined below:

$$\texttt{ll}\langle\texttt{root},\texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge\texttt{n}{=}0)\vee\ \exists\,\texttt{q}\cdot(\texttt{root}{\mapsto}\texttt{node}\langle\_,\texttt{q}\rangle * \texttt{ll}\langle\texttt{q},\texttt{n}-1\rangle)$$

With infinities this is easy: just instantiate m to ∞ since

$$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle \leftrightarrow \texttt{llB}\langle\texttt{x},\texttt{n},\infty\rangle$$

Without infinities, however, this is not so easy since we must first determine an appropriate bound for x's values. Thus, infinities increase the *compositionality* of our logic, which in turn improves the reusability and conciseness of our specifications.

3.2.4 Infinities support (Non-)Termination Reasoning

Le *et al.* [66] developed a technique to reason about termination and non-termination with a resource constraint $\mathtt{RC}\langle \mathtt{min}, \mathtt{max} \rangle$ that tracks the minimum and maximum permitted execution steps. Using Presburger arithmetic with infinity, terminating programs are modeled by $\mathtt{RC}\langle \_, \mathtt{max} \rangle \ \wedge \ \mathtt{max} < \infty$ and non-terminating programs are captured by $\mathtt{RC}\langle \infty, \infty \rangle$. Le *et al.* [66] evaluated the semantics of non-termination reasoning with the help of Omega++.

As an example consider the following predicate definitions for a finite list segment and a circular list.

$$\begin{aligned} \mathtt{ls}\langle \mathtt{root}, \mathtt{p}, \mathtt{n} \rangle &\equiv (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0) \\ &\quad \vee \exists\, \mathtt{q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \_, \mathtt{q} \rangle * \mathtt{ls}\langle \mathtt{q}, \mathtt{p}, \mathtt{n}-1 \rangle \wedge \mathtt{root} \neq \mathtt{p}) \\ \mathtt{cll}\langle \mathtt{root}, \mathtt{n} \rangle &\equiv \exists\, \mathtt{q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \_, \mathtt{q} \rangle * \mathtt{ls}\langle \mathtt{q}, \mathtt{root}, \mathtt{n}-1 \rangle) \end{aligned}$$

Figure 3.3 demonstrates these resource constraints on a `length` function for linked lists. We show two specifications: the first shows that `length` terminates on finite lists `ls`, and the second shows that `length` diverges on circular lists `cll`.

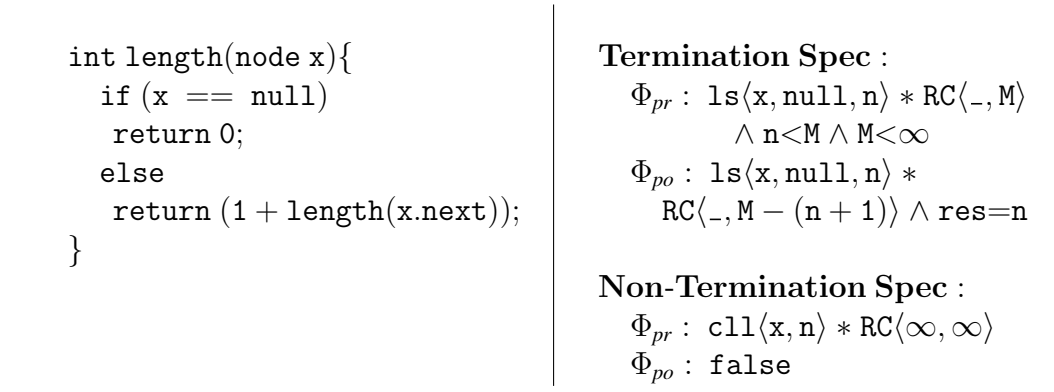


Figure 3.3: `length` terminates on proper lists and diverges on cyclic lists

3.2.5 Infinities support Analysis via Quantifier Elimination

Algorithmic quantifier elimination (QE) is a powerful technique for decision procedures in symbolic logic [52]. Kapur highlights the importance of geometric QE heuristics for the case of generating program invariants, distinguishing between octagonal and max-plus invariants [53]. While Kapur exploits the structure of verification conditions generated from numerical programs, we focus on generating inductive invariants for programs manipulating dynamically allocated data structures.

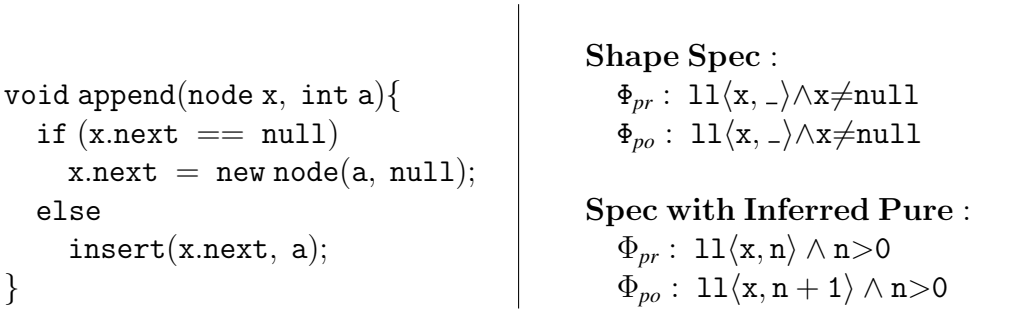


Figure 3.4: Pure Specification Inferred from PAInf QE

Consider the code in figure 3.4, which appends a node to the end of an acyclic linked list. The first specification only captures shape; it would be useful to infer size properties as well. We can do so by using PAInf-based QE to support inference of octagonal constraints with infinities in the presence of heap-

based verification. Forward reasoning generates relational obligations which are then discharged by QE over PAInf, leading to the second specification with numeric properties. In addition to the octagonal constraints we can also infer constraints over min and max relations.

3.3 Syntax and Parameterized Semantics

There are several benefits of adding the notion of infinity to a program logic. However, due to the presence of certain terms like $(\infty - \infty)$, it is an interesting problem to define the correct (or rather desired) semantics. We will now proceed to a formal discussion of Presburger arithmetic with infinity.

Our constraint language extends Presburger arithmetic with two abstract symbols designating positive (∞) infinity and negative ($-\infty$) infinity. The language is detailed in figure 3.5. However, we would like to make some extra notes. First, we use a type based approach to distinguish between the domain of variables. The notation $w : \tau$ denotes that the variable w is of type τ. Second, for performance reasons that are explained in section 3.5 we do not aim for a minimal input constraint language. That is the reason why the input language also supports min and max constraints. The min and max constraints in the input language are automatically translated to $\text{min}_=$ and $\text{max}_=$ (using the following rewriting rules $\pi \rightsquigarrow [v/\max(a_1,a_2)]\pi \wedge \text{max}_=(v, a_1, a_2)$ and $\pi \rightsquigarrow [v/\min(a_1,a_2)]\pi \wedge \text{min}_=(v, a_1, a_2)$).

Next, we present the parameterized semantic model for PAInf and establish theorems and lemmas that show the correctness of our decision procedure. All theorems and lemmas in this chapter are machine checked in Coq. Parameters are introduced to adapt different possible ways of handling the tricky parts of PAInf, such as the terms $(\infty - \infty)$ and $(0 \times \infty)$. Since our semantics is

$$
\begin{array}{lll}
\pi & ::= & \beta \mid \neg\pi \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \pi_1 \rightarrow \pi_2 \mid \exists(w:\tau)\cdot\pi \mid \forall(w:\tau)\cdot\pi \\
\beta & ::= & \texttt{true} \mid \texttt{false} \mid a_1 < a_2 \mid a_1 \leq a_2 \mid a_1 = a_2 \mid a_1 \neq a_2 \\
 & & \mid a_1 \geq a_2 \mid a_1 > a_2 \\
a & ::= & k \mid v \mid c \times a \mid a_1 + a_2 \mid -a \mid a_1 - a_2 \mid \max(a_1,a_2) \mid \min(a_1,a_2) \\
k & ::= & c \mid \infty \mid -\infty
\end{array}
$$

where v, w are variable names; c is an integer constant

Figure 3.5: PAInf: Input Constraint Language

parameterized, all procedures, theorems and lemmas based on the semantics are also parameterized. We start by defining an *environment* to map variables to values.

Definition 1. *An environment for a universe τ of concrete values is a function $\phi_\tau : V \rightarrow \tau$ from the set of variables V to τ. For such a ϕ_τ, we denote by $\phi_\tau[x \mapsto a]$ the function which maps x to a and any other variable y to $\phi_\tau(y)$.*

We define the semantics of arithmetic operations and relations for PAInf formally in figure 3.6 and 3.7 (denoted by $[\![\beta]\!]_{\mathbb{Z}_\infty}$). The subscript of $[\![]\!]$ denotes the domain of constants. $\mathbb{Z}_\infty$ means $\mathbb{Z} \cup \{\infty, -\infty\}$. By analogy, $[\![\beta]\!]_{\mathbb{Z}}$ means the domain is $\mathbb{Z}$. With these definitions one can compute every atomic term into a truth value with respect to an environment ϕ_τ and domain of constants η as described in figure 3.8, and denoted by $\textsc{Eval}^{\eta}_{\phi_\tau}$.

We define the satisfaction relation $\phi_\tau \models^{\text{sat}}_{\eta} \pi$ and dissatisfaction relation $\phi_\tau \models^{\text{dst}}_{\eta} \pi$ (in figure 3.9) for each logical formula π over the environment ϕ_τ and domain of constants η by structural induction on π. Sometimes, a formula π can neither be satisfied nor be dissatisfied. In that case, we say π is undetermined, which can be presented as $\phi_\tau \models^{\text{udt}}_{\eta} \pi$. We define two distinct relations for satisfaction and dissatisfaction as we support both two-valued and

[**ADDITION**]

$$[\![k_1 + k_2]\!]_{\mathbb{Z}_\infty} \stackrel{\text{def}}{=} \begin{cases} \bot & k_1 \text{ or } k_2 \text{ is } \bot \\ \bot & k_1 = \infty, k_2 = -\infty \\ \bot & k_1 = -\infty, k_2 = \infty \\ \infty & k_1 \text{ or } k_2 \text{ is } \infty, \text{ and neither is } -\infty \\ -\infty & k_2 \text{ or } k_2 \text{ is } -\infty, \text{ and neither is } \infty \\ [\![k_1 + k_2]\!]_{\mathbb{Z}} & k_1 \text{ and } k_2 \text{ are finite} \end{cases}$$

[**LESS−THAN−EQ**]

$$[\![k_1 \leq k_2]\!]_{\mathbb{Z}_\infty} \stackrel{\text{def}}{=} \begin{cases} \mathtt{F/U} & k_1 \text{ or } k_2 \text{ is } \bot \\ \mathtt{T} & k_2 = \infty \\ \mathtt{T} & k_1 = -\infty \\ \mathtt{T} & k_1 = k_2 = \infty \\ \mathtt{T} & k_1 = k_2 = -\infty \\ \mathtt{F} & k_1 = \infty, k_2 \neq \infty \\ \mathtt{F} & k_1 \neq -\infty, k_2 = -\infty \\ [\![k_1 \leq k_2]\!]_{\mathbb{Z}} & k_1 \text{ and } k_2 \text{ are finite} \end{cases}$$

[**IDENTITY**]

$$[\![k]\!]_{\mathbb{Z}_\infty} \stackrel{\text{def}}{=} k$$

[**NEGATION**]

$$[\![-k]\!]_{\mathbb{Z}_\infty} \stackrel{\text{def}}{=} \begin{cases} \bot & k = \bot \\ \infty & k = -\infty \\ -\infty & k = \infty \\ [\![-k]\!]_{\mathbb{Z}} & k \text{ is finite} \end{cases}$$

Figure 3.6: Main Operations and Relations in $\mathbb{Z}_\infty$

three-valued logic. In case of three-valued logic a formula can be neither satisfied nor dissatisfied (undetermined).

Much of the semantics for PAInf is "as you might expect". For example, when all the values are finite, all of the operations and relations behave the same way they would in PA. On the other hand, any finite value plus ∞ equals ∞ and any finite value plus $-\infty$ equals $-\infty$. It is trickier to figure out what to do with the sum of ∞ and $-\infty$; we treat this as a meaningless value (much like the "value" of $\frac{0}{0}$ in the reals) denoted by "$\bot$". If ∞ and $-\infty$ were actually inverses,

[OTHER–OPERATIONS–AND–RELATIONS]

$$
\begin{aligned}
[\![0 \times k]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} \begin{cases} 0 & k \text{ is finite} \\ 0/\bot/k & k \text{ is not finite} \end{cases} \\
[\![c \times k]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} \begin{cases} [\![0 \times k]\!]_{\mathbb{Z}_\infty} & c = 0 \\ k & c = 1 \\ [\![k + (c-1) \times k]\!]_{\mathbb{Z}_\infty} & c > 1 \\ [\![-((-c) \times k)]\!]_{\mathbb{Z}_\infty} & c < 0 \end{cases} \\
[\![k_1 \geq k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} [\![k_2 \leq k_1]\!]_{\mathbb{Z}_\infty} \\
[\![k_1 > k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} [\![k_1 \geq k_2]\!]_{\mathbb{Z}_\infty} \wedge [\![k_1 \neq k_2]\!]_{\mathbb{Z}_\infty} \\
[\![k_1 \neq k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} \neg [\![k_1 = k_2]\!]_{\mathbb{Z}_\infty} \\
[\![k_1 = k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} [\![k_1 \leq k_2]\!]_{\mathbb{Z}_\infty} \wedge [\![k_2 \leq k_1]\!]_{\mathbb{Z}_\infty} \\
[\![k_1 - k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} [\![k_1 + (-k_2)]\!]_{\mathbb{Z}_\infty} \\
[\![k_1 < k_2]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} [\![k_1 \leq k_2]\!]_{\mathbb{Z}_\infty} \wedge [\![k_1 \neq k_2]\!]_{\mathbb{Z}_\infty} \\
[\![\max_=(k_1, k_2, k_3)]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} ([\![k_1 = k_2]\!]_{\mathbb{Z}_\infty} \wedge [\![k_3 \leq k_2]\!]_{\mathbb{Z}_\infty}) \\
&\quad \vee ([\![k_1 = k_3]\!]_{\mathbb{Z}_\infty} \wedge [\![k_2 \leq k_3]\!]_{\mathbb{Z}_\infty}) \\
[\![\min_=(k_1, k_2, k_3)]\!]_{\mathbb{Z}_\infty} &\stackrel{\text{def}}{=} ([\![k_1 = k_2]\!]_{\mathbb{Z}_\infty} \wedge [\![k_2 \leq k_3]\!]_{\mathbb{Z}_\infty}) \\
&\quad \vee ([\![k_1 = k_3]\!]_{\mathbb{Z}_\infty} \wedge [\![k_3 \leq k_2]\!]_{\mathbb{Z}_\infty})
\end{aligned}
$$

Figure 3.7: Other Operations and Relations in $\mathbb{Z}_\infty$

we would need to admit the following whopper:

$$0 \;=\; \infty + -\infty \;=\; \infty + (-\infty + 1) \;=\; (\infty + -\infty) + 1 \;=\; 1$$

In fact there is no perfect solution, since it is impossible to add a finite number of symbols to $\mathbb{Z}$ while remaining a group. Lasaruk and Sturm [64] propose dodging part of this problem by using only a single value for both positive and negative infinity, which is both greater than *and* less than all finite values. This approach ensures that every sum is defined, although ∞ still does not have an inverse and you lose antisymmetry for $\leq$. We find the notion of a single infinity to be too restrictive as it prohibits us from expressing some of the motivating examples from section 3.2.

[**ARITH−EVAL**]

$$\textsc{Eval}^{\eta}_{\phi_\tau}(k) \stackrel{\text{def}}{=} [\![k]\!]_\eta \qquad \textsc{Eval}^{\eta}_{\phi_\tau}(v) \stackrel{\text{def}}{=} [\![\phi_\tau(v)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(c \times a) \stackrel{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(c) \times \textsc{Eval}^{\eta}_{\phi_\tau}(a)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 + a_2) \stackrel{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) + \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 - a_2) \stackrel{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) - \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(-a) \stackrel{\text{def}}{=} [\![-\textsc{Eval}^{\eta}_{\phi_\tau}(a)]\!]_\eta$$

[**BOOLEAN−EVAL**]

$$\textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{true}) \stackrel{\text{def}}{=} \texttt{T} \quad \textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{false}) \stackrel{\text{def}}{=} \texttt{F} \quad \textsc{Eval}^{\eta}_{\phi_\tau}(\texttt{undefined}) \stackrel{\text{def}}{=} \texttt{U}$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(a_1 \circ a_2) \stackrel{\text{def}}{=} [\![\textsc{Eval}^{\eta}_{\phi_\tau}(a_1) \circ \textsc{Eval}^{\eta}_{\phi_\tau}(a_2)]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(\max_=(a_1, a_2, a_3)) \stackrel{\text{def}}{=}$$
$$[\![\max_=(\textsc{Eval}^{\eta}_{\phi_\tau}(a_1), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2))]\!]_\eta$$
$$\textsc{Eval}^{\eta}_{\phi_\tau}(\min_=(a_1, a_2, a_3)) \stackrel{\text{def}}{=}$$
$$[\![\min_=(\textsc{Eval}^{\eta}_{\phi_\tau}(a_1), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2), \textsc{Eval}^{\eta}_{\phi_\tau}(a_2))]\!]_\eta$$

where $\circ$ above means one of $\leq, \geq, <, >, =, \neq$.

Figure 3.8: Evaluations on atomic terms

In addition to the issues with using a single infinity, handling comparisons with $\perp$ is another challenge. A possible solution is treating all comparisons with $\perp$ as false. This is reasonable but not perfect. For example, in this context, it is not the case that $x > y$ is equivalent to $\neg(x \leq y)$ when x or y are $\perp$. Interestingly this is the choice made by IEEE floating point standard [2]. Another possibility is to use a three-valued logic and treat any comparison with $\perp$ as the "third unknown value". There are several three-valued logics studied in the literature [10]. We use Kleene's weak three-valued logic which interprets the unknown value as "Error" and propagates it to the entire formula.

The truth tables for this three-valued logic are shown in figure 3.10. In three-valued logic, when x or y are $\perp$, $x > y$ and $\neg(x \leq y)$ are equivalent. In Omega++, users can choose between a two-valued or three-valued logic, which is indicated in [**LESS−THAN−EQ**] rule given in figure 3.6. Note that in three-

$\phi_\tau \models^{\text{sat}}_\eta \beta$	iff $\textsc{Eval}^\eta_{\phi_\tau}(\beta)$ is T.
$\phi_\tau \models^{\text{sat}}_\eta \neg\pi$	iff $\phi_\tau \models^{\text{dst}}_\eta \pi$ holds.
$\phi_\tau \models^{\text{sat}}_\eta \pi_1 \wedge \pi_2$	iff both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{sat}}_\eta \pi_1 \vee \pi_2$	iff both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{sat}}_\eta \pi_1 \rightarrow \pi_2$	iff both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{sat}}_\eta \exists(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\text{sat}}_\eta \pi$ holds for some $k \in \tau$, and forall all $k \in \tau$, either $\phi_\tau[w \mapsto k] \models^{\text{sat}}_\eta \pi$ or $\phi_\tau[w \mapsto k] \models^{\text{dst}}_\eta \pi$ holds.
$\phi_\tau \models^{\text{sat}}_\eta \forall(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\text{sat}}_\eta \pi$ holds for all $k \in \tau$
$\phi_\tau \models^{\text{dst}}_\eta \beta$	iff $\textsc{Eval}^\eta_{\phi_\tau}(\beta)$ is F.
$\phi_\tau \models^{\text{dst}}_\eta \neg\pi$	iff $\phi_\tau \models^{\text{sat}}_\eta \pi$ holds.
$\phi_\tau \models^{\text{dst}}_\eta \pi_1 \wedge \pi_2$	iff both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds, or both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{sat}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{dst}}_\eta \pi_1 \vee \pi_2$	iff both $\phi_\tau \models^{\text{dst}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{dst}}_\eta \pi_1 \rightarrow \pi_2$	iff both $\phi_\tau \models^{\text{sat}}_\eta \pi_1$ and $\phi_\tau \models^{\text{dst}}_\eta \pi_2$ holds.
$\phi_\tau \models^{\text{dst}}_\eta \exists(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\text{dst}}_\eta \pi$ holds for all $k \in \tau$
$\phi_\tau \models^{\text{dst}}_\eta \forall(w:\tau)\cdot\pi$	iff $\phi_\tau[w \mapsto k] \models^{\text{dst}}_\eta \pi$ holds for some $k \in \tau$, and forall all $k \in \tau$, either $\phi_\tau[w \mapsto k] \models^{\text{sat}}_\eta \pi$ or $\phi_\tau[w \mapsto k] \models^{\text{dst}}_\eta \pi$ holds.
$\phi_\tau \models^{\text{udt}}_\eta \pi$	iff neither $\phi_\tau \models^{\text{sat}}_\eta \pi$ or $\phi_\tau \models^{\text{dst}}_\eta \pi$ holds.

Figure 3.9: Definition of satisfaction relation

valued logic, according to the relation definition in figure 3.9, formulae like $\bot < 0$ are neither satisfied nor dissatisfied.

The definition of multiplication in the presence of infinities ($0 \times \infty$) can also be selected by the user as shown in figure 3.6. There are three possible choices for defining $0 \times \infty$: 0, $\bot$ and ∞. For each of these options we can choose a two-valued or three-valued logic, thus Omega++ supports six

P	$\neg P$
true	false
false	true
undefined	undefined

P	Q	$P \vee Q$
true	true	true
true	false	true
true	undefined	undefined
false	true	true
false	false	false
false	undefined	undefined
undefined	true	undefined
undefined	false	undefined
undefined	undefined	undefined

P	Q	$P \wedge Q$
true	true	true
true	false	false
true	undefined	undefined
false	true	false
false	false	false
false	undefined	undefined
undefined	true	undefined
undefined	false	undefined
undefined	undefined	undefined

Figure 3.10: Truth Tables for Three-valued Logic

different customized semantics in total. As described in section 3.6, for our experiments we use the semantics with three-valued logic and $0 \times \infty \stackrel{\text{def}}{=} 0$. However, in general any of the six customized semantics can be used as the decision procedure is parameterized over these choices and our certified proof guarantees that all choices are sound, complete and decidable.

In order to match the intuition of user, by design, most valid formulae in PA remain so in our semantics for PAInf, just as most invalid formulae in PA are

still invalid in PAInf. Here are two short examples which are valid in both (if you drop the universe of quantification as you move from PAInf to PA):

$$\forall(x : \mathbb{Z}_\infty)\cdot\exists(y : \mathbb{Z}_\infty)\cdot x \leq y \qquad \forall(x : \mathbb{Z}_\infty)\cdot\forall(y : \mathbb{Z}_\infty)\cdot x{+}1 = y{+}1 \rightarrow x = y$$

However, there are differences. This formula is valid in PA but invalid in PAInf:

$$\forall(x : \mathbb{Z}_\infty) \cdot \exists(y : \mathbb{Z}_\infty) \cdot x + y = 0$$

The previous formula is false in PAInf when $x = \infty$. More generally, although $\mathbb{Z}_\infty$ is not a group, it still has many useful algebraic properties, such as the following.

Lemma 3.3.1. $+$ **is Associative** $[\![(a + b) + c]\!]_{\mathbb{Z}_\infty}$ *and* $[\![a + (b + c)]\!]_{\mathbb{Z}_\infty}$ *are equal or both undefined.*

Lemma 3.3.2. $+$ **is Commutative** $[\![a + b]\!]_{\mathbb{Z}_\infty}$ *and* $[\![b + a]\!]_{\mathbb{Z}_\infty}$ *are equal or both undefined.*

Lemma 3.3.3. 0 is the Additive Identity $[\![a + 0]\!]_{\mathbb{Z}_\infty}$ *and* a *are equal for all defined* a*.*

Lemma 3.3.4. $+$ **is Monotonic** *If* $[\![a \leq b]\!]_{\mathbb{Z}_\infty}$ *is* `T` *and if both* $[\![a + c]\!]_{\mathbb{Z}_\infty}$ *and* $[\![b + c]\!]_{\mathbb{Z}_\infty}$ *are defined, then* $[\![a + c \leq b + c]\!]_{\mathbb{Z}_\infty}$ *is also* `T`*.*

3.4 Reasoning with Infinity

For the following discussion we assume the existence of a solver for Presburger arithmetic (such as Omega [55]). Our focus is to automate the reasoning of ghost infinities by leveraging on existing solvers.

Note that $v \in \mathbb{Z}_\infty$, is the same as, $v \in \mathbb{Z} \vee v = \infty \vee v = -\infty$. This fact can be used to give a quantifier elimination procedure for PAInf as shown in figure 3.11. However, using this approach naively leads to an explosion in the size of formulae to be checked. As an example, consider the following formula,

$$\forall \mathtt{x}, \mathtt{y}, \mathtt{z} \cdot (\mathtt{z}{=}\infty \wedge \mathtt{y}{=}\mathtt{x} + \mathtt{z} \wedge \mathtt{x}{<}\infty)$$

Using the [FORALL–INF] rule to eliminate the three quantified variables ($\mathtt{x}, \mathtt{y}$ and $\mathtt{z}$), leads to 3^3 (= 27) constraints. To avoid this problem, we support both kinds of quantifiers ($\exists(w : \mathbb{Z})$ and $\exists(w : \mathbb{Z}_\infty)$) in the implementation. This allows for a more efficient quantifier elimination as variables with finite domain do not give rise to new disjunctions in formulae. Since, infinity is added as a ghost constant only in the specification logic, all program variables are still in finite domain. Supporting two kinds of quantifiers matches nicely with the distinction between the domain of specification variables ($\mathbb{Z}_\infty$) and program variables ($\mathbb{Z}$). In section 3.6 we compare our system with an implementation of PAI from [64] and demonstrate the effectiveness of using our procedure for quantifier elimination.

