Program Analysis in A Combined Abstract Domain

Guanhua HE

A Thesis submitted to the University of Durham
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Engineering and Computing Sciences
University of Durham
England
2011

Dedicated to

My Supervisor - Prof. Shengchao QIN

for his guidance and help

My Wife

for her love and encouragement

and

My Parents

for their care and support

Abstract

Automated verification of heap-manipulating programs is a challenging task due to the complexity of aliasing and mutability of data structures used in these programs. The properties of a number of important data structures do not only relate to one domain, but to combined multiple domains, such as sorted list, priority queues, height-balanced trees and so on. The safety and sometimes efficiency of programs do rely on the properties of those data structures. This thesis focuses on developing a verification system for both functional correctness and memory safety of such programs which involve heap-based data structures.

Two automated inference mechanisms are presented for heap-manipulating programs in this thesis. Firstly, an abstract interpretation based approach is proposed to synthesise program invariants in a combined pure and shape domain. Newly designed abstraction, join and widening operators have been defined for the combined domain. Furthermore, a compositional analysis approach is described to discover both pre-/post-conditions of programs with a bi-abduction technique in the combined domain.

As results of my thesis, both inference approaches have been implemented and the obtained results validate the feasibility and precision of proposed approaches. The outcomes of the thesis confirm that it is possible and practical to analyse heap-manipulating programs automatically and precisely by using abstract interpretation in a sophisticated combined domain.

Declaration

The work in this thesis is based on research carried out at School of Engineering and Computing Sciences, Durham University, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

First and foremost, I must thank my supervisor Prof. Shengchao Qin for his invaluable academic guidance over the past few years, for his financial support, for his patience to teach me to prepare papers and talks, and also for his help and care for my personal life. His constructive criticism and comments are highly appreciated. Without his advice, support and encouragement, this dissertation would be impossible. He is truly a great mentor.

I also want to acknowledge Prof Wei-Ngan Chin for advising me many bright ideas to overcome the academic problems, and also guiding me in programming in details. I greatly appreciate that he supported me to visit his research lab in National University of Singapore for six months. I have learned many things from his group and had an enjoyable time there.

I need to thank my main colleague and office-mate Chenguang Luo. Discussing with him in academic research greatly inspired me and benefited my research. It was fun and enjoyable to work with him in one room.

Another research-mate whom I am grateful to is Dr. Florin Craciun. His earnest attitude and rich experience on program analysis promoted my research and programming skills.

I wish to thank Dr. Hongseok Yang for sharing his knowledge and research experience about separation logic and abstract interpretation with me. His wisdom and experience in program analysis did not only benefit my

theoretical research, but also helped me in program experiments that led me to begin to explore the data structures used in Linux kernel and drivers in a better way.

I am indebted to Prof. Malcolm Munro for his help and advisement in Durham. He gave me an interview in my first time in Durham University. I also would like to express my gratitude to all my research group-mates and college-mates in Durham, Yonghong Xiang, Qingzheng Zheng, Wei Xiong, Ryuta Arisaka and Granville Barnett, and research group-mates in Singapore, Cristina David and Cristian Gherghina, and so on. They greatly enriched my research and life experience during my Ph.D. study.

I acknowledge EPSRC for its funding support.

Finally, many many thanks should be given to my families, my wife and my parents. They have made a lot of sacrifices to support my study. I have to thank them for both of their emotional and spiritual support. Without their support, love and encouragement, I cannot carry on and commit to academic researches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since electronic computers were developed in the mid-20th century (1940-1945), electronic computing systems have increasingly become more powerful and more flexible, and played more essential roles in almost every area of human's life than anytime before. Modern human beings heavily rely on various electrical computing apparatus, such as household electric appliances like washing machines, microwave ovens, dryers and dishwashers, in mobile devices like mobile telecommunication devices, navigators and PDAs, as well as in safety critical systems such as automated teller machine, financial trading systems, military weapons, missiles, automotive, trains, e-health devices, aircraft and avionics equipments, and so on. These systems greatly facilitate human's life, improve efficiency of human's works, and extend human's capabilities.

However, computer is a double-edged sword. The greater power of computer also brings the greater risk. A computer system may have unpredictable behaviours because of software bugs, poor design, or other factors. Any failure of a safety critical system could lead to serious damage results (Basse, 2009). Some incidents cost billions of pounds. One recent example was a stock market crash in the United States, the "flash crash" on May 6th, 2010, which caused the second largest point swing (1,010.14 points) and the biggest one-day point decline (998.5 points) in the history of the Dow Jones Industrial Average. For a few minutes, one trillion US dollars in market value vanished. This crash was led by an unknown error and accelerated by inappropriately designed computerized automated trading system (CFTC and SEC, 2010). Another example was the Ariane 5 explosion accident which was caused by an overflow error where the conversion from 64 bits to 16 bits was not protected, resulting in a loss of more than 370 million US dollars (Dowson, 1997). Furthermore, software failure could be fatal to human lives. One classic case of a deadly software failure is a radiation therapy medical device, the Therac-25, which involved in at least six serious injuries with three deaths (Leveson and Turner, 1993). Another extreme example was the Patriot missile system failure which resulted in 28 deaths. The missile failure was ultimately attributable to poor handling of rounding errors (Skeel, 1992; Blair et al., 1992). Moreover, a wide collection of computer mishaps that address problems in reliability, safety, security, and privacy issues in day-to-day computer activities could be found in Communications of ACM's "Inside Risk" columns.