[EXISTS–INF]	[FORALL–INF]
$\exists(w : \mathbb{Z}_\infty) \cdot \pi \rightsquigarrow \exists(w : \mathbb{Z}) \cdot \pi$	$\forall(w : \mathbb{Z}_\infty) \cdot \pi \rightsquigarrow \forall(w : \mathbb{Z}) \cdot \pi$
$\vee [\infty/w]\pi$	$\wedge [\infty/w]\pi$
$\vee [-\infty/w]\pi$	$\wedge [-\infty/w]\pi$

Figure 3.11: PAInf: Quantifier Elimination (INF-TRANS)

For checking satisfiability in the PAInf we use the algorithm shown in figure 3.12. We denote the procedure for satisfiability checking as $SAT(\pi)$. It has the following four steps: (i) first we eliminate the quantifiers starting with

the innermost quantifier, (ii) next we apply a normalization which detects tautologies and contradictions in constraints using infinity, (iii) then we eliminate min-max and constant constraints and (iv) finally we solve the resulting formula using an existing PA solver Omega.

$$
\begin{array}{lll}
\mathit{SAT}(\pi) & \pi_F = \textsc{Inf-Trans}(\pi) & \textit{(1) Quantifier Elimination} \\
\Longrightarrow \mathit{SAT}(\pi_F) & \pi_N = \textsc{Int-Trans}(\pi_F) & \textit{(2) Normalization} \\
\Longrightarrow \mathit{SAT}(\pi_N) & \pi_G = \textsc{Simp}(\pi_N) & \textit{(3) Simplification} \\
\Longrightarrow \mathit{SAT}(\pi_G) & & \textit{(4) Omega}
\end{array}
$$

Figure 3.12: PAInf: SAT Checking

At a high level the intuition behind the SAT checking algorithm is as follows: after quantifier elimination, the π_F formula has quantifiers only on the finite domain variables. The normalization, detects tautologies and contradictions in constraints using infinity and rewrites the formula to π_G. The normalization eliminates all the infinite constants from the formula. The resulting formula (π_G) is in PA and its satisfiability can be checked using Omega. Next we describe the steps in the SAT checking algorithm in detail.

$$
\begin{array}{ccc}
[\mathbf{EVAL{-}FIN}] & [\mathbf{EVAL{-}INF}] & [\mathbf{EVAL{-}BOT}] \\
v \rightsquigarrow Z & \infty{+}\infty \rightsquigarrow \infty & \infty{+}(-\infty) \rightsquigarrow \bot \\
c \rightsquigarrow Z & -\infty{+}(-\infty) \rightsquigarrow -\infty & -\infty{+}\infty \rightsquigarrow \bot \\
-Z \rightsquigarrow Z & -\infty{+}Z \rightsquigarrow -\infty & \bot{+}Z \rightsquigarrow \bot \\
Z + Z \rightsquigarrow Z & Z{+}(-\infty) \rightsquigarrow -\infty & Z{+}\bot \rightsquigarrow \bot \\
Z - Z \rightsquigarrow Z & \infty{+}Z \rightsquigarrow \infty & \bot{+}\bot \rightsquigarrow \bot \\
c \times Z \rightsquigarrow Z & Z + \infty \rightsquigarrow \infty & -\bot \rightsquigarrow \bot
\end{array}
$$

Figure 3.13: PAInf: Evaluation Check

3.4.1 Normalization and Simplification

We define a set of rewriting rules based on the semantics of formulae in PAInf. We work only with closed-form formulae, thus after applying the quantifier elimination given in figure 3.11, all the remaining variables are in the finite domain ($\mathbb{Z}$). It is possible to compare the variables with infinities by evaluating their values (as they are all finite) using the semantics given in the section 3.3. This is performed by the `Evaluation Check` function in figure 3.13 which reduces each expression to a finite value Z. Thus for the normalization rules in figure 3.14 we only need to consider the integer values (Z) and the infinity constants. Note that, the `Evaluation Check` is only applied as part of the normalization process, with the purpose of checking the finiteness and eliminating infinity. In particular, the actual formula is only transformed to a form without infinity constants; it is not evaluated to a value.

The normalization process uses the rewriting rules given in figure 3.14 (rules for $\neq, \geq, <$ and $\min_=$ are similar and omitted for brevity). These rules detect the tautologies and contradictions in the usage of ∞ and $-\infty$, thus all the constraints involving ∞ and $-\infty$ are eliminated. After the application of these rules the given formula is reduced to a form which can be solved by existing PA solvers like Omega.

We also proved the following theorems and lemmas about quantifier elimination INF-TRANS and normalization INT-TRANS. These theorems and lemmas hold for both two-valued/three-valued logics and all choices of $(0 \times \infty)$. Hence, the Coq certified proof of these theorems and lemmas is also parameterized. All our theorems are stated in both directions, thus we prove not only soundness but also completeness of the procedure. Note that for quantifier elimination the universe of environment τ and the domain of

[NORM–INF–EQ]

$$\begin{array}{rcl}
\bot = _ & \rightsquigarrow & \texttt{error} \\
_ = \bot & \rightsquigarrow & \texttt{error} \\
Z = \infty & \rightsquigarrow & \texttt{false} \\
\infty = \infty & \rightsquigarrow & \texttt{true} \\
-\infty = \infty & \rightsquigarrow & \texttt{false} \\
-\infty = Z & \rightsquigarrow & \texttt{false} \\
-\infty = -\infty & \rightsquigarrow & \texttt{true} \\
\infty = Z & \rightsquigarrow & \texttt{false} \\
\infty = -\infty & \rightsquigarrow & \texttt{false} \\
Z = -\infty & \rightsquigarrow & \texttt{false}
\end{array}$$

[NORM–INF–LEQ]

$$\begin{array}{rcl}
\bot \leq _ & \rightsquigarrow & \texttt{error} \\
_ \leq \bot & \rightsquigarrow & \texttt{error} \\
Z \leq \infty & \rightsquigarrow & \texttt{true} \\
\infty \leq \infty & \rightsquigarrow & \texttt{true} \\
-\infty \leq \infty & \rightsquigarrow & \texttt{true} \\
-\infty \leq Z & \rightsquigarrow & \texttt{true} \\
-\infty \leq -\infty & \rightsquigarrow & \texttt{true} \\
\infty \leq Z & \rightsquigarrow & \texttt{false} \\
\infty \leq -\infty & \rightsquigarrow & \texttt{false} \\
Z \leq -\infty & \rightsquigarrow & \texttt{false}
\end{array}$$

[NORM–INF–LT]

$$\begin{array}{rcl}
\bot < _ & \rightsquigarrow & \texttt{error} \\
_ < \bot & \rightsquigarrow & \texttt{error} \\
Z < \infty & \rightsquigarrow & \texttt{true} \\
\infty < \infty & \rightsquigarrow & \texttt{false} \\
-\infty < \infty & \rightsquigarrow & \texttt{true} \\
-\infty < Z & \rightsquigarrow & \texttt{true} \\
-\infty < -\infty & \rightsquigarrow & \texttt{false} \\
\infty < Z & \rightsquigarrow & \texttt{false} \\
\infty < -\infty & \rightsquigarrow & \texttt{false} \\
Z < -\infty & \rightsquigarrow & \texttt{false}
\end{array}$$

[NORM–EQ–MAX]

$$\begin{array}{rcl@{\qquad}rcl}
\max_{=}(\infty,\infty,\infty) & \rightsquigarrow & \texttt{true} & \max_{=}(-\infty,Z,Z) & \rightsquigarrow & \texttt{false} \\
\max_{=}(_,_,\bot) & \rightsquigarrow & \texttt{error} & \max_{=}(-\infty,-\infty,-\infty) & \rightsquigarrow & \texttt{true} \\
\max_{=}(\infty,Z,-\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(\infty,Z,Z) & \rightsquigarrow & \texttt{false} \\
\max_{=}(-\infty,Z,-\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(\infty,-\infty,\infty) & \rightsquigarrow & \texttt{true} \\
\max_{=}(\infty,\infty,Z) & \rightsquigarrow & \texttt{true} & \max_{=}(-\infty,\infty,-\infty) & \rightsquigarrow & \texttt{false} \\
\max_{=}(-\infty,\infty,Z) & \rightsquigarrow & \texttt{false} & \max_{=}(Z,\infty,Z) & \rightsquigarrow & \texttt{false} \\
\max_{=}(\infty,-\infty,-\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(\infty,-\infty,Z) & \rightsquigarrow & \texttt{false} \\
\max_{=}(_,\bot,_) & \rightsquigarrow & \texttt{error} & \max_{=}(-\infty,-\infty,Z) & \rightsquigarrow & \texttt{false} \\
\max_{=}(Z,\infty,-\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(\infty,Z,\infty) & \rightsquigarrow & \texttt{true} \\
\max_{=}(\infty,\infty,-\infty) & \rightsquigarrow & \texttt{true} & \max_{=}(-\infty,Z,\infty) & \rightsquigarrow & \texttt{false} \\
\max_{=}(\bot,_,_) & \rightsquigarrow & \texttt{error} & \max_{=}(Z,-\infty,-\infty) & \rightsquigarrow & \texttt{false} \\
\max_{=}(Z,-\infty,\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(Z,Z,\infty) & \rightsquigarrow & \texttt{false} \\
\max_{=}(-\infty,-\infty,\infty) & \rightsquigarrow & \texttt{false} & \max_{=}(-\infty,\infty,\infty) & \rightsquigarrow & \texttt{false} \\
\max_{=}(Z,\infty,\infty) & \rightsquigarrow & \texttt{false} & &
\end{array}$$

[NORM–INF–ERR]

$$\begin{array}{rcll}
\texttt{error} & \rightsquigarrow & \texttt{false} & \text{(two-valued logic)} \\
\texttt{error} & \rightsquigarrow & \texttt{undefined} & \text{(three-valued logic)}
\end{array}$$

Figure 3.14: PAInf: Normalization (INT-TRANS)

constants η are both instantiated to $\mathbb{Z}_\infty$.

Lemma 3.4.1. Quantifier Elimination *$\phi_{\mathbb{Z}_\infty} \models^{\text{sat}}_{\mathbb{Z}_\infty} \pi$ if and only if $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}_\infty}$ INF-TRANS(π), $\phi_{\mathbb{Z}_\infty} \models^{\text{dst}}_{\mathbb{Z}_\infty} \pi$ if and only if $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}_\infty}$ INF-TRANS(π),*

For infinity elimination τ is $\mathbb{Z}_\infty$ and η is $\mathbb{Z}$. This is due to the fact that after

quantifier elimination the domain of all the variables is finite.

Lemma 3.4.2. Infinity Elimination $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}} \textsc{Int-Trans}(\pi)$, $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}} \textsc{Int-Trans}(\pi)$.

[**ELIM**]

$$\max_{=}(a_1, a_2, a_3) \leadsto (a_1 = a_2 \wedge a_3 \leq a_2) \vee (a_1 = a_3 \wedge a_2 \leq a_3)$$
$$\min_{=}(a_1, a_2, a_3) \leadsto (a_1 = a_2 \wedge a_2 \leq a_3) \vee (a_1 = a_3 \wedge a_3 \leq a_2)$$

[**SIMP**]

$$\beta \leadsto \textsc{Elim}(\beta)$$

$$\begin{array}{rcl@{\qquad}rcl}
\texttt{undefined} \wedge \pi & \leadsto & \texttt{undefined} & \pi \wedge \texttt{undefined} & \leadsto & \texttt{undefined} \\
\texttt{true} \wedge \pi & \leadsto & \pi & \pi \wedge \texttt{true} & \leadsto & \pi \\
\texttt{false} \wedge \pi & \leadsto & \texttt{false} & \pi \wedge \texttt{false} & \leadsto & \texttt{false} \\
\texttt{undefined} \vee \pi & \leadsto & \texttt{undefined} & \pi \vee \texttt{undefined} & \leadsto & \texttt{undefined} \\
\texttt{true} \vee \pi & \leadsto & \texttt{true} & \pi \vee \texttt{true} & \leadsto & \texttt{true} \\
\texttt{false} \vee \pi & \leadsto & \pi & \pi \vee \texttt{false} & \leadsto & \pi \\
\texttt{undefined} \rightarrow \pi & \leadsto & \texttt{undefined} & \pi \rightarrow \texttt{undefined} & \leadsto & \texttt{undefined} \\
\texttt{false} \rightarrow \pi & \leadsto & \texttt{true} & \pi \rightarrow \texttt{true} & \leadsto & \texttt{true} \\
\texttt{true} \rightarrow \pi & \leadsto & \pi & \pi \rightarrow \texttt{false} & \leadsto & \neg\pi \\
\neg\texttt{true} & \leadsto & \texttt{false} & \neg\texttt{false} & \leadsto & \texttt{true}
\end{array}$$

$$\neg\texttt{undefined} \leadsto \texttt{undefined}$$
$$\forall(w : \tau) \cdot \texttt{undefined} \leadsto \texttt{undefined}$$
$$\forall(w : \tau) \cdot \texttt{true} \leadsto \texttt{true}$$
$$\forall(w : \tau) \cdot \texttt{false} \leadsto \texttt{false}$$
$$\exists(w : \tau) \cdot \texttt{true} \leadsto \texttt{true}$$
$$\exists(w : \tau) \cdot \texttt{undefined} \leadsto \texttt{undefined}$$
$$\exists(w : \tau) \cdot \texttt{false} \leadsto \texttt{false}$$

Figure 3.15: Definition of Simplification

So for the total transformation $\textsc{Trans}(\pi) = \textsc{Int-Trans}(\textsc{Inf-Trans}(\pi))$ used in satisfiability checking, we have the following theorem:

Theorem 3.4.3. Satisfiability Checking $\phi_{\mathbb{Z}_\infty} \models^{\text{sat}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{sat}}_{\mathbb{Z}} \textsc{Trans}(\pi)$, $\phi_{\mathbb{Z}_\infty} \models^{\text{dst}}_{\mathbb{Z}_\infty} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\text{dst}}_{\mathbb{Z}} \textsc{Trans}(\pi)$,

Gallina, the internal functional language of Coq is strongly normalizing. Thus, all functions written in Coq must terminate.

Theorem 3.4.4. Termination *Satisfiability checking in PAInf (figure 3.12) terminates.*

The quantifier elimination with infinity expands the logical formula π and the normalization introduces many logical constants. We introduce a simplification function SIMP which recursively eliminates logical constants according to the rules in figure 3.15 in order to reduce the length of a formula. As Omega doesn't support $\max_=$ or $\min_=$ we also include the elimination of $\max_=$ and $\min_=$ in SIMP. Note that for three-valued logic, logical constants contains a third value: `undefined` which is not supported by Omega. Our SIMP function actually propagates `undefined` to the whole formula such that we know if a formula is undetermined before calling Omega due of the following theorem:

Theorem 3.4.5. Decide Undetermined $\phi_{\mathbb{Z}} \models^{\mathrm{udt}}_{\mathbb{Z}} \pi$ *if and only if* SIMP*(*π*)*=`undefined`

Thus, we do not need to extend Omega to support `undefined`. SIMP also preserves the validity of formulae:

Theorem 3.4.6. Simplification $\phi_{\mathbb{Z}} \models^{\mathrm{sat}}_{\mathbb{Z}} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\mathrm{sat}}_{\mathbb{Z}} \mathrm{SIMP}(\pi)$, $\phi_{\mathbb{Z}} \models^{\mathrm{dst}}_{\mathbb{Z}} \pi$ *if and only if* $\phi_{\mathbb{Z}} \models^{\mathrm{dst}}_{\mathbb{Z}} \mathrm{SIMP}(\pi)$.

3.5 Implementation

The Omega++ decision procedure (along with the proofs of all the associated lemmas and theorems) is implemented in Coq. Since Omega++ supports six different customized semantics our implementation of the transformation and

proofs is modular and composable. The customized parts—two- or three-valued logic, value of $(0 \times \infty)$ are abstracted as different module types composed by parameters and axioms. Each concrete choice instantiates the module type via definitions and lemmas. For example, part of our module type for "value" looks like this:

```
Module Type SEM_VAL.
  Parameter Val : Set.
  Parameter truth_and : Val -> Val -> Val.
  Parameter truth_or : Val -> Val -> Val.
  Parameter truth_not : Val -> Val.
  Axiom truth_and_comm : forall v1 v2,
             truth_and v1 v2 = truth_and v2 v1.
  ...
End SEM_VAL.
```

where `truth_and`, `truth_or` and `truth_not` are truth tables for conjunction, disjunction and negation. Given this module type our module implementation for a value in three-valued logic looks like this:

```
Module Three_Val_NoneError <: SEM_VAL.
  Inductive Val_Impl := VTrue | VFalse | VError.
  Definition Val := Val_Impl.
  Definition truth_and (v1 v2 : Val) := ...
  Definition truth_or (v1 v2 : Val) := ...
  Definition truth_not v := ...
  Lemma truth_and_comm : forall v1 v2,
               truth_and v1 v2 = truth_and v2 v1.
  Proof. intros; destruct v1, v2; simpl; trivial.
```

```
Qed.

...
```

We put our definition of semantics in another module which is parameterized by these module types. This method enables us to define transformations and prove theorems in a highly modular and compositional way, regardless of the concrete values of those parameter modules.

The following table presents some statistics for our Coq implementation of Omega++. The first column shows the file name, while the second and third columns are the number of lines in the file taken by the program and its soundness proof, respectively. Our total development is a modest 3,988 lines and the ratio of proof to program is a reasonable 2.35.

Table 3.1: Coq Development Details

Coq File	*Program*	*Proof*	*Time (s)*	*Description*
`Theory.v`	585	737	20.68	*Syntax and Semantics;* SIMP
`Transformation.v`	350	1,203	31.07	INF-TRANS, INT-TRANS
`Simplification.v`	0	856	338.96	*Tactics/lemmas for* SIMP
`Extraction.v`	257	0	1.27	*Module to extract OCaml code*
	1,192	2,796	391.98	*Total Coq*

The fourth column gives the time taken by Coq to verify the file (*i.e.* proof/type checking), using a 2.6 GHz Intel Core i7 with 16 GB of DDR3 RAM. Note that type checking times have very little to do with file length. For example, `Transformation.v` has 1,553 lines (combined program and proof), but takes less than 32 seconds to verify. On the other hand, verifying the 44 lines of the SIMP procedure, whose code is contained in `Theory.v`, takes more than five minutes!

We gain a number of benefits in exchange for implementing Omega++ in Coq. We get proof of termination for free since Gallina (the extractable pure

functional language of Coq) is strongly normalizing. More importantly, we get full machine-checked formal correctness proofs for our source code with respect to a well defined semantics for Presburger arithmetic with infinity. Coq's extraction facility then transforms the Gallina program into OCaml (or Haskell or Scheme), which we then compile and run as normal. A very simple handwritten interface in OCaml (`omegapp.ml`, 162 lines) hides the natural ugliness of auto-generated code from other OCaml modules and enables a useful optimization within the generated Coq code as detailed below.

Although extraction seems straightforward, there are a surprising number of pitfalls. We will next highlight the key optimizations we used to get good performance and discuss how the program affected the proof—and vice versa.

In the implementation we directly handle all of the logical operators and min-max constraints of the constraint language (figure 3.5), even though the "obvious" strategy would be to desugar aggressively. Unfortunately, sugar-free formulae are actually quite a bit larger than their svelter sugared cousins, resulting in a significant performance penalty. Working with fully-sugared formulae has a significant impact on the proofs because we must handle more cases than would otherwise be necessary.

Similarly, we allow the input formulae to specify, for each quantifier, whether the domain of quantification is over $\mathbb{Z}$ or over $\mathbb{Z}_\infty$. Quantifier elimination is expensive, and our "user"—the HIP/SLEEK verification toolset—often knows when a variable must be finite: in particular, program variables must be finite, whereas specification variables need not be. Communicating this fact to Omega++ resulted in significant performance gains, but again increased the proof effort due to the necessity of handling more cases.

To enable min/max, reduce the length of the output, eliminate redundant clauses, and propagate the `undefined` value, we implemented some basic simplifications (figure 3.15). The SIMP procedure was easy to implement but very painful to verify due to the vast number of cases we need to consider. In the end we wrote some custom proof tactics in Ltac (Coq's proof tactic language) which crunched through the tedium while we ate lunch.

The previous examples all trade one-time verification effort for a better-performing algorithm. On the other hand, sometimes the proof improves the program. Before we started on our Coq implementation, we did a OCaml prototype for the quantifier-free fragment of the problem. That prototype's version of normalization did additional case analysis. Due to our careful treatment of quantifier elimination we were able to prove that much of this case analysis was unnecessary in our Coq tool. Moreover, the Coq development identified a soundness bug in the OCaml prototype, which allowed the invalid transformation $x \geq y \rightsquigarrow x+1>y$, which is false when $x = y = \infty$.

We also use one engineering trick to boost the performance of the extracted code. The code uses strings to represent both variables and (arbitrary-sized) integers, but Coq's encoding of strings is less efficient than OCaml's. We therefore usually treat strings as an abstract type within Coq and manipulate them via an interface to OCaml's string functions, passed in using a functor. This interface takes only a few lines of `omegapp.ml` and results in a noticeable performance gain.

Overall, Omega++ is far better than our previous OCaml prototype. Consider:

Of course, our OCaml prototype is a bit of a straw man, but we have been quite convinced that the substantial effort that it took to write Omega++ in Coq was well-rewarded. Moreover, as we will soon see, Omega++ has comparable

Table 3.2: Comparison between Omega++ and Proto

Tool	*Sound*	*Complete*	*Termination*	*Semantics*	*Verified*
OCaml Prototype	*No*	*No*	*Unclear*	*Unclear*	*No*
Omega++	*Yes*	*Yes*	*Guaranteed*	*Precise*	*in Coq*

performance to our OCaml prototype, despite solving a trickier problem in a much more thorough way.

3.6 Experiments

To benchmark Omega++ we integrated it into the HIP/SLEEK verification toolset [23] and developed a suite of tests (mostly searching and sorting programs) whose specifications use ∞ in interesting ways. The source code for each of these programs can be investigated in detail and tested with Omega++ [108] on our web site. In all the experiments we selected three-valued logic in Omega++ and used $0 \cdot \infty \stackrel{\text{def}}{=} 0$ as these are the appropriate choices for program verification.

We use a 3.20GHz Intel Core i7-960 processor with 16GB memory running Ubuntu Linux 10.04 for our benchmarks, the first set of which are detailed in the table below. The first column lists the test name and the second gives its lines of code.

The third and fifth columns show that $\mathbb{Z}_\infty$ enables more readable and concise specifications. Specifically, the third column gives the number of disjunctions required to express the test's specifications using $\mathbb{Z}$, whereas the fifth column expresses the same properties using $\mathbb{Z}_\infty$. For each test in the first group (top six), $\mathbb{Z}_\infty$ requires fewer disjunctions. We do need to be a bit careful: although the specifications are informally for the same property (*e.g.*,

Table 3.3: Verification benchmarks with Infinity

Benchmark	*LOC*	*Disjuncts* ($\mathbb{Z}$)	*Time* (Ω)	*Disjuncts* ($\mathbb{Z}_\infty$)	*Time* (Ω++)
Insertion Sort	30	4	0.14	2	0.15
Selection Sort	69	14	0.36	7	0.35
Binary Search Tree	105	12	0.43	6	0.35
Bubble Sort	110	12	0.29	9	0.50
Merge Sort	91	6	0.32	4	1.81
Priority Queue	207	16	0.84	10	2.73
Total Correctness	21	-	-	2	0.21
Sorting Min and Max	79	-	-	7	1.82

"sortedness"), typically the specifications in $\mathbb{Z}_\infty$ are formally stronger since the embedded quantification occurs over larger sets. Note that we do not claim that Omega++ eliminates the disjunctions from reasoning since the quantifiers over infinities hide the disjunctions inside them. However, using infinities provides a useful abstraction to express the same property as the given specification is more concise.

The difference in formal strength is the fundamental reason why the times given in columns four and six differ. Column four gives the time (including all of HIP/SLEEK) using Omega, whereas column six gives the time using Omega++. For the first four examples Omega++ is comparable to Omega, but in the final two of the first group of tests we believe the difference in the domain of quantification results in a significantly harder theorem in $\mathbb{Z}_\infty$, and thus, a noticeably longer runtime.

The second group of tests (bottom two) shows that $\mathbb{Z}_\infty$ is more expressive: the specifications for each of these tests is not expressible using only $\mathbb{Z}$. The runtimes we get using (HIP/SLEEK and) Omega++ are encouragingly modest.

Comparison with similar tools. Lasaruk and Sturm [64] also propose extending Presburger arithmetic with infinity. Their work differs from ours in several respects. First, they only add a single infinity value, thus dodging any

thorny—but in our view, important—semantic issues involving $\infty - \infty$. More importantly, Lasaruk and Sturm describe an algorithm but do not provide an implementation. For benchmarking purposes, we implemented their algorithm and tested it using the constraints generated from our test suite. We also compared our previous OCaml prototype. Our results are as follows:

Table 3.4: Comparing PAI and Omega++

Benchmark	*Calls*	*Time (PAI)*	*Time (Proto)*	*Time (Ω++)*
Insertion Sort	100	4.58	0.78	0.39
Selection Sort	245	>600.00	0.62	0.78
Binary Search Tree	116	150.00	0.48	0.50
Bubble Sort	336	>600.00	1.25	1.34
Merge Sort	155	>600.00	1.05	1.92
Priority Queue	778	>600.00	FAIL	1.20
Total Correctness	120	>600.00	0.31	0.16
Sorting with Min and Max	376	>600.00	0.29	0.19
Entailment Examples	124	1.89	FAIL	1.42
Lemma Examples	35	1.88	1.27	1.65
Total (except PQ and EE)	1,824	>3,862.14	7.21	8.11

The second column gives the number of times the associated decision procedure was called for each test. The third column gives the times for Lasaruk and Sturm's "PAI" algorithm; many of the tests timed out after 10 minutes. The fourth column gives the times for our OCaml prototype "Proto"; notice that for two of the tests Proto failed (completeness holes). The fifth column gives the times for Omega++.

It is obvious that PAI, at least when implemented directly as given by Lasaruk and Sturm [64], is uncompetitive. Thus, Omega++ is always faster than PAI. When comparing Proto to Omega++, recall that Proto is only trying to solve the simpler problem of quantifier-free formulae. Despite this, for many of our tests the tools perform similarly. For a few tests, some of Proto's heuristics result in appreciably better times; we plan to study these tests in

more detail in the future to try to improve Omega++. Overall, Omega++'s performance is competitive.

Inference. As described in section 3.2.5, quantifier elimination in Presburger arithmetic with infinity can help with invariant generation of octagonal constraints. The table below benchmarks using Omega++ for this analysis technique. The first column gives the test name. The second and third columns give the user-provided spatial pre- and postconditions in separation logic. The fourth column gives the inferred pure (non-heap) specification, while the last column gives the time used by Omega++. The final test is noteworthy because the inferred invariant uses min/max constraints.

Table 3.5: Inference with Omega++

Method	*Pre*	*Post*	*Inferred*	*Time (Omega++)*
Create	true	$\texttt{ll}\langle\texttt{res},\texttt{m}\rangle$	$\texttt{m}{=}\texttt{n}$	0.13
Delete	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle$	$\texttt{ll}\langle\texttt{res},\texttt{m}\rangle$	$\texttt{n}{-}1{\leq}\texttt{m}$	0.17
Insert	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle{\wedge}\texttt{x}{\neq}\texttt{null}$	$\texttt{ll}\langle\texttt{x},\texttt{m}\rangle$	$\texttt{n}{=}\texttt{m}{-}1$	0.13
Copy	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle * \texttt{ll}\langle\texttt{res},\texttt{m}\rangle$	$\texttt{ll}\langle\texttt{x},\texttt{m}\rangle$	$\texttt{m}{=}\texttt{n}$	0.16
Remove	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle{\wedge}\texttt{x}{\neq}\texttt{null}$	$\texttt{ll}\langle\texttt{x},\texttt{m}\rangle$	$\texttt{n}{-}1{\leq}\texttt{m}{\wedge}\texttt{m}{\leq}\texttt{n}$	0.19
Return	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle$	$\texttt{ll}\langle\texttt{x},\texttt{m}\rangle$	$\texttt{m}{=}\texttt{n}{\wedge}0{\leq}\texttt{m}$	0.07
Traverse	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle$	$\texttt{ll}\langle\texttt{x},\texttt{m}\rangle$	$\texttt{m}{=}\texttt{n}$	0.12
Get	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle{\wedge}\texttt{x}{\neq}\texttt{null}$	$\texttt{ll}\langle\texttt{res},\texttt{m}\rangle$	$\texttt{m}{=}\texttt{n}{-}2{\wedge}2{\leq}\texttt{n}$	0.11
Head	$\texttt{ll}\langle\texttt{x},\texttt{n}\rangle{*}\texttt{ll}\langle\texttt{y},\texttt{m}\rangle$	$\texttt{ll}\langle\texttt{res},\texttt{n}{+}\texttt{m}{-}1\rangle$	$1{=}\texttt{min}(\texttt{n},\texttt{m})$	0.21

3.7 Comparative Remarks and Summary

3.7.1 Ghost Variables

Reynolds demonstrated that *ghost variables* [95] were useful for verifying sequential programs. Their importance is highlighted when proving program, object or loop invariants [77], refining between two transition systems [76] or when considering program's security aspects [74].

In a concurrent setting, whether adopting a Hoare [95] or a VDM-style [51] program logic, specification formulae are generally extended with ghost variables in order to explicitly record the information of interest between communicating processes. Our work enriches specifications by extending the domain of ghost values with the mathematical concepts of positive and negative infinity.

3.7.2 Decision Procedures

Presburger arithmetic [93] is one of the canonical examples of an important decidable problem. Kuncak *et al.* [62, 63] presented a decision procedure for a quantifier-free fragment of Boolean Algebra with Presburger arithmetic (QFBAPA) which can be used to prove a mixed set-based constraint with symbolic cardinality and linear arithmetic. QFBAPA was later extended to the more challenging case of multisets [91] and proved to be NP-complete [92]. The VCDryad [94] framework combines separation logic with decision procedures for sets and multi sets to verify programs with natural proofs.

Another line of work is focused on not only on proving decidability of certain logic fragments [16, 7] but also on both the applicability of exhibited decision procedures [77, 39] as well as their efficiency [13]. In particular, arithmetic decision procedures [85, 80, 86] are a fundamental part of interactive systems such as: Isabelle/HOL [84], Coq [11] and ACL2 [54]. Considerable attention is also paid to the problem of exploiting methods which combine decision procedures from different domains [28, 75], resulting in a decision of a union of theories the majority being based on the foundations built by Nelson-Oppen[82] or Shostak [109].

Lasaruk and Sturm [64] were the first to tackle the problem of extending PA

with infinity, proving completeness and decidability. As discussed in section 3.6, our work differs from theirs by providing distinct positive and negative infinities as well as by providing an implementation. In our work, we build an implementation on top of Omega calculator [55], and certify it in Coq [1].

The general problem of adding infinities to the set of reals was addressed by Weispfenning [115]. This was later extended to mixed real and integer quantifier elimination in [116]. Another interesting extension of decision procedures for real arithmetic is the addition of infinitesimals. The proof assistant Isabelle/HOL [84] supports the use of such infinitesimals. Loos and Weispfenning [73] first proposed a virtual substitution approach for quantifier elimination of infinitesimals. In Omega++, we use a similar virtual substitution to eliminate infinities as part of the decision procedure. For the case of linear formulas, the use of substitution for elimination of quantifiers was first proposed in [114].

3.7.3 Summary

We present Omega++, a decision procedure for Presburger arithmetic with infinity $\mathbb{Z}_\infty$. Infinity is a useful abstraction, increasing a program logic's ability to model infinite data structures, reason about termination, and compose more elegantly. Moreover, specifications with infinity are often more concise. Finally, quantifier elimination for infinities enables an extension to an existing analysis technique.

Omega++ itself is a sound and complete decision procedure for $\mathbb{Z}_\infty$, and has been Coq-certified to respect a precise formal semantics for $\mathbb{Z}_\infty$. We integrate Omega++ into an existing verifier and evaluated it on a benchmark of small programs, demonstrating that it can perform well in practice. Omega++

demonstrates that we can develop useful, efficient, and certified programs for program verification and analysis.

Chapter 4

Verified Subtyping with Traits and Mixins

Traits allow decomposing programs into smaller parts and mixins are a form of composition that resemble multiple inheritance. Unfortunately, in the presence of traits, programming languages like Scala give up on subtyping relation between objects. In this chapter, we present a method to check subtyping between objects based on entailment in separation logic. We implement our method as a domain specific language in Scala and apply it on the Scala standard library. We have verified that 67% of mixins used in the Scala standard library do indeed conform to subtyping between the traits that are used to build them.

4.1 Introduction

Traits [35] have been recognized as a mechanism to support fine grained reuse in programming. Several programming languages (Scala, Fortress, Ruby, etc.) support the use of traits in some form or other. Traits and mixins provide

support for code reuse and composition that goes beyond classes and inheritance in object oriented programs. In addition, object oriented (OO) programs themselves are notoriously hard to verify in a modular fashion. Recently [22, 90, 32] separation logic based approach has yielded success in verification of object oriented programs. This include support for verifying inheritance and behavior subtyping, in conformance with OO paradigm. In this chapter, we extend the work done on verification of OO programs in separation logic to verify subtyping with traits and mixins. Separation logic is a good choice for verifying traits because it supports abstraction and information hiding [90] in the presence of inheritance and subtyping. This enables us to avoid re-verification when dealing with OO programs that make sure of subtyping.

Below we consider an example that illustrates the problem of subtyping with traits and mixins. The $ICell$ trait captures an object with an integer value that can be accessed with get and set methods. The $BICell$ trait provides a basic implementation for $ICell$, while the $Double$ and Inc traits extend the $ICell$ trait by doubling and incrementing the integer value respectively.

$$
\begin{array}{l}
\text{trait } ICell \{ \\
\quad \text{def } get() : Int \\
\quad \text{def } set(x : Int) \\
\}
\end{array}
\qquad
\begin{array}{l}
\text{trait } BICell \text{ extends } ICell \{ \\
\quad \text{private var } x : Int \\
\quad \text{def } get() \\
\qquad \{ x \} \\
\quad \text{def } set(x : Int) \\
\qquad \{ this.x = x \} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\text{trait } Double \text{ extends } ICell \{ \\
\quad \text{abstract override def } set(x : Int) \\
\qquad \{ \text{super}.set(2 * x) \} \\
\quad \} \\
\text{trait } Inc \text{ extends } ICell \{ \\
\quad \text{abstract override def } set(x : Int) \\
\qquad \{\text{super}.set(x + 1)\} \\
\quad \}
\end{array}
$$

These traits are used in the following class mixins. The integer value field of the objects of $OddICell$ mixin is always odd, while the value is even for objects of $EvenICell$ mixin. (This is due to the fact that traits are mixed in a linearized order, thus for $OddICell$ we have the integer value x which is doubled by the set method in $Double$ trait to become $2x$ and then incremented by the set method in the Inc trait to be $2x + 1$ which is always odd.)

$$
\begin{array}{l}
\textsf{class } OddICell \textsf{ extends } BICell \textsf{ with } Inc \textsf{ with } Double \\
\textsf{class } EvenICell \textsf{ extends } BICell \textsf{ with } Double \textsf{ with } Inc
\end{array}
$$

In the presence of traits, the type system of Scala is not strong enough to distinguish between accepted uses of the traits. This can be illustrated by the following example.

$$
\begin{array}{l}
\text{def } m\ (c :\ BICell \text{ with } Inc \text{ with } Double) :\ Int = \{c.get\} \\
\text{val } oic = \text{new } OddICell \\
\text{val } eic = \text{new } EvenICell \\
m(oic)\ \ // \ Valid \\
m(eic)\ \ // \ Valid
\end{array}
$$

The method m can be called with an object of both mixins $EvenICell$ and $OddICell$, even though the expected object type is a subtype of $OddICell$ and not $EvenICell$. Thus, the type system in Scala cannot distinguish between the two calls made to method m as it does not check for subtyping between the objects. This means that mixins and traits in Scala violate the Liskov substitution principle [71]. Substitutability of an object by its subtype is a key principle of object oriented programming and is stated in [72] as follows:

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Traits in the mixin class composition in Scala do not adhere to this principle. The key contribution of this chapter is to present a method for checking subtyping in the presence of traits and mixins in Scala. This enables us to verify mixins and validate that they use traits according to Liskov substitution principle. Our method is based on reduction of the subtyping to entailment checking in separation logic.

In section 4.2, we present an approach based on entailment in separation logic to verify subtyping. In section 4.3, we present a domain specific language which is embedded in Scala and can support verified subtyping with traits and mixins. We apply our technique to the mixin class hierarchies in the Scala standard library and verify subtyping in 67% of the traits as shown in section 4.4. Our complete development including the source code of the domain specific language and all examples are available on-line at the following URL.

`http://loris-7.ddns.comp.nus.edu.sg/~project/SLEEKDSL/`

4.2 Verified Subtyping

We consider a core language based on [22] for formalizing our approach. As shown in figure 4.1, to simplify the presentation we focus only on type information for traits and mixins while ignoring all other features in our core language. We also assume that all classes are part of mixin compositions and only traits are used to create mixins. Since, existing approaches [22] can handle class based single inheritance, we focus only on mixin compositions in this chapter. The rest of the constructs in the core language are related to predicates (Φ) in separation logic. Each trait (and mixin) C can be represented by a corresponding predicate $C(v^*)$.

$$
\begin{array}{lll}
mixin & ::= & class\ C\ [extends\ C_1]\ [with\ C_2]^* \\
pred & ::= & C(v^*) \equiv \Phi\ \ [inv\ \ \pi] \\
\Phi & ::= & \bigvee\ (\exists w^* \cdot \kappa \wedge \pi)^* \\
\kappa & ::= & \texttt{emp} \mid C(v^*) \mid \kappa_1 * \kappa_2 \\
\pi & ::= & \alpha \mid \pi_1 \wedge \pi_2 \quad \alpha ::= \beta \mid \neg\beta \\
\beta & ::= & v_1 = v_2 \mid v = \texttt{null} \mid a \leq 0 \mid a = 0 \\
a & ::= & k \mid k \times v \mid a_1 + a_2
\end{array}
$$

Figure 4.1: Core Language for Traits and Mixins

Predicates based on separation logic are sufficient to specify mixins because of class linearization in Scala [87]. After class linearization a mixin class composition (unlike multiple inheritance) has a single linear hierarchy. The translation of mixins and traits from Scala to the core language given in figure 4.1 is done by creating a corresponding predicate for each mixin in the Scala program. E.g. in the case of our running example, the mixins give rise to the following linearizations and predicates respectively:

$$OddICell \leftarrow Double \leftarrow Inc \leftarrow BICell$$

$$\textbf{\textit{OddICell}}(this) \equiv \textbf{\textit{BICell}}(this, v) * \textbf{\textit{Inc}}(v, v_1) * \textbf{\textit{Double}}(v_1, \texttt{null})$$

$$EvenICell \leftarrow Inc \leftarrow Double \leftarrow BICell$$

$$\textbf{\textit{EvenICell}}(this) \equiv \textbf{\textit{BICell}}(this, v) * \textbf{\textit{Double}}(v, v_1) * \textbf{\textit{Inc}}(v_1, \texttt{null})$$

A mixin class composition can be treated as a single inheritance hierarchy based on the linearization and thus, subtyping between the mixins can be decided by checking the entailment based on separation logic predicates. In case of our running example, the call to method m is valid with oic object but not the eic object as the following entailments show.

$$\textbf{\textit{OddICell}}(oic) \vdash \textbf{\textit{BICell}}(c, v) * \textbf{\textit{Inc}}(v, v_1) * \textbf{\textit{Double}}(v_1, \texttt{null}) \quad Valid$$

$$\textbf{\textit{EvenICell}}(eic) \vdash \textbf{\textit{BICell}}(c, v) * \textbf{\textit{Inc}}(v, v_1) * \textbf{\textit{Double}}(v_1, \texttt{null}) \quad Invalid$$

We now show how the problem of checking subtyping between objects belonging to two different mixins is reduced to an entailment between the corresponding predicates in separation logic. This entailment can be checked with the help of existing solvers for separation logic (like SLEEK [21]). The entailment rule for checking subtyping with traits and mixins is given in figure 4.2. An object of mixin D is a subtype of mixin C when the entailment between their corresponding predicates in separation logic is valid.

Entailment checking in separation logic can be used to decide subtyping with traits and mixins. But in order to integrate subtyping support inside Scala we face some engineering challenges. In particular, it is too restrictive and infeasible to do this kind of checking for all the mixins. This requires support for selective subtyping as all mixins will not satisfy the subtype relation. In

$$\frac{\begin{array}{c}[\mathbf{ENT-Subtype-Check}]\\ class\ C\ [extends\ C_1]\ [with\ C_2]^*\\ class\ D\ [extends\ D_1]\ [with\ D_2]^*\\ \boldsymbol{C}_1(this, v_1)[*\boldsymbol{C}_2(v_1, v_2)]^* \vdash \boldsymbol{D}_1(this, u_1)[*\boldsymbol{D}_2(u_1, u_2)]^*\end{array}}{C :> D}$$

Figure 4.2: Checking Subtyping with Entailment

order to provide the programmer the choice of checking subtyping usage in their methods we have implemented an embedded domain specific language (DSL) in Scala. This DSL uses the SLEEK entailment checker for checking the validity of entailments in separation logic. In the next section we describe the SLEEK DSL and how it is integrated in Scala.

4.3 Implementation with SLEEK DSL

Our proposal is based on embedding a domain specific language (SLEEK DSL) in Scala. As shown in figure 4.3, a Scala library (SLEEK lib) interfaces directly with the external application - the SLEEK entailment prover. In addition, we extend Scala with a DSL (SLEEK DSL) which makes use of the Scala library to provide the entailment checking feature inside Scala programs. Further, for using with the Scala interpreter we provide an interactive mode (SLEEK inter) which uses the SLEEK DSL and library to enable interactive entailment proving. Thus, the implementation of the verified subtyping in Scala with SLEEK has three main components:

- a Scala library that supports all SLEEK interactions
- a domain specific language (DSL) implemented in Scala that models the SLEEK input language. With this DSL we get for free embedded type

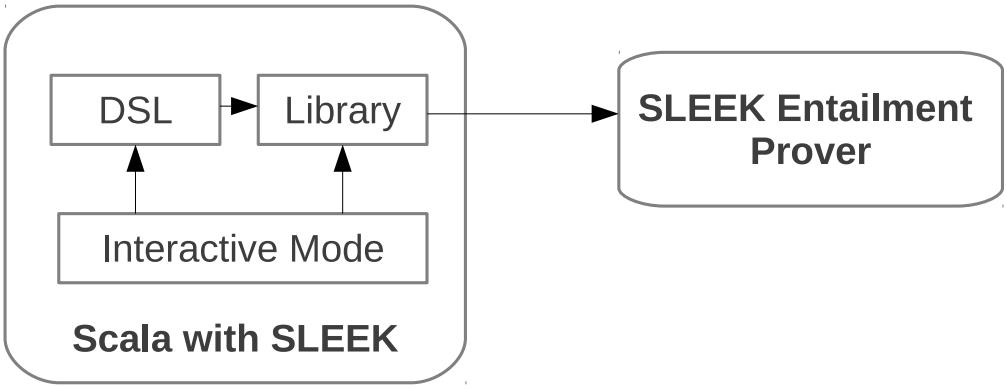


Figure 4.3: Overview of SLEEK DSL

checking in Scala.

- a helper library designed for the Scala interpreter. The library runs SLEEK in interactive mode in the background to provide seamless integration with Scala.

In short, the SLEEK library provides basic functionality for constructing Scala objects representing separation logic formulas. The entailment checking method is in fact the actual front-end for SLEEK. It takes two Scala objects representing separation logic formulas, translates them to the SLEEK input language and invokes SLEEK. The result and the possible residue is captured and parsed using the Scala parser combinator library to extract the Scala representation. To facilitate a better syntax for writing formulas and support for richer interplay with the Scala types we present a domain specific language, SLEEK DSL implemented on top of the Scala library. We will outline the SLEEK DSL by presenting how an entailment checking can be encoded in our DSL.

4.3.1 SLEEK DSL

As an example consider the following entailment check between two separation logic formulas defined using SLEEK DSL.

$$\text{val } r = \quad \texttt{x::node}\langle \_, \texttt{null}\rangle \vdash \texttt{x::ll}\langle \texttt{m}\rangle \ \&\& \ \text{m}===1$$

It encodes an entailment between two formulas, one describing a single heap node, an instance of a data structure called $node$. The second formula describes a state in which x is the root pointer of a data structure described by the ll predicate. This predicate abstracts a linked list of size m.

SLEEK DSL relies on the functions defined in the SLEEK Library to create new easy to use operators that provide a more user friendly syntax. A special operator, the double colon (::) is used to describe the points-to relation commonly used for heap nodes. It also provides the usual arithmetic (e.g. $+$,$-$) and boolean (e.g. $\&\&$, $||$, $===$, $!==$, $\vdash$) operators to help in constructing the separation logic formula. The notation used in the DSL is similar to the one used for SLEEK in [21]. The use of a DSL allows easy intermixing of SLEEK formulas with other Scala types. We use implicit conversions between types (e.g. from $scala.Int$ to $formula[IntSort]$) to make it even easier to use these formulas in Scala programs.

Furthermore, our library provides a definition for the $isValid$ method in the formula class. In order to check the validity of the above entailment it is sufficient to call $r.isValid$ which feeds the entailment to SLEEK and converts the result back into a $scala.Boolean$ for use as a conditional. Implicit methods provide an easy mechanism to convert from one type of object to the desired type. This enables the support for a SLEEK like syntax within Scala. Formulas allow for a variety of types for the parameters used (such as x and m). In the

Scala library, these types are grouped under the following type hierarchy.

$$\begin{aligned}
&\text{sealed trait } Top\\
&\text{trait } BoolSort \text{ extends } Top\\
&\text{trait } IntSort \text{ extends } Top\\
&\text{trait } BagSort \text{ extends } Top\\
&\text{trait } ShapeSort \text{ extends } Top\\
&\text{trait } Bottom \text{ extends } BoolSort\\
&\qquad \text{with } IntSort \text{ with } BagSort \text{ with } ShapeSort
\end{aligned}$$

This trait allows the embedding of the types used in the separation logic formula as Scala types. By defining the various operators using these types, soft type checking for SLEEK formulas is automatically ensured by the underlying Scala type system. The benefit of using a DSL is that it provides a simpler syntax and familiar look and feel for the user of the library. The formula represented by the DSL is also much more concise.

The SLEEK DSL allows programmers to verify entailments written in separation logic. In addition, programmers can use the DSL to encode subtyping check as an entailment check in separation logic as described in section 4.2.

4.3.2 SLEEK Interactive Mode

The Scala runtime provides a good interpreter for rapid prototyping which can be used from the command line. Similarly, SLEEK also has an interactive mode in which it accepts commands and gives the results back to command line. In order to make SLEEK's interactive mode available to the Scala interpreter, we provide a helper library that hides the extra intricacies incurred by using SLEEK

interactively. The benefit of using the interactive mode is that the user defined predicates and data types will not be defined again with each call to $isValid$ method. This makes the interactive mode of SLEEK DSL faster when compared to calling the same function from the basic SLEEK library.

Our implementation for verified subtyping integrates into Scala as an API (SLEEK library), as a language (SLEEK DSL) and as an interpreter (SLEEK Interactive mode). This provides programmers the ability to use our procedure in different ways as desired.

4.4 Experiments

We have used SLEEK DSL to verify subtyping of mixin compositions from the Scala standard library. To the best of our knowledge this is the first such study of subtyping in Scala. The following table presents the results. The first column is the name of the class hierarchy. The second column lists the total number of mixins in the hierarchy, while the third column gives the number of mixins for which we can verify that the subtyping relation holds. The last column gives the percentage of mixins with subtyping. As an example of mixin hierarchy

Table 4.1: Experiments with Traits and Mixins

Class Hierarchy	*Total Num of Mixins*	*Mixins with Subtyping*	*Percentage*
Exceptions	11	11	100
Maths	5	4	80
Parser Combinator	6	6	100
Collections	27	12	44
Total	49	33	67

whose subtyping relations are verified, consider the following which represents the maths library in Scala. The only mixin which breaks the subtyping relation is PartialOrdering. Rest of the mixins can be verified to respect the expected

subtyping. Thus we have verified that subtyping holds for 4 out of 5 mixins that are part of math class hierarchy.

Equiv is SUPERTYPE of PartialOrdering

PartialOrdering is NOT SUPERTYPE of Ordering

Ordering is SUPERTYPE of Numeric

Numeric is SUPERTYPE of Integral

Numeric is SUPERTYPE of Fractional

4.5 Comparative Remarks and Summary

The work that comes closest to our method for checking subtyping is the work of Bierman et.al [12], they provide a mechanism to use SMT solvers for deciding subtyping in a first order functional language. On the other hand, we use SLEEK an entailment checker for separation logic to decide subtyping between traits and mixins. SMT solvers have also been used [4] for verifying typing constraints. Similar to our implementation of SLEEK DSL, the $Scala^{Z3}$ proposal of Köksal et. al [58] integrates the Z3 SMT solver into Scala. Although the integration is similar, the two solvers have different focuses: Z3 is a general SMT solver, while SLEEK is a prover for separation logic.

Another line of work is on specification and verification of traits and mixins. Damiani et. al explore trait verification in [26]. They observe the need for multiple specifications and introduce the concept of proof outline. They support a trait based language with limited composition - symmetric sum of traits and trait alteration. Our work does not directly address the issue of trait verification but checking subtyping is essential part of OO verification using separation logic [22]. We believe that dynamic specifications of [22] along with verified subtyping can be used to verify traits and mixins. Behavior

subtyping is a stronger notion of subtyping between objects. The approach of lazy behavioral subtyping [33] can support incremental verification of classes in presence of multiple inheritance. However, this is overly restrictive for mixin compositions in Scala and our method provides a more flexible support for subtyping in Scala. Even though we present our work in the context of Scala, the same approach can be applied to any other language supporting mixins and traits. By doing a suitable translation to our core language, we can define and verify subtyping based on entailment checking in separation logic.

In this chapter, we present a method to enable verified subtyping in Scala. Our method is based on a reduction to entailment checking in separation logic. We implement a domain specific language (SLEEK DSL) in Scala to enable programmers to check subtyping in their programs. Using SLEEK DSL we carry out a study of the Scala standard library and verified that 67% of the mixins were composed of traits that are in a subtyping relation.

Chapter 5

Specifying Compatible Sharing in Data Structures

Automated verification of programs that utilize data structures with intrinsic sharing is a challenging problem. Verifying such programs is of practical importance because they occur in many device drivers, runtime systems, and operating system kernels. We develop an extension to separation logic that can reason about aliasing in heaps using a notion of *compatible sharing*. Compatible sharing can model a variety of fine grained sharing and aliasing scenarios with concise specifications. Given these specifications, our entailment procedure enables fully automated verification of a number of challenging programs manipulating data structures with intrinsic sharing. We benchmark our prototype with examples derived from practical algorithms found in systems code, such as those using threaded trees and overlaid data structures.

5.1 Introduction

Systems software frequently employs data structures with intrinsic sharing, such as threaded trees (a data structure which can be treated simultaneously as a list and a tree). Sharing enables more efficient use of memory and better performance (*e.g.*, in a binary search tree where the nodes are also part of a linked list that maintains the order in which the elements were inserted, we get $O(\log n)$ lookup for arbitrary elements using the tree pointers and $O(1)$ lookup for the last inserted element using the list pointer). Unfortunately, sharing prevents easy formal reasoning because it precludes simple reasoning about the different parts of the data structure in isolation.

Many common data structures, such as linked lists and binary trees, can be naturally represented with $*$ because their constituent subparts occupy disjoint parts of the heap. However, many more sophisticated data structures, such as threaded trees and graphs, cannot easily be so naturally represented due to the sharing intrinsic to such structures.

We extend the notion of separation to enable local reasoning for such intrinsically-shared data structures by introducing a notion of *compatibility*. In brief, two predicates are compatible when updates to one will not affect the other, despite potential spatial overlap.

Consider the following example. The Linux IO scheduler maintains a structure which overlays a doubly-linked list, which maintains insertion order, and a red-black tree, which provides efficient indexing to arbitrary nodes. If the data fields of the nodes are not updated, then the linked list and tree structures do not affect each other despite sharing the entire heap, and we can "frame out" one while working on the other. Thus, our notion of compatibility relies on restricting access to parts of the described structure in a way that is similar to

fractional permissions but without the attendant bookkeeping.

Prior work on program analysis for overlaid data structures [68] only verifies the shape of the data structure and cannot handle functional properties like order and height balance. In contrast, our proposal is based on user defined inductive predicates with shape, size and bag properties that allows us to express and verify functional correctness of data structures with compatible sharing. In addition, we have certified the correctness proof of *compatibility* checking in Coq.

Our main contribution is an automated procedure to check compatibility and verify programs using compatible data structures. In particular, we describe:

- a specification mechanism that can model sharing and aliasing scenarios,
- an entailment procedure to reason about sharing,
- how to automate the verification of compatible sharing in data structures, and
- our implementation and benchmark its performance. Our prototype, together with a web-based GUI for easy experimentation and machine checked proofs of compatible sharing in Coq, is available at: `http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/`

The rest of the chapter is as follows. In section 5.2, we give some motivating examples. In section 5.3, we formalize our specification language. In section 5.4, we discuss verification with compatible sharing. In section 5.5, we discuss our implementation and experiments. In section 5.6, we review some related work and conclude.

5.2 Motivating Examples

5.2.1 From Separation to Sharing

Separation logic provides a natural way to represent disjoint heaps using the separating conjunction $*$. However, if two assertions both require some shared portion of memory, then $*$ cannot easily combine them. Consider the following simple example:

$$\texttt{data pair} \{ \texttt{int fst}; \texttt{ int snd} \}$$

Here `pair` is a data structure consisting of two fields, `fst` and `snd`. The following assertion indicates that `x` points to such a structure with field values `f` and `s`:

$$\texttt{x} \mapsto \texttt{pair}\langle \texttt{f}, \texttt{s} \rangle$$

We denote two disjoint pairs `x` and `y` with the *separating* conjunction $*$, which ensures that `x` and `y` cannot be aliased:

$$\texttt{x} \mapsto \texttt{pair}\langle \texttt{f}_1, \texttt{s}_1 \rangle * \texttt{y} \mapsto \texttt{pair}\langle \texttt{f}_2, \texttt{s}_2 \rangle$$

In contrast, to capture aliased pairs we use *classical* conjunction $\wedge$ as follows:

$$\texttt{x} \mapsto \texttt{pair}\langle \texttt{f}_1, \texttt{s}_1 \rangle \wedge \texttt{y} \mapsto \texttt{pair}\langle \texttt{f}_2, \texttt{s}_2 \rangle$$

The $\wedge$ operator specifies "must aliasing", that is, $\wedge$ ensures that the pointers `x` and `y` are the equal and that the object field values are identical (*i.e.*, $\texttt{f}_1 = \texttt{f}_2$ and $\texttt{s}_1 = \texttt{s}_2$).

The basic separating and classical conjunctions are sufficient for "tree-like" data structures. However, to represent and reason about more sophisticated

structures we need more subtle specification techniques. Consider a program that is manipulating a threaded tree (a data structure which overlays a list and an ordered tree). The nodes of a common type of threaded tree have six fields: a `data` field; two "list" fields, `next` and `previous`; and three "tree" fields, `left`, `right`, and `parent`. To verify "list operations" such as enqueue and dequeue, we wish to frame out `left`, `right`, and `parent`; conversely, to verify "tree operations" such as insert and lookup, we wish to frame out `next` and `previous`. Tree operations also need access to the `data` field, to support $O(\log\ n)$ access. All of the above means we wish to support field-level framing.

To do so, we add annotations to fields; when the field of an object is absent (or inaccessible) we mark it with `@A`, whereas when it is present (or accessible for read/write) we mark it with `@M`. Consider the following:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1@\mathtt{M}, \mathtt{s}_1@\mathtt{A}\rangle * \mathtt{y}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2@\mathtt{A}, \mathtt{s}_2@\mathtt{M}\rangle$$

This formula asserts that the heap can be split into two disjoint parts, the first of which contains a first-half-pair pointed to by `x`, and the second of which contains a second-half-pair pointed to by `y`. Since by default fields are mutable `@M`, and when a field is absent `@A` we need not bind a variable to its value, the formula can also be written as:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A}\rangle * \mathtt{y}{\mapsto}\mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2\rangle$$

All this seems simple enough, but there is a subtle wrinkle: notice that `x` and `y` may be aliased (if the combined heap contains a single pair that has been split in half fieldwise), but need not be (if the combined heap contains two distinct

half pairs). This ambiguity is inconvenient. We introduce a new operator, the *overlaid* conjunction $\mathbb{A}$ to indicate that the locations are the same although the fields are disjoint. Thus, when we write

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A}\rangle \mathbin{\mathbb{A}} \mathtt{y}{\mapsto}\mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2\rangle$$

we unambiguously mean that `x` and `y` are aliased and have been split fieldwise. On the other hand, hereafter when we use $*$, then `x` and `y` are **not** aliased, just as was the case before we added fieldwise separation. We do not use the ambiguous version of $*$.

Unfortunately, separating fields by `@A` and `@M` is not enough. For example, a field may be required by "both halves" of the separation; alternatively, we may want to restrict how some fields are used more precisely.

For example, in the threaded tree example discussed above, we noted that the "tree" operations also need access to the `data` field to support log-time access. However, clients often want to know that even though `data` may be accessed by the tree operations, it is never modified.

To support these use cases we can also mark a field immutable `@I` along the lines of David *et al.* [27]. The same field can be present (*i.e.*, not absent `@A`) on both sides of an overlaid conjunction $\mathbb{A}$ as long as both sides are `@I`. In addition, any mutable field can be "downgraded" into an immutable field. Our annotations are thus a kind of "poor man's fractional permissions [14]", in which `@A` is analogous to the empty permission, `@M` is analogous to the full permission, and `@I` is analogous to an existentialized permission. Although less precise than fractional permissions, these annotations are sufficient for a number of interesting examples and by using them we avoid some of the hassles of integrating fractional permissions into a verification tool [67].

We are now ready to give an intuition for our notion of *compatible sharing*: essentially, a conjunction ($\wedge$, $\mathbb{A}$, and $*$) expresses compatible sharing when one side can be safely framed away. Or, in other words it is possible to reason over only one side of conjunction and ignore the other since they can be combined together later without conflicts. As the simplest example, the following pairs are compatible because the separating conjunction guarantees that they exist on disjoint heaps:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, \mathtt{s}_1 \rangle * \mathtt{y}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2 \rangle$$

Consider next the following two uses of classical conjunction $\wedge$:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A} \rangle \wedge \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, @\mathtt{A} \rangle$$
$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1 @\mathtt{I}, @\mathtt{A} \rangle \wedge \mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2 @\mathtt{I}, @\mathtt{A} \rangle$$

The difference between the two formulae is that in the second example we have marked the field `fst` as immutable `@I`. Because `fst` is mutable `@M` in the first example, we are not able to frame away half of the conjunction, since we need to maintain the fact that $\mathtt{f}_1 = \mathtt{f}_2$. On the other hand, in the second example, since `fst` is immutable on both sides of the conjunction, we are able to frame away either side. Therefore, we deem the first example incompatible while we consider the second compatible.

Checking for compatibility is useful not only for the $\wedge$ operator but also for $\mathbb{A}$ operator in the presence of aliasing as shown in the following examples:

$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A} \rangle \mathbb{A} \mathtt{y}{\mapsto}\mathtt{pair}\langle \mathtt{f}_2, \mathtt{s}_2 \rangle \quad (Incompatible)$$
$$\mathtt{x}{\mapsto}\mathtt{pair}\langle \mathtt{f}_1, @\mathtt{A} \rangle \mathbb{A} \mathtt{y}{\mapsto}\mathtt{pair}\langle @\mathtt{A}, \mathtt{s}_2 \rangle \quad (Compatible)$$

Our examples so far are for simple pairs. As we will see next, our operators

are especially useful in the context of more complex objects, such as user defined inductive predicates.

5.2.2 Shared Process Scheduler

Consider the implementation of a process scheduler, a key data structure for such an implementation is the list of processes currently in the system. Assume that for simplicity, a process may be in only two states, either running or sleeping. In order to efficiently traverse the list of processes we maintain pointers to the next running or sleeping process as well. The data structure can be represented by the following declaration.

```
data node { int id; node next; node rnext; node snext }
```

The node object consists of an integer field denoting the process identifier (`id`). The `next` field points to the next process in the list of all processes. The `rnext` field points to the next running process and `snext` field points to the next sleeping process. In the list of running processes we can mark the `snext` field as absent (`@A`) while in the list of sleeping processes we can mark the `rnext` field as absent. Thus the same set of nodes have multiple views (lists) representing the processes in the system. We use the following three predicates to describe list of all processes (`al`), running processes (`rl`) and sleeping processes (`sl`).

$$\begin{aligned}
&\mathtt{al}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{q}, \mathtt{@A}, \mathtt{@A}\rangle * \mathtt{al}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S}{=}\mathtt{S_q} \cup \{\mathtt{root}\}) \\
&\mathtt{rl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{@A}, \mathtt{q}, \mathtt{@A}\rangle * \mathtt{rl}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S}{=}\mathtt{S_q} \cup \{\mathtt{root}\}) \\
&\mathtt{sl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{@A}, \mathtt{@A}, \mathtt{q}\rangle * \mathtt{sl}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S}{=}\mathtt{S_q} \cup \{\mathtt{root}\})
\end{aligned}$$

A key safety property for this process scheduler is that all processes in the list `al` should also be in either the list `rl` or the list `sl`. Note that the use of field annotations inside the definition of the predicates ensures that `rl` can only access the running processes as the other fields are marked `@A`. Also the `id` field of node in `al`, `rl` and `sl` is marked immutable with `@I`. The set of addresses reachable from the root are captured using the predicate parameter `S`. We can specify the key invariant for these list of processes using the following formula.

$$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x}\rangle \wedge (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y}\rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z}\rangle) \wedge \mathtt{S_x}{=}\mathtt{S_y} \cup \mathtt{S_z}$$

Even though this formula uses compatible sharing of heaps, it is non-trivial to prove that automatically. Since the field annotations are hidden inside the predicate definition, they cannot be exposed without doing an unfolding of the predicate. In order to expose the information about the fields inside the predicate we introduce the notion of *memory specifications*. We allow the user to specify the memory footprint of the predicate using the **mem** construct which is associated with the predicate definition. The enhanced predicate

definitions for the process scheduler are shown below.

$$
\begin{aligned}
&\mathtt{al}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{q}, \mathtt{@A}, \mathtt{@A}\rangle * \mathtt{al}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S} = \mathtt{S_q} \cup \{\mathtt{root}\}) \\
&\ \mathbf{mem}\, \mathtt{S} \hookrightarrow (\mathtt{node}\langle \mathtt{@I}, \mathtt{@M}, \mathtt{@A}, \mathtt{@A}\rangle) \\
&\mathtt{rl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{@A}, \mathtt{q}, \mathtt{@A}\rangle * \mathtt{rl}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S} = \mathtt{S_q} \cup \{\mathtt{root}\}) \\
&\ \mathbf{mem}\, \mathtt{S} \hookrightarrow (\mathtt{node}\langle \mathtt{@I}, \mathtt{@A}, \mathtt{@M}, \mathtt{@A}\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{sl}\langle \mathtt{root}, \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\}) \\
&\quad \vee \exists\, \mathtt{d}, \mathtt{q}, \mathtt{S_q} \cdot (\mathtt{root} \mapsto \mathtt{node}\langle \mathtt{d@I}, \mathtt{@A}, \mathtt{@A}, \mathtt{q}\rangle * \mathtt{sl}\langle \mathtt{q}, \mathtt{S_q}\rangle \\
&\qquad \wedge \mathtt{S} = \mathtt{S_q} \cup \{\mathtt{root}\}) \\
&\ \mathbf{mem}\, \mathtt{S} \hookrightarrow (\mathtt{node}\langle \mathtt{@I}, \mathtt{@A}, \mathtt{@A}, \mathtt{@M}\rangle)
\end{aligned}
$$

The **mem** construct consists of a memory region along with a list of possible field annotations that the predicate unfolding would generate. It allows us to syntactically check if two predicates that share memory region have compatible field annotations. Looking at the memory specification of `al` and `rl` it is easy to see that `al` does not affect (or is compatible with) `rl`. The `id` field is immutable in `rl` and the only field which is mutable in `al` is absent in `rl`. Thus any updates made to the nodes in memory region `S` using predicate `al` will not have any effect when accessing the same memory region using predicate `rl`.

To avoid writing such verbose predicates with set of addresses and to make the specifications more concise, we use the overlaid conjunction operator ($\mathbb{A}$). Formulas using the $\mathbb{A}$ operator are translated automatically to those that use the $*$ operator with memory specifications. For the shared process scheduler the

memory region shared by the lists al is same as the one shared by rl and sl. The 🙏 operator provides the hint to the system to force the memory on both sides to be the same. Hence, the key invariant of the data structure is captured more concisely as:

$$\mathtt{al}\langle \mathtt{x} \rangle \text{ 🙏 } (\mathtt{rl}\langle \mathtt{y} \rangle * \mathtt{sl}\langle \mathtt{z} \rangle)$$

This formula is automatically translated by first enhancing the predicate definitions with memory specifications by using the XMem function from figure 5.2. And then forcing the memory region on both sides of 🙏 to be the same. As the final translated formula is exactly the same as given before, the use of 🙏 provides a specification mechanism to precisely describe the user intention.

//Provided by User

$\mathtt{al}\langle \mathtt{x} \rangle \text{ 🙏 } (\mathtt{rl}\langle \mathtt{y} \rangle * \mathtt{sl}\langle \mathtt{z} \rangle)$

//Predicate extension with mem

$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \text{ 🙏 } (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle)$

//Translated form

$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \wedge (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle) \wedge \mathtt{S_x} = \mathtt{S_y} \cup \mathtt{S_z}$

Using the 🙏 operator makes the specification of methods utilizing overlaid structures less verbose. Consider the following insert method which is called while scheduling a new process in the system. The new process has to be inserted into al, and depending on the status flag, also in rl or sl. The precondition of the method uses the 🙏 operator to specify the key safety property. The use of overlaid sharing operator allows the user to express the precondition in a concise form. Compatible sharing is used to verify this method as the inserts made to different lists can be shown not to interfere with

each other.

$$
\begin{array}{l}
\texttt{void insert(int id, int status, node x, node y, node z)} \\
\texttt{requires} \quad \texttt{al}\langle\texttt{x}\rangle \mathbin{@} (\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle) \wedge \texttt{status=1} \\
\texttt{ensures} \quad \texttt{al}\langle\texttt{x}\rangle \mathbin{@} (\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle) \\
\texttt{requires} \quad \texttt{al}\langle\texttt{x}\rangle \mathbin{@} (\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle) \wedge \texttt{status=0} \\
\texttt{ensures} \quad \texttt{al}\langle\texttt{x}\rangle \mathbin{@} (\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle) \\
\{ \\
\quad \texttt{node tmp} = \texttt{new node}(\texttt{id}, \texttt{null}, \texttt{null}, \texttt{null}); \\
\quad \texttt{tmp.next} = \texttt{x}; \\
\quad \texttt{x} = \texttt{tmp}; \\
\quad \texttt{if}(\texttt{status} == 1) \\
\qquad \texttt{y} = \texttt{rlinsert}(\texttt{y}, \texttt{tmp}); \\
\quad \texttt{else z} = \texttt{slinsert}(\texttt{z}, \texttt{tmp}); \}
\end{array}
$$

5.2.3 Comparison with Fractional Permissions

In this section, we show the difficulties that arise when using separation logic with fractional permissions (SLfp) to represent overlaid data structures. We avoid these issues by using field annotations and overlaid conjunction operator while specifying compatible sharing in data structures.

Applying fractional permissions (as in SLfp) to fields inside inductive predicates can unintentionally change the meaning of the predicate. E.g consider the following predicate definition of an immutable binary tree in SLfp:

$$
\begin{array}{l}
\texttt{tree}\langle\texttt{root}\rangle \equiv \texttt{root=null} \\
\qquad \vee \exists\, \texttt{d}, \texttt{l}, \texttt{r} \cdot (\texttt{root} \mapsto \texttt{node}\langle\texttt{d@1/2}, \texttt{l@1/2}, \texttt{r@1/2}\rangle * \texttt{tree}\langle\texttt{l}\rangle * \texttt{tree}\langle\texttt{r}\rangle)
\end{array}
$$

We restrict the use of fields in the predicate using the fraction 1/2 to give a read-only permission. However, this predicate does not enforce a tree and is in fact a DAG. In standard SLfp the $*$ operator does not enforce strict separation,

thus the left and right children can point to the same node and combine using the 1/2 permissions given to each node. A more sophisticated permission system like tree-shares [67] can avoid this problem, but it is not known how to extend a tree-shares like model to fields.

We avoid this problem by using a definition of the $*$ operator that enforces strict object level separation. Also, we use field annotations that provide a simpler way to specify mutable, immutable and absent fields. If we use $*$ for object level separation and $\wedge$ for object level sharing then it is natural to introduce another operator $\mathbb{A}$ for object level sharing and field level separation. The overlaid conjunction ($\mathbb{A}$) is also practically useful to represent several data structures as shown in section 5.5.

5.3 Syntax and Semantics

Our specification language is based on separation logic, as given by Chin et. al. in [23]. We extend the language described in [23] with *memory enhanced* predicate definitions. The extended language is shown in figure 5.1 (we use the superscript * to denote a list of elements). $\Phi_{pr} *\!\!\rightarrow \Phi_{po}$ captures a precondition Φ_{pr} and a postcondition Φ_{po} of a method or a loop. They are abbreviated from the standard representation `requires` Φ_{pr} and `ensures` Φ_{po}, and formalized by separation logic formula Φ.

In turn, the separation logic formula is a disjunction of a heap formula and a pure formula ($\kappa \wedge \pi$). The pure part π captures a rich constraint from the domains of Presburger arithmetic, monadic set constraint or polynomial real arithmetic. We use the set constraints for representing memory regions as shown in figure 5.1. The predicate definition allows optional **mem** construct to be specified. The **mem** construct is useful in cases like the overlaid data structures where it

$$
\begin{array}{lll}
\textit{pred} & ::= & p(v^*) \equiv \Phi \;\; [\textit{inv} \;\; \pi][\textit{mem} \;\; S \hookrightarrow ([c(@u^*)]^*)] \\
\textit{mspec} & ::= & \Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po} \\
\Phi & ::= & \bigvee \; (\exists w^* \cdot \kappa \wedge \pi)^* \\
\kappa & ::= & \texttt{emp} \mid v \mapsto c(v[@u]^*) \mid p(v^*) \mid \kappa_1 \sharp \kappa_2 \;\; (\sharp \in \{*, \wedge, \mathbb{A}\}) \\
\pi & ::= & \alpha \mid \pi \wedge \varphi \quad \alpha ::= \beta \mid \neg \beta \\
\beta & ::= & v_1 = v_2 \mid v = \texttt{null} \mid a \leq 0 \mid a = 0 \\
a & ::= & k \mid k \times v \mid a_1 + a_2 \mid \max(a_1,a_2) \mid \textit{min}(a_1,a_2) \\
\varphi & ::= & v \in S \mid S_1 = S_2 \mid S_1 \subset S_2 \mid \forall v \in S \cdot \pi \mid \exists v \in S \cdot \pi \\
S & ::= & S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 - S_2 \mid \{\} \mid \{v\} \\
u & ::= & M \mid I \mid A \quad (M <: I <: A) \\
\textit{where} & & p \textit{ is a predicate name}; \; v, w \textit{ are variable names}; \\
& & c \textit{ is a data type name}; \; u \textit{ is a field annotation};
\end{array}
$$

Figure 5.1: Specification Language

is important to be able to specify that the memory regions of both overlaying structures are exactly the same.

For predicate definition, we also declare a pure invariant (*inv* π) that is valid for each instance of the predicate. For predicates that also have **mem**, we do not allow the set of addresses S to contain any null pointers. Hence, whenever the predicate has a memory specification, we strengthen the invariant by automatically adding a constraint using the $addMemInv(\pi, S)$ function as shown below.

$$addMemInv(\pi, S) \;\; =_{df} \;\; \pi \wedge (\forall x \in S \cdot x \neq \texttt{null})$$

Before we can use the memory specification in the entailment we need to check whether the predicate definition implies the memory specified by the user. In order to do that we take help of the $XMem(\kappa)$ function. The $XMem(\kappa)$ function, whose definition is given in figure 5.2, returns a sound approximation of the memory footprint of heap κ as a tuple of the form:

$(S, [c(@u^*)]^*)$ which corresponds to the set of addresses and the list of field annotations used in memory specifications.

The function $isData(c)$ returns `true` if c is a data node, while $isPred(c)$ returns true if c is a heap predicate. We use lists L_1 and L_2 to represent the field annotations. The function $union(L_1, L_2)$ returns the union of lists L_1 and L_2.

We do not need to consider the pure formula π in $XMem$ as it doesn't correspond to any heap. In general, Φ can be disjunctive, so we can have a number of possible approximations of memory for a predicate, each corresponding to a particular disjunct.

$$XMem(\texttt{emp}) =_{df} (\{\}, [])$$

$$\frac{isData(c)}{XMem(c\langle p, v@u^*\rangle) =_{df} (\{p\}, [c\langle @u^*\rangle])}$$

$$\frac{isPred(c) \quad c\langle p, S, v^*\rangle \equiv \Phi[\mathit{inv}\ \ \pi][\mathit{mem}\ \ S \hookrightarrow L]}{XMem(c\langle p, S, v^*\rangle) =_{df} (S, L)}$$

$$\frac{XMem(\kappa_1)=(S_1, L_1) \quad XMem(\kappa_2)=(S_2, L_2)}{XMem(\kappa_1 \sharp \kappa_2) =_{df} (S_1 \cup S_2, union(L_1, L_2))}$$

Figure 5.2: XMem: Translating to Memory Form

We illustrate how the approximation function works by using the example of a linked list.

$$\texttt{data node } \{ \texttt{ int val; node next } \}$$

$$\texttt{ll}\langle \texttt{root}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\{\})$$

$$\quad \vee \exists\, \texttt{d}, \texttt{q}, \texttt{S}_\texttt{q} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d}, \texttt{q}\rangle * \texttt{ll}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{q} \cup \{\texttt{root}\})$$

$$\mathbf{mem}\ \texttt{S} \hookrightarrow (\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle)$$

As an example consider the memory approximation of the following

predicate.

$$XMem(\texttt{x}{\mapsto}\texttt{node}\langle \texttt{d}, \texttt{p}\rangle * \texttt{ll}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle)$$

We proceed by using the rules from figure 5.2 for the data node x and predicate ll.

$$XMem(\texttt{x}{\mapsto}\texttt{node}\langle \texttt{d}, \texttt{p}\rangle) = (\{\texttt{x}\}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle])$$
$$XMem(\texttt{ll}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle) = (\texttt{S}_\texttt{y}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle])$$
$$XMem(\texttt{x}{\mapsto}\texttt{node}\langle \texttt{d}, \texttt{p}\rangle * \texttt{ll}\langle \texttt{y}, \texttt{S}_\texttt{y}\rangle) = (\{\texttt{x}\} \cup \texttt{S}_\texttt{y}, [\texttt{node}\langle @\texttt{M}, @\texttt{M}\rangle])$$

As a consistency check on the memory specification, we use the predicate definition to validate the user supplied memory specification. In case, the user doesn't provide a memory specification (e.g. when using the $\Ash$ operator), we automatically extend the predicate definition with set of addresses returned by the $XMem$ function.

We use an existing underlying [27] entailment procedure (denoted by $\vdash$) to discharge the entailment during validation of memory specifications. The rules for checking the memory specification are given in figure 5.3. In the following discussion for brevity, we represent a list of field annotations used in memory specification ($c(@u^*)^*$) with L. We define a $subtype(L_1, L_2)$ function on lists of field annotations. The function returns true if all the field annotations of data nodes in L_1 have a corresponding node in L_2 and their field annotations are in the subtyping relation (as defined in figure 5.1).

$$subtype(L_1, L_2) \quad =_{df} \quad \forall\, c(@u_1^*)\ in\ L_1, \exists\, c(@u_2^*)\ in\ L_2$$
$$s.t.\ u_1 <: u_2$$

The $subtype$ function is used to check the validity of the memory specification by ensuring that the field annotations defined inside the predicate are really subtype of those given by the memory specification. For a predicate $p(v^*) \equiv \Phi \quad mem \quad S \hookrightarrow L$, the following judgment defines the validity of the memory specification.

$$\Phi \vdash_{mem} S \hookrightarrow L$$

$$\frac{\begin{array}{c}[\textbf{CHECK}-\textbf{MEM}] \\ \Phi = \exists w^* \cdot \kappa \wedge \pi \\ XMem(\kappa) = (S_x, L_x) \\ \Phi \vdash^{\kappa}_{V,I} (S = S_x) * \Delta \\ subtype(L, L_x) \wedge subtype(L_x, L)\end{array}}{\Phi \vdash_{mem} \mathcal{S} \hookrightarrow L}$$

$$\frac{\begin{array}{c}[\textbf{CHECK}-\textbf{OR}-\textbf{MEM}] \\ \Phi_1 = \exists w_1^* \cdot \kappa_1 \wedge \pi_1 \quad \Phi_2 = \exists w_2^* \cdot \kappa_2 \wedge \pi_2 \\ XMem(\kappa_1) = (S_1, L_1) \quad XMem(\kappa_2) = (S_2, L_2) \\ \Phi_1 \vdash^{\kappa}_{V,I} (S = S_1) * \Delta \quad \Phi_2 \vdash^{\kappa}_{V,I} (S = S_2) * \Delta \\ subtype(L, union(L_1, L_2)) \wedge subtype(union(L_1, L_2), L)\end{array}}{\Phi_1 \vee \Phi_2 \vdash_{mem} \mathcal{S} \hookrightarrow L}$$

Figure 5.3: Validating the Memory Specification

Rule [CHECK–MEM] is used when the Φ formula doesn't contain a disjunction, while the rule [CHECK–OR–MEM] is used for the disjunctive case. The main difference in the disjunctive case is in the handling of list of field annotations. For the set of addresses (S), we can approximate the heap in each disjunctive formula. However, the field annotations have to be computed for the entire predicate as the annotations may differ in different disjuncts. Since memory specifications are essential to check compatibility in data structures, we have machine checked the soundness proof of these rules and the $XMem$ function in Coq. Appendix B shows the details of the certified proof. For a

given formula in our specification language P, we prove that the $XMem$ transformation preserves the satisfiability of the formula.

$$SAT(P) \implies SAT(XMem(P))$$

Since the $XMem$ function is similar to the $XPure$ function from [27], we have also checked the soundness of the $XPure$ function in Coq. We prove the following statement which signifies that the $XPure$ transformation also preserves the satisfiability of the formula.

$$SAT(P) \implies SAT(XPure(P))$$

We discovered a bug in the previous paper and pen proof given in [27] (a missing extra condition, $p \neq 0$). Interestingly, this condition is also omitted from the proof in [23].

5.3.1 Storage Model

The storage model is similar to classical separation logic [96], with the difference that we support field annotations, memory specifications and sharing operators. Accordingly, we define our storage model by making use of a domain of heaps, which is equipped with a partial operator for gluing together disjoint heaps. $\mathtt{h}_0 \cdot \mathtt{h}_1$ takes the union of partial functions when $\mathtt{h}_0$ and $\mathtt{h}_1$ have disjoint domains of definition, and is undefined when $\mathtt{h}_0(\mathtt{l})$ and $\mathtt{h}_1(\mathtt{l})$ are both defined for at least one location $\mathtt{l} \in \mathit{Loc}$.

To define the model, we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in \mathit{Val}$ denoting `null`, *Var* of variables (program and logical variables), and *ObjVal* of object values stored in the heap,

with $c[f_1 \mapsto \nu_1, .., f_n \mapsto \nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. Each field has an attached annotation from $\{M, I, A\}$. I means that the corresponding field value cannot be modified, while M allows its mutation, and A denotes no access.

$$
\begin{array}{rcll}
h & \in & \mathit{Heaps} & =_{df} \quad \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{ObjVal} \times \{M, I, A\} \\
s & \in & \mathit{Stacks} & =_{df} \quad \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}
\end{array}
$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [96, 47].

5.3.2 Semantic Model of the Specification Formula

The semantics of our separation heap formula is similar to the model given for separation logic [96], except that we have extensions to handle our user-defined heap predicates together with the field annotations and new sharing operators. Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy the constraint Φ. Function $\mathit{dom}(f)$ returns the domain of function f. Now we use $\mapsto$ to denote mappings, not the points-to assertion in separation logic. The model relation for separation heap formulae is given in definition 2. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s.

The last case in definition 2 is split into two cases: (1) c is a data node defined in the program *P*; (2) c is a heap predicate defined in the program *P*. In the first case, h has to be a singleton heap. In the second case, the heap predicate c may be inductively defined. Note that the semantics for an inductively defined heap predicate denotes the least fixpoint, i.e. for the set of states (s, h) satisfying the predicate. The monotonic nature of our heap predicate definition guarantees the

Definition 2 (Model for Specification Formula)**.**

$$
\begin{array}{lll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \textit{ or } s,h \models \Phi_2 \\
s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi & \texttt{iff} & \exists \nu_{1..n} \cdot s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n], h \models \kappa \textit{ and} \\
 & & \quad s[v_1 \mapsto \nu_1, .., v_n \mapsto \nu_n] \models \pi \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot h_1 \perp h_2 \textit{ and } h = h_1 \cdot h_2 \textit{ and} \\
 & & \quad s,h_1 \models \kappa_1 \textit{ and } s,h_2 \models \kappa_2 \\
s,h \models \kappa_1 \wedge \kappa_2 & \texttt{iff} & s,h \models \kappa_1 \textit{ and } s,h \models \kappa_2 \\
s,h \models \kappa_1 \mathbb{A} \kappa_2 & \texttt{iff} & s,h \models \kappa_1 \textit{ and } s,h \models \kappa_2 \textit{ and } Compatible(\kappa_1 \mathbb{A} \kappa_2) \\
s,h \models \texttt{emp} & \texttt{iff} & \textit{dom}(h) = \emptyset \\
s,h \models \texttt{c}(\texttt{x}, \texttt{v}_{\texttt{1..n}} @ \texttt{u}_{\texttt{1..n}}) & \texttt{iff} & \texttt{data}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \in P, \\
 & & \quad h = [s(x) \mapsto r],\ \textit{dom}(h) = \{x\} \\
 & & \quad \textit{and } r = c[f_1 \mapsto_{w_1} s(v_1), .., f_n \mapsto_{w_n} s(v_n)] \textit{ and } u_i <: w_i \\
 & & \quad \texttt{or}\ (c\langle x, v_{1..n} \rangle \equiv \Phi\ \texttt{inv}\ \pi) \in P \textit{ and } s,h \models [x/\texttt{root}]\Phi
\end{array}
$$

existence of the descending chain of unfoldings, thus the existence of the least solution.

To define the *overlaid* conjunction operator ($\mathbb{A}$) we must first identify the pairs of field annotations that are compatible. The following table can be used to look up compatible field annotations.

Table 5.1: Compatible pairs of Field Annotations

u_1	u_2	$Compatible_{FA}$
$@M$	$@M$	`false`
$@M$	$@I$	`false`
$@M$	$@A$	`true`
$@I$	$@I$	`true`
$@I$	$@A$	`true`
$@A$	$@A$	`true`

Based on $Compatible_{FA}$, we can now give the definition for the *overlaid* conjunction operator. As shown in definition 2 the case $\mathbb{A}$ is similar to $\wedge$, except

that the shared heaps must be compatible which can be checked using the $Compatible$ function given below.

$$Compatible(\kappa_1 \mathbb{A} \kappa_2) \quad =_{df}$$
$$(S_1, L_1){=}XMem(\kappa_1) \quad (S_2, L_2){=}XMem(\kappa_2)$$
$$\forall\, c(@u_1^*)\ in\ L_1, \quad \exists\, c(@u_2^*)\ in\ L_2\ s.t.\ Compatible_{FA}(u_1, u_2)$$
$$\forall\, c(@u_2^*)\ in\ L_2, \quad \exists\, c(@u_1^*)\ in\ L_1\ s.t.\ Compatible_{FA}(u_2, u_1)$$

The rest of the operators are defined in the standard way as in classical separation logic.

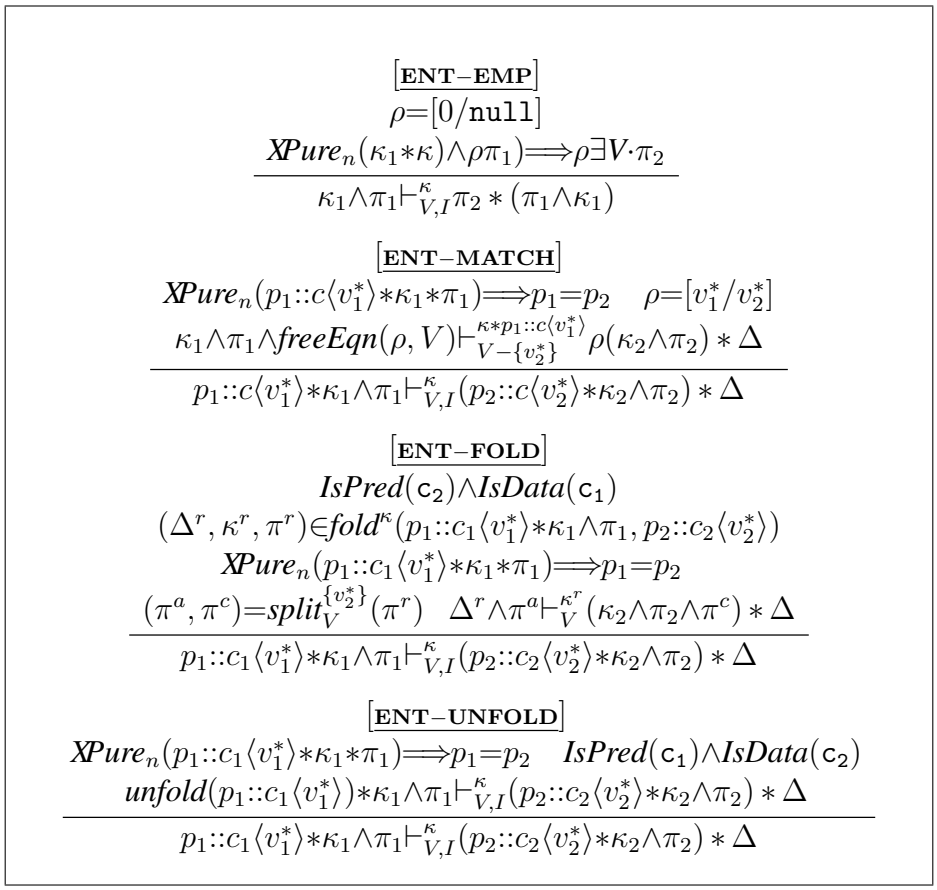
5.4 Verification with Compatible Sharing

To verify programs with compatible sharing, we make use of an existing entailment procedure for separation logic (denoted by $\vdash$ [23]). The inference rules for the entailment procedure are the same as in [23] and are reproduced below (figures 5.4 and 5.5) for completeness.

The only additional operator we have is the *overlaid* conjunction. We first describe the automatic translation used to eliminate $\mathbb{A}$ operator.

As shown in figure 5.6, the [ELIM–OVER–CONJ] rule first checks for compatible sharing of heaps (using $Compatible$ function) and then uses the $XMem$ function to get the set of addresses $S1$ and $S2$ which are added to the formula when $\mathbb{A}$ operator is replaced with $*$. Thus for the process scheduler example from section 5.2 we get the following.

$$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \ \mathbb{A}\ (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle) \quad \leadsto$$
$$\mathtt{al}\langle \mathtt{x}, \mathtt{S_x} \rangle \wedge (\mathtt{rl}\langle \mathtt{y}, \mathtt{S_y} \rangle * \mathtt{sl}\langle \mathtt{z}, \mathtt{S_z} \rangle) \wedge \mathtt{S_x} {=} \mathtt{S_y} \cup \mathtt{S_z}$$



$$\frac{\rho{=}[0/\texttt{null}] \quad \mathit{XPure}_n(\kappa_1{*}\kappa){\wedge}\rho\pi_1){\Longrightarrow}\rho{\exists}V{\cdot}\pi_2}{\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}\pi_2 * (\pi_1{\wedge}\kappa_1)} \;[\textbf{ENT-EMP}]$$

$$\frac{\mathit{XPure}_n(p_1{::}c\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \quad \kappa_1{\wedge}\pi_1{\wedge}\mathit{freeEqn}(\rho,V){\vdash}^{\kappa*p_1::c\langle v_1^*\rangle}_{V-\{v_2^*\}}\rho(\kappa_2{\wedge}\pi_2) * \Delta}{p_1{::}c\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(p_2{::}c\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \Delta} \;[\textbf{ENT-MATCH}]$$

$$\frac{\mathit{IsPred}(\texttt{c}_2){\wedge}\mathit{IsData}(\texttt{c}_1) \quad (\Delta^r,\kappa^r,\pi^r){\in}\mathit{fold}^{\kappa}(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1, p_2{::}c_2\langle v_2^*\rangle) \quad \mathit{XPure}_n(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad (\pi^a,\pi^c){=}\mathit{split}^{\{v_2^*\}}_V(\pi^r) \quad \Delta^r{\wedge}\pi^a{\vdash}^{\kappa^r}_V(\kappa_2{\wedge}\pi_2{\wedge}\pi^c) * \Delta}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \Delta} \;[\textbf{ENT-FOLD}]$$

$$\frac{\mathit{XPure}_n(p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{*}\pi_1){\Longrightarrow}p_1{=}p_2 \quad \mathit{IsPred}(\texttt{c}_1){\wedge}\mathit{IsData}(\texttt{c}_2) \quad \mathit{unfold}(p_1{::}c_1\langle v_1^*\rangle){*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \Delta}{p_1{::}c_1\langle v_1^*\rangle{*}\kappa_1{\wedge}\pi_1{\vdash}^{\kappa}_{V,I}(p_2{::}c_2\langle v_2^*\rangle{*}\kappa_2{\wedge}\pi_2) * \Delta} \;[\textbf{ENT-UNFOLD}]$$

Figure 5.4: Entailment - Base Case

Figure 5.6 also lists some of the other rules required during entailment with field annotations. These rules are based on the definition of field annotations and the semantic model of the specification formula. Rule [DOWNCAST–FA] says that we can always downcast a field annotation. It follows directly from the last case in definition 2. This means that a *write* ($@M$) annotation can be downcast to *read* ($@I$) and a *read* annotation to *absent* ($@A$). The following examples illustrate how [DOWNCAST–FA] rule can be used to check validity of entailments with field annotations.

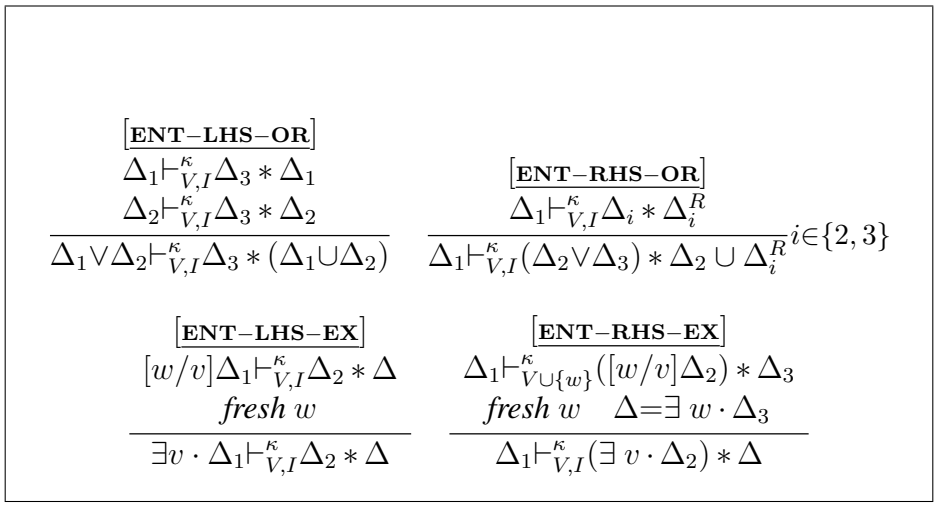


Figure 5.5: Entailment - Inductive Cases

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@M},\texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@A}) \quad (\mathit{Valid})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@A}) \quad (\mathit{Valid})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@M},\texttt{p@A}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@A}) \quad (\mathit{Valid})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@A},\texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@A}) \quad (\mathit{Invalid})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@M},\texttt{p@A}) \quad (\mathit{Invalid})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@M},\texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I},\texttt{p@M}) \quad (\mathit{Invalid})$$

The *absent* annotation can always be split off (or combined with) any other annotation as shown in rule [SPLIT–COMBINE–FA]. Finally, as given in rule [SPLIT–READ–FA] the *read* annotation can be split into two *read* annotations. Together, these three set of rules allow exclusive write access and shared read access to fields. Entailments showing the use of [SPLIT–COMBINE–FA] rule are given below.

$$[\textbf{ELIM}-\textbf{OVER}-\textbf{CONJ}]\quad \frac{\mathit{Compatible}(\kappa_1 \mathbb{\&} \kappa_2) \quad (S_1, L_1){=}\mathit{XMem}(\kappa_1) \quad (S_2, L_2){=}\mathit{XMem}(\kappa_2)}{\kappa_1 \mathbb{\&} \kappa_2 \rightsquigarrow \kappa_1 \wedge \kappa_2 \wedge S_1{=}S_2}$$

$$[\textbf{DOWNCAST}-\textbf{FA}]\quad x{\mapsto}c(v[@u]^*) \Longrightarrow_{u<:w} x{\mapsto}c(v[@w]^*)$$

$$[\textbf{SPLIT}-\textbf{COMBINE}-\textbf{FA}]\quad x{\mapsto}c(v[@u]^*) \iff x{\mapsto}c(v[@u]^*) \wedge x{\mapsto}c(v[@A]^*)$$

$$[\textbf{SPLIT}-\textbf{READ}-\textbf{FA}]\quad x{\mapsto}c(v[@I]^*) \iff x{\mapsto}c(v[@I]^*) \wedge x{\mapsto}c(v[@I]^*)$$

Figure 5.6: Rules with Field Annotations

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@M}, \texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@I}) \wedge \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@A})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@M}, \texttt{p@M}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@M}, \texttt{p@A}) \wedge \texttt{x}{\mapsto}\texttt{node}(\texttt{v@A}, \texttt{p@M})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@A}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@A}) \wedge \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@A})$$

$$\texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@I}) \vdash \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@I}) \wedge \texttt{x}{\mapsto}\texttt{node}(\texttt{v@I}, \texttt{p@A})$$

5.4.1 Forward Verification Rules

We now present the inference rules for Hoare's triples used for forward verification with compatible sharing. The rules below are reproduced from [23] with modifications made to incorporate field annotations required for compatible sharing. Verification of a method starts with each precondition, and proves that the postcondition is guaranteed at the end.

$$[\underline{\textbf{FV}-[\textbf{METH}]}]$$

$$\frac{\begin{array}{c} V{=}\{v_m..v_n\} \quad W{=}\mathit{prime}(V) \\ \forall i=1,..,p \cdot (\ \vdash \{\Phi^i_{pr}{\wedge}\mathit{nochange}(V)\}\, e\, \{\Psi^i_1\} \\ (\exists W{\cdot}\Psi^i_1) \vdash^{\kappa}_{V,I} \Phi^i_{po} * \Psi^i_2 \qquad \Psi^i_2{\neq}\{\} \end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\}}$$

$$[\underline{\textbf{FV}-[\textbf{CALL}]}]$$

$$\frac{\begin{array}{c} t_0\ mn((\texttt{ref}\ t_j\ v_j)_{j=1}^{m-1},(t_j\ v_j)_{j=m}^{n})\ \{\texttt{requires}\ \Phi^i_{pr}\ \texttt{ensures}\ \Phi^i_{po}\}_{i=1}^{p}\ \{e\} \in P \\ \rho{=}[v'_j/v_j]_{j=m}^{n} \qquad \Delta{\vdash}^{\kappa}_{V,I}\rho\Phi^i_{pr} * \Psi_i \quad \forall i{=}1,..,p \\ \Psi = \bigcup_{i=1}^{p} \Phi^i_{po} * \Psi_i \qquad \Psi \neq \{\} \end{array}}{\vdash \{\Delta\}\, mn(v_1..v_n)\, \{\Psi\}}$$

The verification is formalized in the rule $[\underline{\textbf{FV}-[\textbf{METH}]}]$:

- function *prime(V)* returns $\{v' \mid v \in V\}$.
- predicate *nochange(V)* returns $\bigwedge_{v\in V}(v = v')$. If $V = \{\}$, *nochange(V)=true*.
- $\exists W \cdot \Psi$ returns $\{\exists W \cdot \Psi_i | \Psi_i \in \Psi\}$.

At a method call, each of the method's precondition is checked, $\Delta{\vdash}^{\kappa}_{V,I}\rho\Phi^i_{pr} * \Psi_i$, where ρ represents a substitution of v_j by v'_j, for all $j = 1,..,n$. The combination of the residue Ψ_i and the postcondition is added to the poststate. If a precondition is not entailed by the program state Δ, the corresponding residue is not added to the set of states. The test $\Psi{\neq}\{\}$ ensures that at least one precondition is satisfied. Note that we use the primed notation for denoting the latest value of a variable. Correspondingly, $[v'_0/v_i]$ is a substitution that replaces the value v_i with the latest value of v'_0. We need to modify the rules related reading and updating of fields to the following.

$$\frac{[\underline{\textbf{FV}-[\textbf{FIELD}-\textbf{READ}]}]}{\begin{array}{c}\Delta \vdash^{\kappa}_{V,I} \mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v}_1@\mathtt{A}, \mathtt{v}_2@\mathtt{A}, .., \mathtt{v}_\mathtt{i}@\mathtt{I}, .., \mathtt{v}_\mathtt{n}@\mathtt{A}\rangle * \Psi_1 \quad fresh\ v_1..v_n \quad \Psi_1 \neq \{\} \\ \Psi_2 = \exists v_1..v_n \cdot (\Delta \wedge \mathtt{res} = v_i)\end{array}}{\vdash \{\Delta\} v.f_i \{\Psi_2\}}$$

$$\frac{[\underline{\textbf{FV}-[\textbf{FIELD}-\textbf{UPDATE}]}]}{\begin{array}{c}\Delta \vdash^{\kappa}_{V,I} \mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v}_1@\mathtt{A}, \mathtt{v}_2@\mathtt{A}, .., \mathtt{v}_\mathtt{i}@\mathtt{M}, .., \mathtt{v}_\mathtt{n}@\mathtt{A}\rangle * \Psi_1 \quad fresh\ v_1..v_n \quad \Psi_1 \neq \{\} \\ \Psi_2 = \exists v_1..v_n \cdot (\Delta * [v_0'/v_i]\mathtt{v}' \mapsto \mathtt{c}\langle \mathtt{v}_1@\mathtt{A}, \mathtt{v}_2@\mathtt{A}, .., \mathtt{v}_\mathtt{i}@\mathtt{M}, .., \mathtt{v}_\mathtt{n}@\mathtt{A}\rangle)\end{array}}{\vdash \{\Delta\} v.f_i := v_0 \{\Psi_2\}}$$

Whenever there is a field access (read or update), the current state, Δ, must contain the node to be dereferenced. For $[\underline{\textbf{FV}-[\textbf{FIELD}-\textbf{READ}]}]$ only the field that is been read is marked with @I annotation. In case of $[\underline{\textbf{FV}-[\textbf{FIELD}-\textbf{UPDATE}]}]$ the field that is updated is marked with the mutable annotation (@M). As shown in the $\texttt{Entail}_{\texttt{FA}}$ rule from section 5.4 entailing a @I or @M field from RHS with a corresponding node on LHS will consume the field from LHS. Hence, we discard the residue from the entailment (Ψ_1) and instead keep the original field annotation on LHS (Δ) so as prevent it from getting consumed.

5.4.2 Soundness

The soundness of rules given in figure 5.6 can be established using the semantic model and the definition of field annotations. We now present the proof of soundness of these rules, we start first with the rules for field annotations.

The downcast rule can be proven directly from the semantic model as a points-to assertion valid in the current heap is also valid in a weaker heap as defined by the subtyping relation between the annotations. The split-combine

rule can be proven using the fact that for all field annotations it is always the case that they are a subtype of the absent (@A) annotation. And finally for the read rule of field annotations we use the fact that immutable (@I) annotation is a subtype of itself.

Using the rules for field annotations we then prove the soundness of the elimination rule. Since, there are two ways of splitting the *overlaid* heaps, we have two cases to prove - in the first case we use the [SPLIT−COMBINE−FA] to combine them back as the fact that they are in compatible sharing means that the field annotations can only be from the pairs given in table for $Compatible_{FA}$ in section 5.3.2 and we prove the second case similarly using the [SPLIT−READ−FA] rule.

Rule [DOWNCAST−FA]:

$$
\begin{aligned}
& s, h \models x{\mapsto}c(v[@u]^*) \\
& \iff h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}u{<:}w \quad (\textit{definition 2}) \\
& \implies h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}h'{\subset}h \quad (\textit{weakening}) \\
& \iff s, h' \models x{\mapsto}c(v[@w]^*){\wedge}h'{\subset}h \quad (\textit{definition 2}) \\
& \iff s, h \models x{\mapsto}c(v[@w]^*)
\end{aligned}
$$

$$
Thus, x{\mapsto}c(v[@u]^*) \implies_{u<:w} x{\mapsto}c(v[@w]^*) \qquad \square
$$

Rule [SPLIT–COMBINE–FA]:

$$
\begin{array}{lll}
s,h \models x{\mapsto}c(v[@u]^*) & & \\
\iff h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}u{<:}w & (\textit{definition } 2) \\
\iff h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_{@A} s(v)]^*{\wedge}h'{\subset}h & (\forall u \cdot u{<:}@A) \\
\iff s,h' \models x{\mapsto}c(v[@A]^*){\wedge}h'{\subset}h & (\textit{definition } 2) \\
\iff s,h' \models x{\mapsto}c(v[@A]^*){\wedge}h'{\subset}h & \\
\quad {\wedge}s,h \models x{\mapsto}c(v[@u]^*) & \\
\iff s,h \models x{\mapsto}c(v[@A]^*){\wedge}x{\mapsto}c(v[@u]^*) & (\textit{definition } 2)
\end{array}
$$

$$
\begin{array}{l}
\textit{Thus}, x{\mapsto}c(v[@u]^*) \iff \\
\quad x{\mapsto}c(v[@u]^*) \wedge x{\mapsto}c(v[@A]^*)
\end{array}
\qquad \square
$$

Rule [SPLIT–READ–FA]:

$$
\begin{array}{lll}
s,h \models x{\mapsto}c(v[@I]^*) & & \\
\iff h{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_w s(v)]^*{\wedge}I{<:}w & (\textit{definition } 2) \\
\iff h'{=}[s(x){\mapsto}r]{\wedge}r{=}c[f{\mapsto}_{@I} s(v)]^*{\wedge}h'{\subset}h & (@I{<:}@I) \\
\iff s,h' \models x{\mapsto}c(v[@I]^*){\wedge}h'{\subset}h & (\textit{definition } 2) \\
\iff s,h' \models x{\mapsto}c(v[@I]^*){\wedge}h'{\subset}h & \\
\quad {\wedge}s,h \models x{\mapsto}c(v[@I]^*) & \\
\iff s,h \models x{\mapsto}c(v[@I]^*){\wedge}x{\mapsto}c(v[@I]^*) & (\textit{definition } 2)
\end{array}
$$

$$
\begin{array}{l}
\textit{Thus}, x{\mapsto}c(v[@I]^*) \iff \\
\quad x{\mapsto}c(v[@I]^*) \wedge x{\mapsto}c(v[@I]^*)
\end{array}
\qquad \square
$$

Rule [**ELIM–OVER–CONJ**]:

$$
\begin{aligned}
& s,h \models \kappa_1 \circledast \kappa_2 \wedge (S_1, L_1) = XMem(\kappa_1) \\
& \quad \wedge (S_2, L_2) = XMem(\kappa_2) \\
& \iff s,h \models \kappa_1 \wedge s,h \models \kappa_2 \wedge \\
& \quad Compatible(\kappa_1 \circledast \kappa_2) \wedge s \models S_1 = S_2 (= h) \qquad (\textit{definition } 2)
\end{aligned}
$$

case [**SPLIT–COMBINE–FA**] :

$$
\begin{aligned}
& \iff h = [s(\_) \mapsto r] \wedge r = c[f \mapsto_u s(\_)]^* \wedge \\
& \quad h' = [s(\_) \mapsto r] \wedge r = c[f \mapsto_{@A} s(\_)]^* \wedge h' \subset h \wedge \\
& \quad Compatible(\kappa_1 \circledast \kappa_2) \wedge s \models S_1 = S_2 \\
& \iff s,h \models \kappa_1 \wedge s, h' \models \kappa_2 \wedge h' \subset h \\
& \quad \wedge s \models S_1 = S_2 \qquad (Compatible_{FA}) \\
& \implies s,h \models \kappa_1 \wedge \kappa_2 \wedge S_1 = S_2 \qquad (\textit{definition } 2)
\end{aligned}
$$

case [**SPLIT–READ–FA**] :

$$
\begin{aligned}
& \iff h = [s(\_) \mapsto r] \wedge r = c[f \mapsto_{@I} s(\_)]^* \wedge \\
& \quad h' = [s(\_) \mapsto r] \wedge r = c[f \mapsto_{@I} s(\_)]^* \wedge h' \subset h \wedge \\
& \quad Compatible(\kappa_1 \circledast \kappa_2) \wedge s \models S_1 = S_2 \\
& \iff s,h \models \kappa_1 \wedge s, h' \models \kappa_2 \wedge h' \subset h \\
& \quad \wedge s \models S_1 = S_2 \qquad (Compatible_{FA}) \\
& \implies s,h \models \kappa_1 \wedge \kappa_2 \wedge S_1 = S_2 \qquad (\textit{definition } 2)
\end{aligned}
$$

$$
Thus, \kappa_1 \circledast \kappa_2 \rightsquigarrow \kappa_1 \wedge \kappa_2 \wedge S_1 = S_2 \qquad \square
$$

Soundness of the underlying entailment procedure (as shown in [23]) and the soundness of the rules given in figure 5.6 together establish the soundness

of verification with compatible sharing.

5.5 Experiments

We have built a prototype system using Objective Caml called HIPComp[1]. The web interface of HIPComp allows testing the examples without downloading or installing the system. The proof obligations generated by HIPComp are discharged using off-the-shelf constraint solvers (Omega Calculator [55] and Mona [56]). In addition to the examples presented in this chapter we can do automated verification of a number of challenging data structures with complex sharing. The examples are hard to reason with separation logic due to inherit sharing and aliasing in heap. For each of these examples, we verify methods that insert, find and remove nodes from the overlaid data structure. We use overlaid conjunction ($\mathbb{A}$) to concisely capture safety properties of programs, as seen by the following invariants verified in our experiments. The key invariant of the overlaid data structure can also be a composite structure which intermixes $*$ and $\mathbb{A}$ operators. It is essential to reason about compatible sharing when specifying and verifying such programs.

$$\texttt{al}\langle\texttt{x}\rangle \mathbb{A} (\texttt{rl}\langle\texttt{y}\rangle * \texttt{sl}\langle\texttt{z}\rangle) \; //\textit{Process Scheduler}$$

$$\texttt{llnext}\langle\texttt{x}\rangle \mathbb{A} \texttt{lldown}\langle\texttt{y}\rangle \; //\textit{Doubly Circular List}$$

$$\texttt{ll}\langle\texttt{x}\rangle \mathbb{A} \texttt{tree}\langle\texttt{t}\rangle \; //\textit{LL (Linked List) and Tree}$$

$$\texttt{ll}\langle\texttt{x}\rangle \mathbb{A} \texttt{sll}\langle\texttt{y}\rangle \; //\textit{LL and SortedLL}$$

$$(\texttt{ll}\langle\texttt{x}\rangle \mathbb{A} \texttt{tree}\langle\texttt{t}\rangle) * \texttt{ll}\langle\texttt{y}\rangle \; //\textit{Disk IO Scheduler}$$

The following table summarizes the suite of examples verified by HIPComp. All experiments were done on a 3.20GHz Intel Core i7-960 processor with 16GB

[1] http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/

memory running Ubuntu Linux 10.04. The first column gives the name of the program, second column lists the lines of code (including specifications) in the program. The annotation burden due to specifications is about 30% of the total number of lines of code. In the third column we show the sharing degree, it is defined as the percentage of specifications that use compatible sharing using field annotations. The sharing degree varies across examples depending on the percentage of methods that use overlaid conjunction in their specifications.

As is clear from our benchmark programs, the ability to specify sharing is important to verify these data structures. The last column (*Comp*) is the percentage of total entailments generated that make use of compatible sharing. The compatibility percentage depends on the number of entailments that make use of the [ELIM–OVER–CONJ] rule to eliminate the overlaid conjunction. The compatibility check is essential to verify sharing in these programs.

Table 5.2: Verification with Compatible Sharing

Program	*LOC*	*Time [s]*	*Sharing*	*Comp*
Parameterized List	30	0.28	100	40
Compatible Pairs	12	0.09	100	25
LL and SortedLL	175	0.61	22	22
LL and Tree	70	0.24	16	7
Doubly Circular List	50	0.41	50	32
Process Scheduler	70	0.47	33	23
Disk IO Scheduler	88	1.3	16	27

5.6 Comparative Remarks and Summary

The problem of sharing has also been explored in the context of concurrent data structures and objects [31, 112]. Our work is influenced by them but for a sequential setting, indeed the notion of self-stable concurrent abstract predicates

is analogous to our condition for compatibility. However, since we are focused on sequential programs, we avoid the use of environment actions and instead focus on checking compatibility between shared predicates. Regional logic [5] also uses set of addresses as footprint of formulas. These regions are used with dynamic frames to enable local reasoning of programs. Memory layouts [37] are used by Gast, as a way to formally specify the structure of individual memory blocks. A grammar of memory layouts enables distinguishing between variable, array, or other data structures. This shows that when dealing with shared regions of memory, knowing the layout of memory can be quite helpful for reasoning. We use field annotations to specify access to memory in shared and overlaid data structures.

Similarly, the recent work of Dragoi et al. [34] considers only the shape analysis of overlaid lists. We extend these separation logic based techniques by going beyond shape properties and handling arbitrary data structures. Our proposal is built on top of user defined predicates with shape, size and bag properties that can express functional properties (order, sorting, height balance etc.) of overlaid data structures. A separation logic based program analysis has been used to handle non-linear data structures like trees and graphs [20]. In order to handle cycles, they keep track of the nodes which are already visited using multi-sets.

We have proposed a specification mechanism to express different kinds of sharing and aliasing in data structures. The specifications can capture correctness properties of various kinds of programs using compatible sharing. We present an automated verification system which can be used to reason about sharing in data structures. We have implemented a prototype based on our approach. An initial set of experiments with small but challenging programs have confirmed the usefulness of our method. For future work, we want to

explore the use of memory regions and field annotations to enable automated verification of other intrinsic shared data structures that do not satisfy compatible sharing (like DAGs and graphs).

Chapter 6

Automated Verification of Ramifications in Separation Logic

We present an automated entailment procedure that can reason with ramified updates. We show, how to calculate ramifications for predicates representing different sharing and aliasing scenarios. We have implemented our approach and verified a small but comprehensive benchmark of challenging programs with significant heap sharing. Our experiments show that we can verify many different sharing scenarios using automated ramifications.

6.1 Introduction

Data structures with heap sharing are widely used in system software. Sharing enables more efficient use of memory and allows programmers to write compact programs. However, it can be challenging to formally reason about such programs. In addition, certain data structures like acyclic and cyclic graphs have intrinsic sharing. Sharing makes it harder to reason about different parts of the structure in isolation.

Many common data structures can be represented using the separating conjunction. For example, in a linked list the head of the list is separated from the rest of the list and in a binary tree the left and right children are separated. However, there are many data structures (like graphs) where it is not always possible to isolate them using separating conjunction. Data structures like graphs involve unrestricted sharing, the left and right children of a binary DAG (directed acyclic graph) may point into each other. Recently, Hobor and Villard [46] designed a new proof technique called ramification to deal with such cases.

The Ramification procedure presented in [46] is proven sound but is not currently handled by automated verification. Ramifications in separation logic can be represented using septraction ($-\circledast$ [113]) operator. However automated reasoning with the septraction operator has been known to be challenging. In this chapter, we present the first automated procedure to calculate ramifications and enable verification of programs using DAGs and graphs. To support automated ramifications, we use an extended form of separation logic with two additional conjunction operators, $\curlywedge$ and $\wedge$, that express different degrees of sharing and aliasing. While these operators are not new, we have formulated a new sound method 6.3 to reason with them which enables us to automatically verify several programs (using DAGs, graphs and overlapping structures) that have so far evaded compositional proofs. Our key contributions include:

- An entailment procedure to automatically reason with ramified updates.
- Use of lemmas to handle reasoning with the septraction operator.
- A prototype implementation of our approach and its evaluation on a benchmark of programs with complex heap sharing.

The rest of the chapter is structured as follows. In section 6.2, we start with some motivating examples which illustrate our method. Section 6.3 introduces the formal notion and entailment with logical operators in our system. The implementation details and experiments are presented in section 6.4. Finally, we compare with related work and conclude in section 6.5.

6.2 Motivating Examples

6.2.1 Updates on Shared Heaps

To illustrate the effect updates can have on shared heaps, we make use of a simple example of the single data node `cell`. It can be represented by the following data structure.

$$\texttt{data cell} \{ \texttt{int val} \}$$

Consider two cells `x` and `y` which may be aliased with each other. In our specification logic we can represent them using the $\Yup$ operator as follows.

$$\texttt{x}{\mapsto}\texttt{cell}\langle\texttt{f}\rangle \Yup \texttt{y}{\mapsto}\texttt{cell}\langle\texttt{s}\rangle$$

Since the operator $\Yup$ represents may aliasing an update to `val` field of one of the cells may possibly effect the other cell. This can lead to problems when calculating post conditions during verification. As an example consider the following precondition and a command that updates `val`.

$$\{\texttt{x}{\mapsto}\texttt{cell}\langle\texttt{f}\rangle \Yup \texttt{y}{\mapsto}\texttt{cell}\langle\texttt{s}\rangle \wedge \texttt{f}{\neq}1 \wedge \texttt{s}{\neq}1\}$$

$$\texttt{x.val} = 1;$$

$$\{?\}$$

Since `x` and `y` may be aliased we cannot ignore the effect of update to `x` on

y. We need to be able to calculate the effect updating x has on y. We note that this is not a problem in case of separation logic using only $*$ operator, as we can give the following pre/post conditions to the command.

$$\{\texttt{x}\mapsto\texttt{cell}\langle\texttt{f}\rangle * \texttt{y}\mapsto\texttt{cell}\langle\texttt{s}\rangle \wedge \texttt{f}\neq 1 \wedge \texttt{s}\neq 1\}$$
$$\texttt{x.val} = 1;$$
$$\{\texttt{x}\mapsto\texttt{cell}\langle 1\rangle * \texttt{y}\mapsto\texttt{cell}\langle\texttt{s}\rangle \wedge \texttt{s}\neq 1\}$$

Sharing and aliasing leads to indirect consequences for local actions. We use the $-\circledast$ operator [46] for specifying the update to the shared region. The $-\circledast$ operator can help in capturing state which may be missing some heap, it is defined as follows.

$$s, h \models \kappa_1 - \circledast \kappa_2 \quad \texttt{iff} \quad \exists h_1, h_2\ h_2 = h_1 \cdot h$$
$$s, h_1 \models \kappa_1 \text{ and } s, h_2 \models \kappa_2$$

And we can use it to express the fact that update to x may affect y as well.

$$\{\texttt{x}\mapsto\texttt{cell}\langle\texttt{f}\rangle \uplus \texttt{y}\mapsto\texttt{cell}\langle\texttt{s}\rangle \wedge \texttt{f}\neq 1 \wedge \texttt{s}\neq 1\}$$
$$\texttt{x.val} = 1;$$
$$\{\texttt{x}\mapsto\texttt{cell}\langle\texttt{f}\rangle - \circledast (\texttt{x}\mapsto\texttt{cell}\langle\texttt{f}\rangle \uplus \texttt{y}\mapsto\texttt{cell}\langle\texttt{s}\rangle) * \texttt{x}\mapsto\texttt{cell}\langle 1\rangle \wedge \texttt{f}\neq 1 \wedge \texttt{s}\neq 1\}$$

We split the update into two parts, first we take away the original cell from precondition using $-\circledast$ and then we add the updated cell back. As shown in [46] this indeed is the strongest postcondition. In general, it is quite difficult to reason with $-\circledast$ operator directly so we try to eliminate it by explicitly calculating its effect. For this example we can do a may alias case analysis to reduce the formula to one without $-\circledast$ as shown below.

$\{(\mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{f}\rangle -\circledast \mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{f}\rangle \uplus \mathtt{y}\mapsto\mathtt{cell}\langle\mathtt{s}\rangle) * \mathtt{x}\mapsto\mathtt{cell}\langle 1\rangle \wedge \mathtt{f}\neq 1 \wedge \mathtt{s}\neq 1\}$

// Fresh t

$\{(\mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{f}\rangle -\circledast \mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{f}\rangle \uplus \mathtt{y}\mapsto\mathtt{cell}\langle\mathtt{s}\rangle) * \mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{t}\rangle \wedge \mathtt{t}=1 \wedge \mathtt{f}\neq 1 \wedge \mathtt{s}\neq 1\}$

// Eliminate $-\circledast$

$\{(\mathtt{y}\mapsto\mathtt{cell}\langle\mathtt{t}\rangle \uplus \mathtt{x}\mapsto\mathtt{cell}\langle 1\rangle) \wedge (\mathtt{t}=\mathtt{s} \vee \mathtt{t}=1) \wedge \mathtt{s}\neq 1\}$

// Simplify

$\{(\mathtt{y}\mapsto\mathtt{cell}\langle\mathtt{t}\rangle \uplus \mathtt{x}\mapsto\mathtt{cell}\langle 1\rangle) \wedge (\mathtt{t}=1 \vee \mathtt{t}\neq 1)\}$

// Eliminate t

$\{(\mathtt{y}\mapsto\mathtt{cell}\langle\_\rangle \uplus \mathtt{x}\mapsto\mathtt{cell}\langle 1\rangle)\}$

We use a fresh variable t to represent the val field of y. If x and y were to be aliased after update to x, the val field of y will also be updated and we capture this fact by the pure formula t=s. If x and y were not aliased, then the val field of y remains unchanged, so t=1. We remove the $-\circledast$ operator and connect x and y with the $\uplus$ operator to denote that they may be shared (or aliased), while we add the condition (t=s$\vee$t=1) to the formula. This in turn can be further simplified to the formula at the end. With this, we can now give concise post-condition to the command from our example, as follows.

$\{\mathtt{x}\mapsto\mathtt{cell}\langle\mathtt{f}\rangle \uplus \mathtt{y}\mapsto\mathtt{cell}\langle\mathtt{s}\rangle \wedge \mathtt{f}\neq 1 \wedge \mathtt{s}\neq 1\}$

$\mathtt{x.val} = 1;$

$\{\mathtt{x}\mapsto\mathtt{cell}\langle 1\rangle \uplus \mathtt{y}\mapsto\mathtt{cell}\langle\_\rangle\}$

Thus, the user only needs to provide concise pre/post specification using the three conjunction operators, while we automatically calculate the ramifications required to verify them. A key distinguishing feature of our system when compared to [46] is that we automatically handle ramifications

during entailment. The user doesn't need to know or use the $-\circledast$ operator in the specification for verifying sharing with ramifications. This example shows how we ramify with a singleton heap. For each field we do an alias analysis for calculating the effect of the update. In case $\wedge$ operator we ramify using must alias analysis while in case of $\uplus$ operator we do a may alias analysis. Hence if x and y cells were specified using $\wedge$, we would have the following.

$$\{\mathtt{x}{\mapsto}\mathtt{cell}\langle\mathtt{f}\rangle{\wedge}\mathtt{y}{\mapsto}\mathtt{cell}\langle\mathtt{s}\rangle{\wedge}\mathtt{f}{\neq}1{\wedge}\mathtt{s}{\neq}1\}$$
$$\mathtt{x.val} \;=\; 1;$$
$$\{\mathtt{x}{\mapsto}\mathtt{cell}\langle 1\rangle{\wedge}\mathtt{y}{\mapsto}\mathtt{cell}\langle 1\rangle\}$$

6.2.2 Septraction Lemmas

In case of inductive predicates, unrestricted updates made on the shared heap can malign the shape of the structure. An example of this kind of update can be deletion of a node from a predicate using the sharing operators. In order to capture the change of shape, we allow users to relate predicates beyond their definitions by means of lemmas. In this section, we introduce a new kind of lemma called septraction lemma that can aid in verifying such algorithms. Septraction lemmas are based on a general approach described in [83] where lemmas are user-supplied, but automatically proven and systematically applied. Consider the example of deleting a node from the following overlapping structure.

$$\mathtt{cache}{\mapsto}\mathtt{node}\langle \_\,,\mathtt{p}\rangle{\uplus}\mathtt{ll}\langle \mathtt{x},\mathtt{S}\rangle$$

Suppose, we have a function, delete(x) that removes the node x from the list. We are interested in verifying the following.

$$\{\texttt{cache}{\mapsto}\texttt{node}\langle \_ , \texttt{p}\rangle \circledast \texttt{ll}\langle \texttt{x}, \texttt{S}\rangle\}$$

$$\texttt{delete}(\texttt{cache});$$

$$\{?\}$$

The difficulty in verifying this method is that the deletion of node may affect the linked list since the node may be aliased with the linked list. In that case the `ll` will not capture the broken list and we need another predicate `lseg` to capture a list segment. The list segment can be defined by the following predicate.

$$\begin{array}{l}\texttt{data node \{ int val; node next \}}\\ \texttt{lseg}\langle \texttt{root}, \texttt{p}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{S}{=}\{\}\\ \quad \vee \exists\, \texttt{q} \cdot (\texttt{root}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle * \texttt{lseg}\langle \texttt{q}, \texttt{p}, \texttt{S}_\texttt{q}\rangle)\\ \qquad \wedge \texttt{S}{=}\texttt{S}_\texttt{q}{\cup}\{\texttt{root}\}\end{array}$$

We can use the `lseg` predicate to capture the deletion of a node in the middle of the list `ll`. For the `delete` function when we know that the `cache` node is inside the list we can do the following.

$$\begin{array}{l}\{\texttt{cache}{\mapsto}\texttt{node}\langle \_ , \texttt{p}\rangle \circledast \texttt{ll}\langle \texttt{x}, \texttt{S}\rangle \wedge \texttt{cache}{\in}\texttt{S}\}\\ \quad \texttt{delete}(\texttt{cache});\\ \{\texttt{lseg}\langle \texttt{x}, \texttt{cache}, \texttt{S}_1\rangle * \texttt{ll}\langle \texttt{p}, \texttt{S}_2\rangle \wedge \texttt{S}{=}\texttt{S}_1{\cup}\texttt{S}_2{\cup}\{\texttt{cache}\}\}\end{array}$$

The removal of the node from the middle of the list leaves a list segment from the beginning to that node and the rest of the list after the node. To enable this kind of reasoning automatically during the entailment, we allow users to specify lemmas using the $-\!\circledast$ operator. For this example we use the following septraction lemma

$$\texttt{Septraction Lemma}$$

$$(\texttt{p}\mapsto\texttt{node}\langle \_ , \texttt{q}\rangle -\circledast \texttt{ll}\langle \texttt{root}, \texttt{S}\rangle)\wedge \texttt{p}\in\texttt{S} \rightarrow$$

$$\texttt{lseg}\langle \texttt{root}, \texttt{p}, \texttt{S}_1\rangle * \texttt{ll}\langle \texttt{q}, \texttt{S}_2\rangle \wedge \texttt{S}=\texttt{S}_1\cup\texttt{S}_2\cup\{\texttt{p}\}$$

To check the postcondition at the end of the method we use the given septraction lemma. The septraction lemma application is guided by a potential match with the `lseg` predicate from the body of the lemma. After the septraction lemma is applied entailment proving can verify the method. The septraction lemma itself can be proven by the same entailment procedure by applying the lemma as an instance of cyclic proof following the general approach given in [83].

6.3 Verification with Ramifications

In this section, we use the same specification language as used in section 5.3. In addition, the entailment procedure and inference rules for Hoare's triple are the same as given in section 5.4.

The use of septraction lemma allows us to precisely capture the behavior using $-\circledast$ operator. We extend the existing lemma mechanism from [83] to the one shown in figure 6.1.

We add a new kind of lemma (called septraction lemma) that uses the $-\circledast$ operator in the specification. Septraction lemmas are of the form $(E-\circledast H) \wedge G \rightarrow B$, where E denotes the heap that is taken away from H and G is a pure formula which specifies the condition under which we can apply the lemma. Septraction lemmas are proved and applied to support sound proof search by the entailment prover. Since the proof of a septraction lemma may apply the lemma itself inductively, we first present the proof rule that applies the lemmas

$$
\begin{array}{ll}
\textit{lemma}\ (L) & ::= H \wedge G \bowtie B \mid (E \mathrel{-\!\circledast} H) \wedge G \rightarrow B \\
\textit{head}\ (H) & ::= [\mathtt{root} ::] c(v^*) \\
\textit{body}\ (B) & ::= \Phi \\
\textit{extra}\ (E) & ::= \kappa \\
\textit{guard}\ (G) & ::= \pi \\
\bowtie & ::= \ \rightarrow \mid \leftarrow \mid \leftrightarrow \\
\textit{where} & c \textit{ is a data type name}; p \textit{ is a predicate name}; \\
& v, w \textit{ are variable names};
\end{array}
$$

Figure 6.1: Septraction Lemma Syntax

:

$$
[\mathbf{L{-}LEFT{-}RAMIFY}]
$$

$$
\frac{\begin{array}{c} isPred(c_1) \\ \rho{=}match(E \mathrel{-\!\circledast} H, \mathtt{p_2}{::}\mathtt{c_2}\langle \mathtt{v_2^*}\rangle \mathrel{-\!\circledast} \mathtt{p_1}{::}\mathtt{c_1}\langle \mathtt{v_1^*}\rangle) \\ (\mathtt{p_2}{::}\mathtt{c_2}\langle \mathtt{v_2^*}\rangle \mathrel{-\!\circledast} \mathtt{p_1}{::}\mathtt{c_1}\langle \mathtt{v_1^*}\rangle) {*} \kappa_1 {\wedge} \pi_1 {\vdash} \rho G \\ (\rho B) {*} \kappa_1 {\wedge} \pi_1 \vdash^{\kappa * (\mathtt{p_2}::\mathtt{c_2}\langle \mathtt{v_2^*}\rangle \mathrel{-\!\circledast} p_1::c\langle v_1^*\rangle)}_{V,I} (\kappa_2 {\wedge} \pi_2) * \Psi \end{array}}{(\mathtt{p_2}{::}\mathtt{c_2}\langle \mathtt{v_2^*}\rangle \mathrel{-\!\circledast} \mathtt{p_1}{::}\mathtt{c_1}\langle \mathtt{v_1^*}\rangle) {*} \kappa_1 {\wedge} \pi_1 \vdash^{\kappa}_{V,I} (\kappa_2 {\wedge} \pi_2) * \Psi}
$$

The septraction lemma can be applied to only the antecedent and is treated as an unfolding. Its application is formalized below which says that the lemma is applied if we can find a substitution ρ that matches H to $\mathtt{p_1}{::}\mathtt{c_1}\langle \mathtt{v_1^*}\rangle$, E to $\mathtt{p_2}{::}\mathtt{c_2}\langle \mathtt{v_2^*}\rangle$ and satisfies the guard. Entailment $\Phi{\vdash}\pi$ checks if the guard π holds under Φ, and match function is defined as:

$$
\begin{array}{c}
match(\mathtt{p_2}{::}\mathtt{c_2}\langle \mathtt{v_2^*}\rangle \mathrel{-\!\circledast} \mathtt{p_1}{::}\mathtt{c_1}\langle \mathtt{v_1^*}\rangle, \mathtt{p_4}{::}\mathtt{c_4}\langle \mathtt{v_4^*}\rangle \mathrel{-\!\circledast} \mathtt{p_3}{::}\mathtt{c_3}\langle \mathtt{v_3^*}\rangle) \\
=_{df} [p_1 {\mapsto} p_3, p_2 {\mapsto} p_4, v_1^* {\mapsto} v_3^*, v_2^* {\mapsto} v_4^*]
\end{array}
$$

For a goal-directed lemma application, we shall only apply this rule when there exists a predicate in the consequent that would (subsequently) match up

via aliasing with a predicate in the RHS of lemma ρB. To prevent non-termination during application of septraction lemmas, we assign a history to each heap constraint (κ). The history is a set of predicate names which are transitively written to κ. Lemma application is possible only if it does not rewrite a predicate to some predicate already in the history. Folding and unfolding of predicate instances pass the history on to the predicate instances in the body. A detailed worked out example of lemma proof and application is provided in section 6.3.1.

Septraction lemmas increase the precision of our entailment prover. That is, the verification succeeds in more cases than before. In addition, they are essential to verify certain algorithms. Correctness of septraction lemmas is automatically proven by our system via the entailment prover. To prove a septraction lemma, we need to show that the predicate in the head of the lemma entails the body. During this entailment proving, the lemma being proven can be soundly used in the proof itself as an instance of cyclic proof. Formally proving a septraction lemma amounts to discharging the following proof obligation:

$$unfold(E -\!\circledast H \wedge G, \texttt{root}) \vdash unfold(B, \texttt{root}) * \texttt{emp}$$

At the start of lemma proving, we always unfold the head predicate in the antecedent and consequent. This ensures that infinite descent occurs for the resulting cyclic proof which guarantees a progress condition needed for sound induction. During the lemma proving, the septraction lemma being proven may be applied to the unfolded formulas as an instance of cyclic proving. Furthermore, we also check that the entailment derives an empty residual heap. This ensures that both side of the lemma cover the same heap region.

To handle the $-\!\circledast$ operator during lemma proving we cannot use the

septraction rewriting rules, since during the proof we will not have an updated heap to ramify with. Instead the septraction operator is handled by doing a case analysis and checking the resulting entailments. The rules for doing the case splits during lemma proving are given in figure 6.2.

$$\frac{\begin{array}{c}\mathtt{S_e, L_e{=}XMem(E)} \quad \mathtt{S_1, L_1{=}XMem}(\kappa_1) \quad \mathtt{S_2, L_2{=}XMem}(\kappa_2) \\ (E \mathrel{-\!\circledast} \kappa_1) * \kappa_2 \wedge S_e \subset S_1 \wedge S_e \not\subset S_2 \wedge G \vdash B * \mathtt{emp} \\ \kappa_1 * (E \mathrel{-\!\circledast} \kappa_2) \wedge S_e \not\subset S_1 \wedge S_e \subset S_2 \wedge G \vdash B * \mathtt{emp}\end{array}}{E \mathrel{-\!\circledast} (\kappa_1 * \kappa_2) \wedge G \vdash B * \mathtt{emp}} \; [\mathbf{CASE{-}SPLIT}_*]$$

$$\frac{\begin{array}{c}\mathtt{S_e, L_e{=}XMem(E)} \quad \mathtt{S_1, L_1{=}XMem}(\kappa_1) \quad \mathtt{S_2, L_2{=}XMem}(\kappa_2) \\ (E \mathrel{-\!\circledast} \kappa_1) \wedge (E \mathrel{-\!\circledast} \kappa_2) \wedge S_e \subset S_1 \wedge S_e \subset S_2 \wedge G \vdash B * \mathtt{emp}\end{array}}{E \mathrel{-\!\circledast} (\kappa_1 \wedge \kappa_2) \wedge G \vdash B * \mathtt{emp}} \; [\mathbf{CASE{-}SPLIT}_\wedge]$$

$$\frac{\begin{array}{c}\mathtt{S_e, L_e{=}XMem(E)} \quad \mathtt{S_1, L_1{=}XMem}(\kappa_1) \quad \mathtt{S_2, L_2{=}XMem}(\kappa_2) \\ (E \mathrel{-\!\circledast} \kappa_1) \uplus (E \mathrel{-\!\circledast} \kappa_2) \wedge S_e \subset S_1 \wedge S_e \subset S_2 \wedge G \vdash B * \mathtt{emp} \\ (E \mathrel{-\!\circledast} \kappa_1) \uplus \kappa_2 \wedge S_e \subset S_1 \wedge S_e \not\subset S_2 \wedge G \vdash B * \mathtt{emp} \\ \kappa_1 \uplus (E \mathrel{-\!\circledast} \kappa_2) \wedge S_e \not\subset S_1 \wedge S_e \subset S_2 \wedge G \vdash B * \mathtt{emp} \\ \kappa_1 \uplus \kappa_2 \wedge S_e \not\subset S_1 \wedge S_e \not\subset S_2 \wedge G \vdash B * \mathtt{emp}\end{array}}{E \mathrel{-\!\circledast} (\kappa_1 \uplus \kappa_2) \wedge G \vdash B * \mathtt{emp}} \; [\mathbf{CASE{-}SPLIT}_\uplus]$$

Figure 6.2: Case Analysis for Septraction Lemma Proving

The cases depend on the sharing operators used in the unfolded predicates from the head of the lemma. We assume that the head of the lemma H is unfolded into a heap $\kappa_1 \sharp \kappa_2$, where $\sharp$ is one of the following three conjunction operators : $*, \wedge, \uplus$. We express the different cases using memory specifications via the `XMem` function. We use the set of addresses captured as part of the memory specifications to cover all possible cases for various sharing operators. Each case split has to be verified in order to prove the given septraction lemma.

In addition, we also use the following axioms. The basic axioms can be

instantiated with any data node ($\mathtt{d}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle$) or any predicate ($\mathtt{c_p}\langle\mathtt{p},\mathtt{S},\mathtt{v}^*\rangle$) with memory specifications. These axioms can be proven semantically from the definitions and are applied directly in our system. We use the septraction lemma to be proven, basic axioms, case splitting rules and the entailment prover to generate a cyclic proof by infinite descent. An illustration of the cyclic proof of a lemma is given in the section 6.3.1.

$$\mathtt{Basic\ Axioms}$$

$$\mathtt{emp}-\circledast\mathtt{d}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle \rightarrow \mathtt{d}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle$$

$$\mathtt{emp}-\circledast c\langle p,S,v^*\rangle \rightarrow c\langle p,S,v^*\rangle$$

$$\mathtt{d}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle-\circledast\mathtt{emp} \rightarrow \mathtt{emp}$$

$$c\langle p,S,v^*\rangle-\circledast\mathtt{emp} \rightarrow \mathtt{emp}$$

$$\mathtt{d_2}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle-\circledast\mathtt{d_1}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle\wedge d_1{=}d_2 \rightarrow \mathtt{emp}$$

$$\mathtt{d_2}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle-\circledast\mathtt{d_1}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle\wedge d_1{\neq}d_2 \rightarrow \mathtt{d_1}{\mapsto}\mathtt{c}\langle\mathtt{v}^*\rangle$$

$$c\langle p,S,v^*\rangle-\circledast\mathtt{d}{\mapsto}\mathtt{c_d}\langle\mathtt{v}^*_\mathtt{d}\rangle\wedge d{\in}S \rightarrow \mathtt{emp}$$

$$c\langle p,S,v^*\rangle-\circledast\mathtt{d}{\mapsto}\mathtt{c_d}\langle\mathtt{v}^*_\mathtt{d}\rangle\wedge d{\notin}S \rightarrow \mathtt{d}{\mapsto}\mathtt{c_d}\langle\mathtt{v}^*_\mathtt{d}\rangle$$

$$\mathtt{d}{\mapsto}\mathtt{c_d}\langle\mathtt{v}^*_\mathtt{d}\rangle-\circledast c\langle p,S,v^*\rangle\wedge d{\notin}S \rightarrow c\langle p,S,v^*\rangle$$

$$c_2\langle p_2,S_2,v_2^*\rangle-\circledast c_1\langle p_1,S_1,v_1^*\rangle\wedge S_1{\cap}S_2{=}\{\} \rightarrow c_1\langle p_1,S_1,v_1^*\rangle$$

6.3.1 Proof of a Septraction Lemma

Septraction lemmas are proven automatically by the entailment prover (SLEEK) using cyclic proof. We illustrate this process using the following detailed example.

$$\texttt{Septraction Lemma}$$

$$(\texttt{p}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle {-\!\circledast} \texttt{ll}\langle \texttt{root}, \texttt{S}\rangle) {\wedge} \texttt{p}{\in}\texttt{S} \rightarrow$$

$$\texttt{lseg}\langle \texttt{root}, \texttt{p}, \texttt{S}_1\rangle * \texttt{ll}\langle \texttt{q}, \texttt{S}_2\rangle {\wedge} \texttt{S}{=}\texttt{S}_1{\cup}\texttt{S}_2{\cup}\{\texttt{p}\}$$

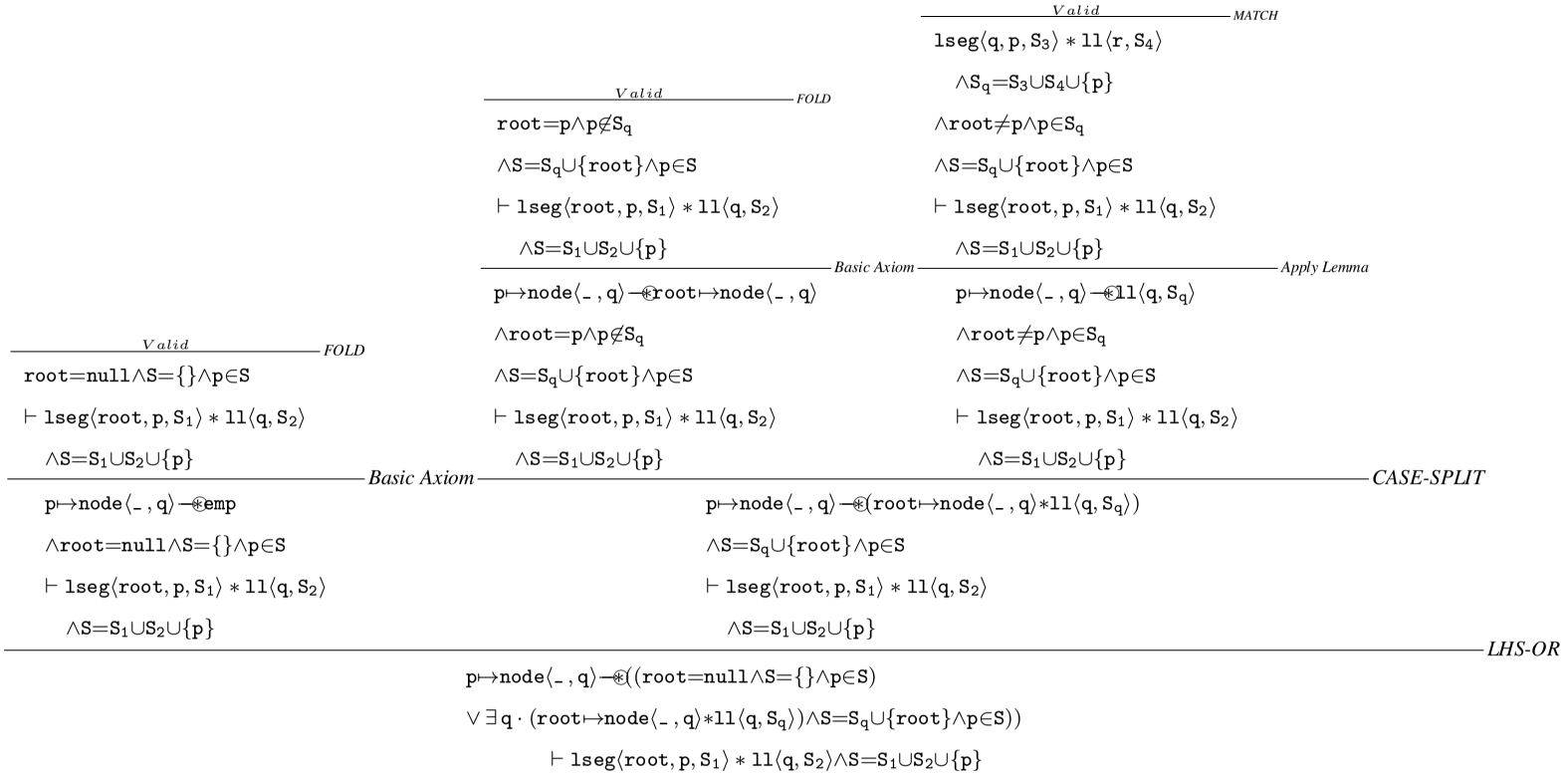
In order to prove the above septraction lemma, we need to check the entailment given below.

$$\begin{array}{l} \texttt{p}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle {-\!\circledast} ((\texttt{root}{=}\texttt{null} {\wedge} \texttt{S}{=}\{\}) \\ \quad \vee \exists\, \texttt{q} \cdot (\texttt{root}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle {*} \texttt{ll}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle) \\ \quad {\wedge} S{=}S_q{\cup}\{\texttt{root}\}{\wedge}p{\in}S) \\ \qquad\qquad \vdash \texttt{lseg}\langle \texttt{root}, \texttt{p}, \texttt{S}_1\rangle * \texttt{ll}\langle \texttt{q}, \texttt{S}_2\rangle {\wedge} \texttt{S}{=}\texttt{S}_1{\cup}\texttt{S}_2{\cup}\{\texttt{p}\} \end{array}$$

After pushing the guard of lemma into the terms of disjunctions we get

$$\begin{array}{l} \texttt{p}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle {-\!\circledast} ((\texttt{root}{=}\texttt{null} {\wedge} \texttt{S}{=}\{\} {\wedge} p{\in}S) \\ \quad \vee \exists\, \texttt{q} \cdot (\texttt{root}{\mapsto}\texttt{node}\langle \_ , \texttt{q}\rangle {*} \texttt{ll}\langle \texttt{q}, \texttt{S}_\texttt{q}\rangle) \\ \quad {\wedge} S{=}S_q{\cup}\{\texttt{root}\}{\wedge}p{\in}S) \\ \qquad\qquad \vdash \texttt{lseg}\langle \texttt{root}, \texttt{p}, \texttt{S}_1\rangle * \texttt{ll}\langle \texttt{q}, \texttt{S}_2\rangle {\wedge} \texttt{S}{=}\texttt{S}_1{\cup}\texttt{S}_2{\cup}\{\texttt{p}\} \end{array}$$

The detailed derivation tree of the cyclic proving done in SLEEK is given on the next page. In the proof, the [CASE–SPLIT$_*$] rule is used by the entailment prover (SLEEK) to do a case analysis. Then, the lemma is applied inductively as an instance of cyclic proof by infinite descent. The leaves of the proof tree show how some entailments are decided using basic axioms and others use the MATCH and FOLD rules of the underlying entailment prover SLEEK (described in [83, 23]).



Derivation Tree for Septraction Lemma Proving

6.4 Experiments

We have implemented automated reasoning with ramifications in our prototype HIPComp. Preliminary experiments were conducted by testing our system on a suite of examples as shown in table 6.1. In addition to all the examples presented in this chapter we can do automated verification with ramifications for a number of challenging data structures with sophisticated sharing.

The first example ($Ramified\ Cells$) in the experiments is the same as the first motivating example from section 6.2. The second example is similar to the first one but using pairs instead of single cells. The example with $Node\ and\ LL$ is the second motivating example from section 6.2. The last two examples are of a binary acyclic graph or DAG. A binary DAG can be specified using the $\Cup$ operator as follows:

$$\texttt{data node \{ int val; node left; node right \}}$$

$$\texttt{dag}\langle\texttt{root},\texttt{S}\rangle \equiv \texttt{root}=\texttt{null} \wedge \texttt{S}=\{\} \vee \exists\, \texttt{d},\texttt{l},\texttt{r} \cdot (\texttt{root} \mapsto \texttt{node}\langle\texttt{d},\texttt{l},\texttt{r}\rangle$$
$$* \texttt{dag}\langle\texttt{l},\texttt{S}_\texttt{l}\rangle \Cup \texttt{dag}\langle\texttt{r},\texttt{S}_\texttt{r}\rangle \wedge \texttt{S}=\{\texttt{root}\} \cup \texttt{S}_\texttt{l} \cup \texttt{S}_\texttt{r})$$

For the experiment, we verify the memory safety (shape) of a method that marks the DAG recursively. In case of the $DAG\ (Values)$ example, we also check the property that after calling the method, the DAG only contains values that are marked (in additional to the memory safety property).

Table 6.1: Experiments with Automated Ramifications

$Program$	LOC	LOS	$Timings[secs]$	$Sharing[\%]$	$Ramification[\%]$
$Ramified\ Cells$	46	8	0.14	100	48
$Ramified\ Pairs$	22	2	0.27	100	66
$Node\ and\ LL$	112	20	0.71	45	27
DAG	53	6	2.06	100	42
$DAG\ (Values)$	53	6	1.96	100	56
$Total$	286	42	5.14	89	47.8

The first column in table 6.1 gives the name of the program, second column lists the lines of code. While the third column shows the lines of specifications. On average there is about 13% annotation burden on the user. In the fifth column we show the sharing degree, it is defined as the percentage of specifications that use sharing operators ($\wedge$ and $\curlywedge$) introduced in this chapter. As is clear from our benchmark programs that ability to specify sharing is important to verify these data structures. The last column is the percentage of total entailments generated that have to be ramified to eliminate the $-\circledast$ operator during entailment. Ramifications are essential to verify sharing in these programs. We have validated our approach and found it to be useful for verifying programs using sharing (with ramifications) in data structures.

In terms of the limitations of the current approach, since we provide automated proofs of ramifications using septraction lemmas, we require the user to supply the lemmas along with the pre and post conditions of the method. These lemmas are used to eliminate the $-\circledast$ operator during the proof. In addition, the current technique only works for lightweight shape proofs. In future we hope to extend the mechanism to enable verification of more functional properties by integrating automated proving using lemmas in the verifier with interactive proofs done in Coq. The appendix A provides fully worked out detailed examples of how this can be achieved.

6.5 Comparative Remarks and Summary

Our sharing and aliasing logic is inspired by work of Hobor and Villard [46]. They present the RAMIFY rule of separation logic and show how to mechanically reason with graphs, DAGs and overlaid structures using their ramification library. Our work can be seen as a specific instance where we seek

to automatically do verification of programs with ramifications. The operator of overlapping conjunction ($\circledast$) is used in [46] to specify shared heaps between two predicates. As discussed in Sec 6.2, we also use two other operators ($\wedge$ and $\circledast$) that characterize must and may aliasing scenarios to support concise specification with automated reasoning.

The use of new operators for handling sharing is further motivated by recent discovery of sepish operator by Gardner et al. [36] in the context of verification of JavaScript programs. However, they also present only the logic and do not provide an automated system for reasoning. The concept of ramification was introduced in [60] for verifying event-driven programs. They show how to calculate ramified frames in a domain specific logic with particular semantics. Other formalisms to reason with shared structures include logics for reasoning with graphs [17] and views [49].

The problem of sharing has also been explored in the context of concurrent data structures and objects [31, 112]. Our work is influenced by them but for a sequential setting. Regional logic [5] also uses a notion of set of addresses as footprint of formulas. These regions are used with dynamic frames to enable local reasoning of programs. Memory layouts [37] were used by Gast, to formally specify the structure of individual memory blocks. A grammar of memory layouts enable distinguishing between variable, array, or other data structures. When dealing with shared regions of memory, knowing layout of memory is helpful for reasoning.

Shape analysis for other composite and complex structures has been done through the use of higher-order predicates [6] and abstract modeling of containers [30]. These approaches cannot handle unrestricted sharing and aliasing across containers. In [41] Hawkins et al. describe a high level relational algebra based specification mechanism to specify complex sharing,

which is then used to generate the physical data structure that has sharing. They extend their approach in [42] to generate data representation as well as the code to query the data structure in form of relational queries. In their paper they identify the challenge of specifying invariants on multiple overlapping data structures and mention that existing verification techniques are insufficient to reason about them. Our work is an attempt to provide a specification and verification mechanism for such shared structures.

We have proposed a specification mechanism to express different kinds of sharing and aliasing in data structures. The specifications can capture correctness properties of various kinds of programs using ramifications. Our entailment procedure provides the first such mechanism to deal with ramifications in an automated manner. We present an automated verification system which can be used to reason about sharing in data structures. We have implemented a prototype based on our approach. An initial set of experiments with small but challenging programs have confirmed the usefulness of our method.

Chapter 7

Conclusions

In this thesis, we introduced *certified reasoning* as a mechanism to improve the correctness and expressivity of automated verification. We present a certified decision procedure (chapter 3), a certified program (chapter 4) and a certified proof (chapter 5) that improves an existing program verifier (`HIP`). We have benchmarked out approach on a set of example programs and shown that the overhead from *certified reasoning* is small in practice. Careful construction of certified proof and certified program provides a good balance between scalability and expressivity.

We also present a logic to specify different kinds of sharing and aliasing in data structures. We show that our specifications can capture correctness properties of various kinds of programs using unrestricted sharing. We present an automated verification system which can be used to reason about sharing in data structures. Our method can work with a large class of user defined inductive predicates and user specified lemmas. We implement a prototype `HIPComp` based on our approach. An initial set of experiments with small but challenging programs confirm the usefulness of our method.

7.1 Results

Certified Decision Procedure. In chapter 3, we presented `Omega++`, a sound and complete decision procedure for Presburger arithmetic with infinities (including arbitrary quantifier use). `Omega++` does not sacrifice any of the computational advantage normally gained by restricting to Presburger arithmetic, despite the addition of infinities. `Omega++` is written in the Coq theorem prover [1], allowing us to formally certify it (modulo the correctness of Omega itself, which we utilize as our backend). We extract our performance-tuned Coq implementation into OCaml and package it as a library, which we benchmarked using the `HIP`/`SLEEK` verification toolset [23]. The additional of infinity adds to our specification framework's readability and conciseness. We also apply the notion of quantifier elimination in Presburger arithmetic with infinities to infer pure (non-heap) properties of programs.

Certified Program. In chapter 4, we formalized the traits and mixins hierarchies in Scala as required for checking subtyping. We presented an approach based on entailment in separation logic to verify subtyping. We design a domain specific language (`SLEEK DSL`) which is embedded in Scala and can support verified subtyping with traits and mixins. The `SLEEK DSL` extends the Scala language and allows programmers to insert separation logic entailments in their code. We apply our technique to the Scala standard library and verify subtyping in 67% of mixins. This shows that even though mixins do not enforce subtyping, 67% of usage of mixins is in conformance with behavior subtyping.

Certified Proof. In chapters 5 and 6, we provided a specification mechanism to express different kinds of sharing and aliasing scenarios. We enhanced automated reasoning in separation logic with new operators ($\Yinyang$, $\wedge$ and $\mathbb{A}$). We

showed how to check for non interference for data structures with sharing. This enables the automated verification of programs using overlaid data structures. We certify the correctness of key functions used in compatible sharing by proving them in Coq. We found an error in the paper and pen proof of XPure function as given in [23]. In addition, for interfering data structures we provide a method to do automated ramifications. Automated ramifications support unrestricted sharing in data structures. Our entailment procedure preserves the principle of local reasoning which ensures scalability during modular verification. We have implemented our procedure in a prototype (`HIPComp`) and applied to a small benchmark of programs using data structures with complex sharing.

7.2 Future Work

For future work we are looking at *certified reasoning* for functional correctness. It will allow us to provide more natural predicate definitions and proofs for programs manipulating graphs and DAGs. As an example consider the following definition of a `graph` predicate which includes a mathematical graph `G`. This definition is much more natural and close to the definition of a standard `tree` predicate in separation logic.

$$\texttt{data node}\ \{\ \texttt{int val};\ \texttt{node left};\ \texttt{node right}\ \}$$

$$\texttt{graph}\langle \texttt{root}, \texttt{G}\rangle \equiv \texttt{root}=\texttt{null} \vee \exists\, \texttt{d}, \texttt{l}, \texttt{r} \cdot (\texttt{root} \mapsto \texttt{node}\langle \texttt{d}, \texttt{l}, \texttt{r}\rangle \\ \circledast \texttt{graph}\langle \texttt{l}, \texttt{G}\rangle \circledast \texttt{graph}\langle \texttt{r}, \texttt{G}\rangle \wedge \texttt{lookup}(\texttt{x}, \texttt{d}, \texttt{l}, \texttt{r}, \texttt{G}))$$

This definition allows us to provide concise and natural specifications for programs like `mark` which manipulate such graphs. The pre and post condition

of the following function illustrates that the specification matches user intention naturally.

```
void mark(node x)
  requires  graph⟨x, G⟩
  ensures   graph⟨x, G1⟩ ∧ mark(G, x, G1)
  {node l, r;
  if(x == null) return;
  else {if(x.val == 1) return;
    l = x.left; r = x.right;
    x.val = 1; mark(l); mark(r); }}
```

We seek to use our framework of *certified reasoning* to be able to certify mathematical functions like mark(G, x, G1) in Coq. The automated verification system can use lemmas to relate different predicates while the mathematical reasoning is relegated to the proof assistant. This will allow a clean separation between spatial and mathematical aspect of the proof. This kind of reasoning is essential for paper and pen proof of ramifications.

Another application of our approach is to handle incomplete and complex domains for pure properties like sequences, trees and maps. Reasoning over such domains is typically axiomatized in the program verifier. However the correctness of those axioms are not proven. By certifying the axioms in Coq we can provide end to end guarantees for the proof. Such domains can help in proving full functional correctness using an automated verifier. Appendix A presents two fully worked out examples (fibonacci and abstract lists) that illustrate the use of certified reasoning for proving functional properties.

In addition, we will also like to be able to do lightweight shape analysis of graphs, DAGs etc. without the need of specifications for full correctness.

Our extended logic captures various sharing and aliasing scenarios, we would want to infer shape predicates for programs using such structures. Aliasing and deep sharing are challenging problems for current shape analysis tools. We can capture more precise shape predicates using the various sharing operators ($\curlywedge$, $\wedge$ and $\barwedge$) described in this thesis.

The formulation of logic in this thesis is focused on heaps but we also seek to apply our logic for reasoning about other shared resources. Another interesting future direction is to explore the use and interpretation of $\wedge$, $\curlywedge$ and $\barwedge$ operators in concurrent separation logic.

Appendix

A Certified Reasoning Coq Examples

In chapter 3, we presented a certified decision procedure for Presburger arithmetic with infinity. The following two examples show how Coq can be used to support certified reasoning for arbitrary properties. These include theories that are not decidable (like abstract lists) and functional properties of recursive functions (like fibonacci).

A.1 Fibonacci

In this section, we show how we can use certified reasoning with an entailment prover like SLEEK. We use the example of the fibonacci function that can be defined recursively as follows :

```
fib(n) = if (n<=0) then return 0
    else if(1<=n<=2) then return 1
    else return fib(n-1) + fib(n-2)
```

This definition can be axiomatized in SLEEK using an uninterpreted relation (fib) with the following axioms :

```
relation fib(int n, int f).
axiom n<=0 ==> fib(n,0).
```

```
axiom 1<=n<=2 ==> fib(n,1).
axiom n>0 & fib(n,f1) & fib(n+1,f2) ==> fib(n+2,f1+f2).
```

The following entailments about the fib relation are valid, but SLEEK fails to prove them automatically as it requires induction over the fib relation.

```
//1
checkentail fib(1,n) & fib(2,m) |- n = m.
//2
checkentail fib(n,p) & fib(n+1,m) & n = 1 |- m = p.
```

In order to prove these entailments we generate the Coq module type MFIB from the SLEEK file. The MFIB module type parameterizes the logical operators and specifies the fib relation with axioms (1, 2 and 3). In Coq, we provide an implementation for the MFIB type using the MFIBIMPL module. This module provides a certified implementation of the fib relation in Coq. This enables us to prove entailments 1 and 2 from above in Coq. Thus, completing the end to end proof of the entailment which starts from SLEEK and is certified in Coq.

```
Module Type MFIB.
  Parameter formula : Type.
  Parameter valid : formula → Prop.
  Parameter and : formula → formula → formula.
  Parameter imp : formula → formula → formula.
  Parameter not : formula → formula.
  Parameter leq : Z → Z → formula.
  Parameter fib : Z → Z → formula.
  Axiom axiom_1 : ∀ n, valid (imp (leq n 0) (fib n 0)).
  Axiom axiom_2 : ∀ n, valid (imp (and (leq 1 n) (leq n 2)) (fib n 1)).
```

Axiom *axiom_3* : $\forall$ *n f1 f2*, *valid* (*imp* (*and* (*not* (*leq n* 0)) (*and* (*fib n f1*) (*fib* (*n*+1) *f2*))) (*fib* (*n*+2) (*f1*+*f2*))).

Axiom *entail_1* : $\forall$ *n m*, *valid* (*imp* (*and* (*fib* 1 *n*) (*fib* 2 *m*)) (*and* (*leq n m*) (*leq m n*))).

Axiom *entail_2* : $\forall$ *n m p*, *valid* (*imp* (*and* (*and* (*leq n* 1) (*leq* 1 *n*)) (*and* (*fib n p*) (*fib* (*n*+1) *m*))) (*and* (*leq m p*) (*leq p m*))).

End MFIB.

Module MFIBIMPL <: FIB.MFIB.

Inductive **PF** : Type :=

| F_and : **PF** → **PF** → **PF**

| F_imp : **PF** → **PF** → **PF**

| F_not : **PF** → **PF**

| F_leq : **Z** → **Z** → **PF**

| F_fib : **Z** → **Z** → **PF**.

Definition formula := **PF**.

Definition and := F_and.

Definition imp := F_imp.

Definition not := F_not.

Definition leq := F_leq.

Definition fib := F_fib.

Fixpoint FIB_nat *n* : **nat** :=

match *n* with

$0 \Rightarrow 0$

| (S *p*) ⇒

match *p* with

$0 \Rightarrow 1$

```
    | (S m) ⇒ plus (FIB_nat p) (FIB_nat m)
  end
end.

Definition FIB (n:Z) : Z := Z.of_nat (FIB_nat (Z.to_nat n)).

Lemma plus_1_Sn :
 ∀ n:nat, n + 1 = S n.

Lemma plus_2_SSn :
 ∀ n:nat, n + 2 = S (S n).

Lemma FIB_nat_2 : ∀ n, (FIB_nat (n+2)) = (FIB_nat (n)) + (FIB_nat
(n+1)) .

Fixpoint satis (f:formula) : Prop :=
match f with
  | F_and f1 f2 ⇒ satis f1 ∧ satis f2
  | F_imp f1 f2 ⇒ satis f1 → satis f2
  | F_not f ⇒ ~ (satis f)
  | F_leq n1 n2 ⇒ (Z.le n1 n2)
  | F_fib n f ⇒ f = (FIB n)
end.

Definition valid (f:formula) := satis f.

Lemma axiom_1 : ∀ n, valid (imp (leq n 0) (fib n 0)).

Lemma axiom_2 : ∀ n, valid (imp (and (leq 1 n) (leq n 2)) (fib n 1)).

Lemma axiom_3 : ∀ n f1 f2, valid (imp (and (not (leq n 0)) (and (fib n f1) (fib
(n+1) f2))) (fib (n+2) (f1+f2))).

Lemma entail_1 : ∀ n m, valid (imp (and (fib 1 n) (fib 2 m)) (and (leq n m)
(leq m n))).
```

`Lemma` entail_2 : $\forall$ *n m p*, valid (imp (and (and (leq *n* 1) (leq 1 *n*)) (and (fib *n p*) (fib (*n*+1) *m*))) (and (leq *m p*) (leq *p m*))).

`End` MFIBIMPL.

A.2 LinkedList

In this section, we show how to enable certified reasoning with an automated verified like HIP. We use the example of the abstract lists along with separation logic to illustrate how certified reasoning about theories that are not decidable can be integrated in HIP. Consider the following definition about linked lists in HIP which also uses the abstract list L.

```
ll<L> == self=null & isempty(L)
or self::node<v, p> * p::ll<Lp> & cons(L,v,Lp)
inv (self=null & isempty(L) |
     self!=null & !(isempty(L)));
```

The abstract list L, is itself defined recursively using the following uninterpreted relations and axioms.

```
relation cons(abstract L, int v, abstract Lt).
relation reverse(abstract L, abstract L1).
relation append(abstract L, abstract L1, abstract L2).
relation isempty(abstract L).
axiom cons(L,v,Lp) ==> !(isempty(L)).
axiom isempty(L) ==> append(L1,L,L1).
axiom isempty(L) ==> reverse(L,L).
axiom cons(L,v,Lt) & reverse(Tr,Lt) ==>
exists (Le: exists (Lv: exists (Lr: append(Lr,Tr,Lv)
& reverse(Lr,L) & cons(Lv,v,Le) & isempty(Le)))).
```

In order to certify the correctness of these axioms we automatically generate the Coq module type MLL from the HIP file. The MLL module type parameterizes the operators of separation logic and specifies the axioms about abstract lists. In Coq, we provide an implementation of the MLL type using the MLLIMPL module. We give a certified implementation of separation logic and use the standard list library in Coq to prove the axioms about abstract lists. This completes the end to end proof for reasoning with abstract lists in HIP.

```
Module Type MLL.
  Parameter formula : Type.
  Parameter valid : formula → Prop.
  Parameter node : Type.
  Parameter null_node : node.
  Parameter ptto_node : node → Z → node → formula.
  Parameter A : Type.
  Parameter ll : node → A → formula.
  Parameter star : formula → formula → formula.
  Parameter and : formula → formula → formula.
  Parameter imp : formula → formula → formula.
  Parameter not : formula → formula.
  Parameter eq : node → node → formula.
  Parameter isempty : A → formula.
  Parameter append : A → A → A → formula.
  Parameter reverse : A → A → formula.
  Parameter cons : A → node → A → formula.
  Axiom axiom_1 : ∀ Lt Tr L v,∃ Le Lv Lr, valid (imp (and (cons L v Lt)
(reverse Tr Lt)) (and (and (and (append Lr Tr Lv) (reverse Lr L)) (cons
Lv v Le)) (isempty Le))).
```

```
  Axiom axiom_2 : ∀ L, valid (imp (isempty L) (reverse L L)).
  Axiom axiom_3 : ∀ L L1, valid (imp (isempty L) (append L1 L L1)).
  Axiom axiom_4 : ∀ v Lp L, valid (imp (cons L v Lp) (not (isempty L))).
End MLL.

Module MLLIMPL <: LL.MLL.

Definition A := list nat.
Definition node := nat.
Definition null_node := 0.

Inductive HF : Type :=
  | H_emp : HF
  | H_ptto : nat → Z → nat → HF
  | H_star : HF → HF → HF
  | H_and : HF → HF → HF
  | H_imp : HF → HF → HF
  | H_not : HF → HF
  | H_eq : nat → nat → HF
  | H_cons : A → nat → A → HF
  | H_reverse : A → A → HF
  | H_append : A → A → A → HF
  | H_isempty : A → HF
  | H_ll : nat → A → HF.

Definition formula := HF.
Definition star := H_star.
Definition and := H_and.
Definition imp := H_imp.
Definition not := H_not.
```

```
Definition eq := H_eq.

Definition ptto_node := H_ptto.

Definition cons := H_cons.

Definition reverse := H_reverse.

Definition append := H_append.

Definition isempty := H_isempty.

Definition ll := H_ll.

Definition heap := Ensemble nat.

Definition empty_heap := Empty_set nat.

Definition heap_union h1 h2 := Union nat h1 h2.

Definition heap_is_disjoint h1 h2 := Disjoint nat h1 h2.

Inductive LL (n:nat) (l:A) : heap → Prop :=
  | NIL_LL : LL n l empty_heap
  | CONS_LL : ∀ h h1 h2 n1 n2, h = heap_union h1 h2
                → heap_is_disjoint h1 h2
                → n1 > 0 → n1 = (hd 0 l)
                → LL n2 (tl l) h1 → LL n l h.

Fixpoint satis (f:formula) (h:heap) :Prop :=
match f with
  | H_emp ⇒ h = empty_heap
  | H_ptto n _ _ ⇒ n > 0
  | H_star f1 f2 ⇒ ∃ h1 h2, h = heap_union h1 h2 ∧ heap_is_disjoint h1 h2
∧ satis f1 h1 ∧ satis f2 h2
  | H_and f1 f2 ⇒ satis f1 h ∧ satis f2 h
  | H_imp f1 f2 ⇒ satis f1 h → satis f2 h
  | H_not f ⇒ ~ (satis f h)
```

```
  | H_eq n1 n2 ⇒ n1 = n2
  | H_cons l n l1 ⇒ l = n::l1
  | H_reverse l l1 ⇒ l = (rev l1)
  | H_append l l1 l2 ⇒ l = l1 ++ l2
  | H_isempty l ⇒ l = nil
  | H_ll n l ⇒ LL n l h
end.

Definition valid (f:formula) := ∀ h, satis f h.

Lemma axiom_1 : ∀ Lt Tr L v,∃ Le Lv Lr, valid (imp (and (cons L v Lt)
(reverse Tr Lt)) (and (and (and (append Lr Tr Lv) (reverse Lr L)) (cons Lv
v Le)) (isempty Le))).

Lemma axiom_2 : ∀ L, valid (imp (isempty L) (reverse L L)).

Lemma axiom_3 : ∀ L L1, valid (imp (isempty L) (append L1 L L1)).

Lemma axiom_4 : ∀ v Lp L, valid (imp (cons L v Lp) (not (isempty L))).

End MLLIMPL.
```

B Certified Reasoning for Separation Logic

In chapter 5, we presented an extension to separation logic that allows us to specify and verify compatible sharing in data structures. As part of the compatible sharing extension to separation logic we formalized and certified the proof of correctness of XPure and XMem functions in Coq.

B.1 Reduction to PA (XPure)

The XPure function checks the satisfiability of a formula in separation logic by reducing it to Presburger arithmetic (PA). While proving this procedure in Coq

we identified an error in the previous paper and pen proof given by Chin et. al. in [23].

```
Module Type STRVAR <: VARIABLE.
  Parameter var : Type.   Parameter var_eq_dec : ∀ v1 v2 : var, {v1 = v2} + {v1 ≠ v2}.
  Parameter var2string : var → string.
  Parameter string2var : string → var.
  Parameter freshvar : var.
  Axiom var2String2var : ∀ v, string2var(var2string v) = v.
  Axiom String2var2String : ∀ s, var2string(string2var s) = s.
End STRVAR.

Module HEAPSOLVER(sv:STRVAR).

Module PA := ARITHSEMANTICS PURENAT SV.

Inductive HF : Type :=
  | H_Emp : HF
  | H_Ptto : PA.ZExp → PA.ZExp → HF
  | H_Star : HF → HF → HF
  | H_List : PA.ZExp → HF
  | H_List_Size : PA.ZExp → nat → HF
  | H_Exists : var → HF → HF
  | H_And : HF → PA.ZF → HF
  | H_Pure : PA.ZF → HF.

Definition heap := Ensemble nat.

Definition empty_heap := Empty_set nat.

Definition single_heap e := Singleton nat (PA.dexp2ZE e).
```

```
Definition heap_union h1 h2 := Union nat h1 h2.

Definition heap_is_disjoint h1 h2 := Disjoint nat h1 h2.

Inductive LL (e:PA.ZExp) : heap → Prop :=
  | NIL_LL : LL e empty_heap
  | CONS_LL : ∀ h h1 h2 e1, h = heap_union h1 h2
                → heap_is_disjoint h1 h2
                → h1 = single_heap e
                → LL e1 h2 → LL e h.

Inductive LLSIZE (e:PA.ZExp) (n: nat) : heap → Prop :=
  | NIL_LLSIZE : (PA.dexp2ZE e) = 0 → n = 0 → LLSIZE e n empty_heap
  | CONS_LLSIZE : ∀ h h1 h2 e1 n1, h = heap_union h1 h2
                → heap_is_disjoint h1 h2
                → h1 = single_heap e
                → (PA.dexp2ZE e) > 0
                → n = n1 + 1
                → LLSIZE e1 n1 h2 → LLSIZE e n h.

Theorem LLSIZE_implies_LL: ∀ e h n, LLSIZE e n h → LL e h.

Fixpoint subs (p : var × PureNat.N.A) (form : HF) : HF :=
match form with
  | H_Emp ⇒ form
  | H_Ptto e1 e2 ⇒ H_Ptto (PA.subst_exp p e1) (PA.subst_exp p e2)
  | H_Star f1 f2 ⇒ H_Star (subs p f1) (subs p f2)
  | H_List e ⇒ H_List (PA.subst_exp p e)
  | H_List_Size e n ⇒ H_List_Size (PA.subst_exp p e) n
  | H_Exists v g ⇒ if var_eq_dec (fst p) v then form else H_Exists v
(subs p g)
```

```
  | H_And f g ⇒ H_And (subs p f) (PA.substitute p g)
  | H_Pure g ⇒ H_Pure (PA.substitute p g)
end.

Fixpoint length_hform (form : HF) : nat :=
match form with
  | H_Exists v g ⇒ S (length_hform g)
  | H_Star f1 f2 ⇒ S (length_hform f1 + length_hform f2)
  | H_And f g ⇒ S (length_hform f)
  | _ ⇒ 1
end.

Lemma length_hform_gteq_one: ∀ f, length_hform f ≥ 1.

Fixpoint dvalid_hform' (form: HF) (h:heap) (c:nat): Prop :=
match c with
  0 ⇒ False
| S c' ⇒ match form with
            H_Emp ⇒ h = empty_heap
          | H_Ptto e1 e2 ⇒ h = (single_heap e1) ∧ (PA.dexp2ZE e1) >
O
          | H_Star f1 f2 ⇒ ∃ h1 h2,
                              (dvalid_hform' f1 h1 c') ∧ (dvalid_hform'
f2 h2 c')
                              ∧ (heap_is_disjoint h1 h2) ∧ h =
(heap_union h1 h2)
          | H_List e ⇒ LL e h
          | H_List_Size e n ⇒ LLSIZE e n h
          | H_Exists v g ⇒ ∃ x, dvalid_hform' (subs (v,x) g) h c'
```

```
      | H_And f g ⇒ (dvalid_hform' f h c') ∧ (PA.dvalid_zform g)
      | H_Pure g ⇒ (PA.dvalid_zform g)
    end
end.

Definition dvalid_hform f h := dvalid_hform' f h (length_hform f).

Lemma pure_valid_in_all_heap: ∀ h g, (dvalid_hform (H_Pure g) h) ↔
PA.dvalid_zform g.

Lemma subs_length_inv : ∀ f x v, length_hform f = length_hform (subs
(v, x) f).

Lemma large_c_holds : ∀ f h c1 c2, c1 ≥ length_hform f → c2 ≥
length_hform f →
                                        (dvalid_hform' f h c1 ↔
dvalid_hform' f h c2).

Definition unfold_list_pure (e: PA.ZExp) : PA.ZF :=

(PA.ZF_Or (PA.ZF_BF (PA.ZBF_Eq e (PA.ZExp_Const
PureNat.N.Const0)))
(PA.ZF_BF (PA.ZBF_Gt e (PA.ZExp_Const PureNat.N.Const0))))
.

Definition unfold_list_size_pure (e: PA.ZExp) (n: nat) : PA.ZF :=
(PA.ZF_Or (PA.ZF_And
(PA.ZF_BF (PA.ZBF_Eq e (PA.ZExp_Const PureNat.N.Const0)))
(PA.ZF_BF (PA.ZBF_Eq (PA.ZExp_Const n) (PA.ZExp_Const
PureNat.N.Const0))))
(PA.ZF_And
(PA.ZF_BF (PA.ZBF_Gt e (PA.ZExp_Const PureNat.N.Const0)))
```

(PA.ZF_BF (PA.ZBF_Gt (PA.ZExp_Const n)(PA.ZExp_Const PureNat.N.Const0)))))

.

```
Fixpoint xpure' (form: HF) : PA.ZF :=

          match form with
            | H_Emp ⇒ PA.ZF_BF (PA.ZBF_Const true)
            | H_Ptto e1 e2 ⇒ PA.ZF_BF (PA.ZBF_Gt e1 (PA.ZExp_Const
PureNat.N.Const0))
            | H_Star f1 f2 ⇒ PA.ZF_And (xpure' f1) (xpure' f2)
            | H_List e ⇒ (unfold_list_pure e)
            | H_List_Size e n ⇒ (unfold_list_size_pure e n)
            | H_Exists v g ⇒ PA.ZF_Exists v tt (xpure' g)
            | H_And f g ⇒ PA.ZF_And (xpure' f) g
            | H_Pure g ⇒ g
          end
```

.

```
Definition xpure f := H_Pure (xpure' f).

Lemma PA_dexp2ZE_always_positive: ∀ e, (PA.dexp2ZE e) ≥ 0.

Lemma xpure_length_gt : ∀ f, (length_hform f) ≥ length_hform (xpure f).

Lemma xpure_length_one : ∀ f, (length_hform (xpure f)) = 1.

Lemma substitute_xpure'_eq_xpure'_subs : ∀ v x f,
                                        PA.substitute (v, x) (xpure' f) =
                                        xpure' (subs (v, x) f).

Theorem xpure_valid: ∀ f h,
(dvalid_hform f h) → dvalid_hform (xpure f) h.
```

`Definition` entail $P\ Q$:= $\forall\ h$, (dvalid_hform $P\ h$) $\rightarrow$ (dvalid_hform $Q\ h$).

`End` HEAPSOLVER.

B.2 Compatible Sharing (XMem)

The XMem function is used to check compatible sharing in data structures, it reduces a separation logic formula to a constraint on sets of addresses. The validity of the resulting set formula can be checked using monadic second order logic (MONA).

`Module Type` STRVAR <: VARIABLE.

`Parameter` *var* : `Type`. `Parameter` *var_eq_dec* : $\forall$ *v1 v2* : *var*, $\{v1 = v2\} + \{v1 \neq v2\}$.

`Parameter` *var2string* : *var* $\rightarrow$ **string**.

`Parameter` *string2var* : **string** $\rightarrow$ *var*.

`Parameter` *freshvar* : *var*.

`Axiom` *var2String2var* : $\forall\ v$, *string2var*(*var2string* v) = v.

`Axiom` *String2var2String* : $\forall\ s$, *var2string*(*string2var* s) = s.

`End` STRVAR.

`Module` HEAPSOLVER(*sv*:STRVAR).

`Definition` heap := Ensemble **nat**.

`Definition` empty_heap := **Empty_set nat**.

`Definition` single_heap n := **Singleton nat** n.

`Definition` heap_union $h1\ h2$:= **Union nat** $h1\ h2$.

`Definition` heap_is_disjoint $h1\ h2$:= **Disjoint nat** $h1\ h2$.

`Inductive` **SE** : `Type` :=

| H_Set_Union : **SE** $\rightarrow$ **SE** $\rightarrow$ **SE**

```
  | H_Set : heap → SE
  | H_Set_Single : nat → SE
  | H_Set_Emp : SE.

Inductive SF : Type :=
  | H_Set_Eq : SE → SE → SF
  | H_Set_Disj : SE → SE → SF
  | H_Set_And : SF → SF → SF.

Inductive HF : Type :=
  | H_Emp : HF
  | H_Ptto : nat → nat → HF
  | H_Star : HF → HF → HF
  | H_List : nat → heap → HF.

Fixpoint SE2Set (s:SE) : heap :=
match s with
 | H_Set_Emp ⇒ empty_heap
 | H_Set h ⇒ h
 | H_Set_Single n ⇒ single_heap n
 | H_Set_Union s1 s2 ⇒ heap_union (SE2Set s1) (SE2Set s2)
end.

Inductive LL (e:nat) : heap → Prop :=
  | NIL_LL : LL e empty_heap
  | CONS_LL : ∀ h h1 h2 e1, h = heap_union h1 h2
                → heap_is_disjoint h1 h2
                → h1 = single_heap e
                → LL e1 h2 → LL e h.

Inductive LLSET (n:nat) (S: heap): Prop :=
```

```
  | NIL_LLSET : S = empty_heap → LLSET n S
  | CONS_LLSET : ∀ S1 S2 n1, S = heap_union S1 S2
                 → heap_is_disjoint S1 S2
                 → S1 = single_heap n
                 → LLSET n1 S2 → LLSET n S.

Theorem LL_is_same_as_LLSET: ∀ h e, LLSET e h ↔ LL e h.

Fixpoint valid (form: HF) (h:heap) : Prop :=
match form with
    H_Emp ⇒ h = empty_heap
  | H_Ptto n1 _ ⇒ h = (single_heap n1)
  | H_Star f1 f2 ⇒ ∃ h1 h2, (valid f1 h1) ∧ (valid f2 h2)
                    ∧ (heap_is_disjoint h1 h2) ∧ h = (heap_union h1
h2)
  | H_List n h ⇒ LLSET n h
end.

Fixpoint set_valid (form: SF) : Prop :=
match form with
    H_Set_Eq s1 s2 ⇒ (SE2Set s1) = (SE2Set s2)
  | H_Set_And f1 f2 ⇒ set_valid f1 ∧ set_valid f2
  | H_Set_Disj s1 s2 ⇒ heap_is_disjoint (SE2Set s1) (SE2Set s2)
end.

Fixpoint XMem (f: HF) : SE :=
match f with
  | H_Emp ⇒ H_Set_Emp
  | H_Ptto n1 _ ⇒ (H_Set_Single n1)
  | H_List n h ⇒ H_Set h
```

```
  | H_Star f1 f2 ⇒ H_Set_Union (XMem f1) (XMem f2)
end.

Fixpoint XMem_Form (f: HF) x : Prop :=
match f with
 | H_Star f1 f2 ⇒
          ∃ x1 x2 x3 x4, (set_valid
          (H_Set_And
          (H_Set_Disj (H_Set x1) (H_Set x2))
          (H_Set_Eq (H_Set x) (H_Set_Union (H_Set x1) (H_Set x2)))))
          ∧ (XMem_Form f1 x3) ∧ (XMem_Form f2 x4)
 | _ ⇒ set_valid (H_Set_Eq (H_Set x) (XMem f))
end.

Theorem valid_xmem_form: ∀ f, (∃ h, (valid f h) )
→ (∃ x, (XMem_Form f x) ).

End HEAPSOLVER.
```

Glossary

Automated Verification	An alternative to testing wherein a formal (mathematical) model of a system is built and analyzed, algorithmically, with respect to logical specifications, 2
Certified Reasoning	Use of proof assistants like Coq to specify and verify reasoning algorithms, 8
Code Synthesis	A form of automatic programming where the goal is to construct automatically a program that provably satisfies a given high-level specification, 13
Compatible Sharing	A form of heap sharing where the operations defined on the data structure do not interfere, 8
DAG	Directed Acyclic Graph, 7
DSL	Domain-Specific Language, 10
Functional Correctness	It refers to the input-output behaviour of the algorithm (i.e., for each input it produces the correct output), 2

JML	Java Modeling Language, 17
Local Reasoning	A form of reasoning where specifications and proofs concentrate on the portion of memory used by a program component, and not the entire global state of the system, 3
OO	Object Oriented, 66
PA	Presburger Arithmetic, 34
PAI	Presburger Arithmetic with Infinity, 48
PAInf	Presburger Arithmetic with positive and negative infinities, 34
PAL	Pointer Assertion Logic, 20
QE	Quantifier Elimination, 39
QFBAPA	Quantifier Free Boolean Algebra with Presburger Arithmetic, 62
Ramification	The ramification problem is concerned with the indirect consequences of an action or how to represent what happens implicitly due to an action, 8
SAT	Satisfiability, 49

Septraction	The existential magic wand ($-\circledast$) operator from separation logic, 114
Septraction Lemma	A lemma relating two formulas in separation logic that makes use of the septraction ($-\circledast$) operator, 118
SMT	Satisfiability Modulo Theories, 76
TVLA	Three Valued Logic Analysis Engine, 20

Bibliography

[1] The Coq Proof Assistant. `http://coq.inria.fr/`. 12, 32, 63, 132

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Standard 754-2008*, pages 1–70, Aug 2008. 44

[3] Andrew W Appel. Verismall: Verified smallfoot shape analysis. In *Certified Programs and Proofs*, pages 231–246. Springer, 2011. 22

[4] Michael Backes, Cătălin Hriţcu, and Thorsten Tarrach. Automatically verifying typing constraints for a data processing language. In *Certified Programs and Proofs*, pages 296–313. Springer, 2011. 76

[5] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming*, pages 387–411, 2008. 26, 110, 129

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *International Conference on Computer-Aided Verification*, pages 178–192, 2007. 28, 129

[7] J. Berdine, C. Calcagno, and P. W. O'Hearn. A Decidable Fragment of Separation Logic. In *24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 97–109. Springer-Verlag, December 2004. 19, 62

[8] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, volume 3780, pages 52–68. Springer-Verlag, November 2005. 19

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposia on Formal Methods for Components and Objects*, Springer LNCS 4111, pages 115–137, 2006. 2, 3, 19

[10] Merrie Bergmann. *An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems*. Cambridge University Press, 2008. 44

[11] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. 62

[12] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an smt solver. In *The ACM SIGPLAN International Conference on Functional Programming*, pages 105–116, 2010. 76

[13] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transactions on Computational Logic*, 6(3):614–633, July 2005. 62

[14] John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6):22:1–22:33, August 2010. 28, 29, 84

[15] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005. 17

[16] Cristiano Calcagno, Hongseok Yang, and PeterW. OHearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2001. 62

[17] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. Springer Berlin Heidelberg, 2002. 25, 129

[18] Patrice Chalin. Improving jml: For a safer and more effective language. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 440–461. Springer Berlin Heidelberg, 2003. 17

[19] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208. ACM, 2013. 22

[20] Renato Cherini, Lucas Rearte, and Javier Blanco. A shape analysis for non-linear data structures. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 201–217, Berlin, Heidelberg, 2010. Springer-Verlag. 28, 110

[21] Wei-Ngan Chin, Cristina David, and Cristian Gherghina. A hip and sleek verification system. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 9–10, New York, NY, USA, 2011. ACM. 19, 70, 73

[22] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 87–99, New York, NY, USA, 2008. ACM. 20, 66, 69, 76

[23] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006 – 1036, 2012. 2, 3, 12, 19, 33, 34, 58, 91, 96, 99, 102, 107, 125, 132, 133, 146

[24] Wei-Ngan Chin, Cristian Gherghina, Rzvan Voicu, QuangLoc Le, Florin Craciun, and Shengchao Qin. A specialization calculus for pruning disjunctive predicates to support verification. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2011. 19

[25] Andreea Costea, Asankhaya Sharma, and Cristina David. Hipimm: verifying granular immutability guarantees. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, pages 189–194, 2014. 19

[26] Ferruccio Damiani, Johan Dovland, EinarBroch Johnsen, and Ina Schaefer. Verifying traits: an incremental proof system for fine-grained reuse. volume 26, pages 761–793. Springer London, 2014. 76

[27] Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 359–374, New York, NY, USA, 2011. ACM. 84, 94, 96

[28] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, May 2008. 62

[29] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970. 1

[30] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 187–200, New York, NY, USA, 2011. ACM. 28, 129

[31] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, MatthewJ. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-Oriented Programming*, volume 6183, pages 504–528. Springer Berlin Heidelberg, 2010. 25, 109, 129

[32] Dino Distefano and Matthew J. Parkinson J. jstar: Towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM. 20, 66

[33] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915 – 941, 2011. 77

[34] Cezara Drgoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 150–171. Springer Berlin Heidelberg, 2013. 110

[35] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006. 65

[36] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, New York, NY, USA, 2012. ACM. 24, 129

[37] Holger Gast. Reasoning about memory layouts. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 628–643. Springer Berlin Heidelberg, 2009. 27, 110, 129

[38] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 386–401. Springer Berlin Heidelberg, 2011. 2

[39] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer Berlin Heidelberg, 2006. 2, 62

[40] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 310–323, New York, NY, USA, 2005. ACM. 20

[41] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data structure fusion. In *Programming Languages and Systems*,

volume 6461 of *Lecture Notes in Computer Science*, pages 204–221. Springer Berlin Heidelberg, 2010. 29, 30, 129

[42] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 38–49, New York, NY, USA, 2011. ACM. 30, 130

[43] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 417–428, New York, NY, USA, 2012. ACM. 30

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. 2

[45] Tony Hoare and Jay Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2008. 3

[46] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 523–536, New York, NY, USA, 2013. ACM. 7, 23, 114, 116, 117, 128, 129

[47] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM. 2, 34, 97

[48] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag. 18, 29

[49] Jonas Braband Jensen, Lars Birkedal, and Peter Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 10:2: 1–20, 2011. 25, 129

[50] C. Jones, P. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, April 2006. 3

[51] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. 62

[52] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, number 05431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. 39

[53] Deepak Kapur, Zhihai Zhang, Matthias Horbach, Hengjun Zhao, Qi Lu, and ThanhVu Nguyen. Automated reasoning and mathematics. chapter Geometric Quantifier Elimination Heuristics for Automatically Generating Octagonal and Max-plus Invariants, pages 189–228. Springer-Verlag, Berlin, Heidelberg, 2013. 39

[54] Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997. 62

[55] P. Kelly, V. Maslov, W. Pugh, and et al. *The Omega Library Version 1.1.0 Interface Guide*, November 1996. 32, 47, 63, 108

[56] N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001. 108

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 22

[58] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: integrating smt and programming. In *International Conference on Automated Deduction*, pages 400–406, 2011. 76

[59] Nikolaevich Andrey Kolmogorov. *"Infinity." Encyclopaedia of Mathematics: An Updated and Annotated Translation of the Soviet "Mathematical Encyclopaedia,"*, volume 3. Reidel, 1995. 33

[60] Neel Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *ACM SIGPLAN Workshop on. Types in Language Design and Implementation*, pages 63–76, 2010. 25, 129

[61] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 17–32, New York, NY, USA, 2002. ACM. 20

[62] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. An algorithm for deciding bapa: Boolean algebra with presburger arithmetic. In *20th International Conference on Automated Deduction (CADE'05)*, pages 260–277, Tallinn, Estonia, Jul 2005. 62

[63] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *International*

Conference on Automated Deduction, volume 4603 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin Heidelberg, 2007. 62

[64] Aless Lasaruk and Thomas Sturm. Effective quantifier elimination for presburger arithmetic with infinity. In *Computer Algebra in Scientific Computing*, volume 5743 of *Lecture Notes in Computer Science*, pages 195–212. Springer Berlin Heidelberg, 2009. 43, 48, 59, 60, 62

[65] Quang Loc Le, Asankhaya Sharma, Florin Craciun, and Wei-Ngan Chin. Towards complete specifications with an error calculus. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 291–306, 2013. 19

[66] TonChanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A resource-based logic for termination and non-termination proofs. In *Formal Methods and Software Engineering*, volume 8829 of *Lecture Notes in Computer Science*, pages 267–283. Springer International Publishing, 2014. 38

[67] Xuan Bach Le, Cristian Gherghina, and Aquinas Hobor. Decision procedures over sophisticated fractional permissions. In *Asian Symposium on Programming Languages and Systems*, pages 368–385, 2012. 84, 91

[68] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *International Conference on Computer-Aided Verification*, pages 592–608, 2011. 7, 27, 81

[69] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design*, pages 195–222, 2009. 18, 29

[70] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record*

of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM. 22

[71] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. 68

[72] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. 68

[73] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 36(5):450–462, 1993. 63

[74] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 293–304, New York, NY, USA, 2013. ACM. 61

[75] Z. Manna and C. G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003. 62

[76] M. Marcus and A. Pnueli. Using ghost variables to prove refinement. In *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 226–240. Springer Berlin Heidelberg, 1996. 61

[77] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Proceedings of the 17th International*

Conference on Computer Aided Verification, CAV'05, pages 476–490, Berlin, Heidelberg, 2005. Springer-Verlag. 61, 62

[78] Edward James McShane. *Unified integration*, volume 107. Academic Press, 1983. 33

[79] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 221–231, New York, NY, USA, 2001. ACM. 20

[80] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 243–257. Springer Berlin Heidelberg, 2008. 62

[81] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 557–570, New York, NY, USA, 2012. ACM. 29

[82] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, October 1979. 62

[83] HuuHai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 355–369. Springer Berlin Heidelberg, 2008. 19, 118, 120, 125

[84] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 62, 63

[85] Tobias Nipkow. Linear quantifier elimination. In *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin Heidelberg, 2008. 62

[86] Michael Norrish. Complete integer decision procedures as derived rules in hol. In *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 71–86. Springer Berlin Heidelberg, 2003. 62

[87] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2006. 69

[88] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag. 3

[89] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM. 20

[90] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM. 20, 66

[91] Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *Proceedings of the 9th International*

Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'08, pages 218–232. Springer-Verlag, Berlin, Heidelberg, 2008. 62

[92] Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 268–280. Springer Berlin Heidelberg, 2008. 62

[93] Mojzesz Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt*. 1929. 11, 32, 62

[94] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 231–242, New York, NY, USA, 2013. ACM. 62

[95] John C. Reynolds. *The craft of programming*. Prentice Hall International series in computer science. Prentice Hall, 1981. 61, 62

[96] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. 2, 11, 32, 34, 96, 97

[97] K. Rustan and M. Leino. Developing verified programs with dafny. In *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 82–82. Springer Berlin Heidelberg, 2012. 17

[98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM. 20

[99] Asankhaya Sharma. Towards a verified cardiac pacemaker. Technical report, NUS, 2010. available at http://www.comp.nus.edu.sg/~asankhs/pdf/Towards_Verified_Cardiac_Pacemaker.pdf. 21

[100] Asankhaya Sharma. Building extensible parsers with camlp4. Technical report, NUS, 2011. available at http://www.comp.nus.edu.sg/~asankhs/pdf/BuildingExtensibleParserswithCamlp4.pdf. 19

[101] Asankhaya Sharma. A critical review of dynamic taint analysis and forward symbolic execution. Technical report, NUS, 2012. available at http://www.comp.nus.edu.sg/~asankhs/pdf/ACriticalReviewofDynamicTaintAnalysisandForwardSymbolicExecution.pdf. 15

[102] Asankhaya Sharma. An empirical study of path feasibility queries. *CoRR*, abs/1302.4798, 2013. 15

[103] Asankhaya Sharma. End to end verification and validation with SPIN. *CoRR*, abs/1302.4796, 2013. 21

[104] Asankhaya Sharma. A refinement calculus for promela. In *2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013*, pages 75–84, 2013. 21

[105] Asankhaya Sharma. Exploiting undefined behaviors for efficient symbolic execution. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 727–729, 2014. 15

[106] Asankhaya Sharma. Verified subtyping with traits and mixins. In *Proceedings 2nd French Singaporean Workshop on Formal Methods and*

Applications, FSFMA 2014, Singapore, 13th May 2014., pages 45–51, 2014. 12

[107] Asankhaya Sharma, Aquinas Hobor, and Wei-Ngan Chin. Specifying compatible sharing in data structures. Technical report, NUS, 2013. available at http://loris-7.ddns.comp.nus.edu.sg/~project/HIPComp/HIPComp.pdf. 12

[108] Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, and Wei-Ngan Chin. Certified reasoning with infinity. Technical report, NUS, 2014. available at http://loris-7.ddns.comp.nus.edu.sg/~project/SLPAInf/SLPAInf.pdf. 11, 58

[109] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984. 62

[110] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 3–14, New York, NY, USA, 2012. ACM. 22

[111] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: towards flexible verification under fairness. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 709–714, 2009. 21

[112] Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 247–258, New York, NY, USA, 2011. ACM. 25, 109, 129

[113] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 Concurrency Theory*, volume

4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2007. 114

[114] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1/2):3–27, 1988. 63

[115] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997. 63

[116] Volker Weispfenning. Mixed real-integer linear quantifier elimination. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, ISSAC '99, Vancouver, B.C., Canada, July 29-31, 1999*, pages 129–136, 1999. 63