Because of the serious risk of safety critical system failure, a systematic method is needed to ensure that the developed software is error-free, safe,

robust and fulfilled by the requirement of users. Due to the growing of computer software in both size and complexity, checking by hand is almost impossible. Besides that, software testing is very limited in its ability to discover all latent defects to establish confidence that the software is fit for its intended use (Dijkstra, 1972). Hoare and Milner (2004) suggest developing a strong software engineering toolset to assist the formal verification of computer software system. It has been identified as an international grand challenge to verify software by formal methods as automatically as possible (Jones et al., 2006; Woodcock, 2006).

1.1 Background

To ensure the quality of software, a number of approaches have been applied to assist in design, development, analysis, testing and formal verification of computer programs, such as formal specification language design, program language design, software testing, model checking, program analysis, program verification, and so on.

To formally describe the requirement and behaviour of computer software, formal specification languages are designed and used during the requirement analysis, systems design and systems analysis, such as UML (Jacobson et al., 1999), CSP (Hoare, 1978), CASL (Bidoit and Mosses, 2004), Z (Spivey, 1989), and so on. Specification languages offer a standard way to specify and construct the requirement and behaviour of programs. An important

application of specification languages is for the proofs of program correctness, likely used in theorem proving and model checking. The specification languages are generally at a much higher level than a programming language, and normally not directly executable.

To avoid certain kinds of programming errors, a set of specifically designed languages was emerged. A live example is the design of Java language (Venners, 1999) which eliminates the needs of explicit memory allocation and deallocation to try to prevent memory leaking by automatic garbage collection. An extension of Java language, Safety Critical Java (SCJ) (Henties et al., 2009), requires explicit memory usage of each mission to ensure the memory safety and avoid the delay that is caused by garbage collection. Another example is the design of Haskell (Jones, 2003) and OCaml (Leroy et al., 2010) which eliminates casting of types to be able to infer the types of programs automatically and prevents type errors in running-time. Moreover, Ada (Rogers, 1984) programming language was designed to meet the requirement of the United States Department of Defense and targeted at safety critical embedded and real-time systems. On the other hand, the design of these languages cannot guarantee the fully correctness of programs, and they may come with other problems or introduce extra complications, for example, garbage collector threads of Java may delay other threads to make it unsuitable for real-time systems (Petit-Bianco, 1998). The constraints of Ada on the language implementation limit the definition of priority and the task scheduling algorithm to preclude the use of selected algorithm for scheduling jobs with hard deadlines (Cornhilll et al., 1987).

1.1. Background

Software testing is a widely applied approach to improve the reliability and quality of software. By executing programs with certain sets of input values and determining whether the programs terminate successfully and whether the output results meet the required results, software testing is able to discover program bugs, and estimate quality and reliability. An abundance of testing techniques are exploited to generate test cases automatically to improve the applicability and scalability of this method (Bezier, 1990; Hetzel and Hetzel, 1991). However, unlike physical processes, software is bizarre. Due to the complexity of some software, complete test is infeasible (Dijkstra, 1972). One cannot prove the absence of bugs only by testing. Any one of the undiscovered bugs may be the Achilles' heel of the software and may cause catastrophic accidents.

To overcome the weakness of dynamic testing for safety critical applications, formal verifications which use mathematically based approaches to statically analysing and verifying software systems are required. Model checking (Clarke and Emerson, 1981; Queille and Sifakis, 1982) is one of the formal methods which was originally applied to automatically verify the correctness of finite-state systems by an exhaustive exploration of the space of computation states according to a specification in temporal logic. To check software, model checking was equipped with abstraction and specific techniques to summarise potentially infinite sets of computation states to finite states. A well known example is the SLAM project (Ball and Rajamani, 2002) which was successfully applied to Microsoft's Static Driver Verifier (SDV) (Ball et al., 2004). Another successful model checker applied to software verification is BLAST (Henzinger et al., 2003) which employs counterexample-guided automatic abstraction refinement to construct abstract models for C

programs.

Other formal verification methods are deductive verification and static program analysis. Deductive verification was pioneered by Floyd (1967) and Hoare (1969) to use logical formulas to describe program behaviour and interpret program statements as predicate transformers to reason about programs with axioms and inference rules. The formal verification system is named Hoare logic, and the central feature of Hoare logic is the Hoare triple. This triple describes the requirement and effect of program codes. A major task of deductive verification is to supply the specifications and loop invariants. A loop invariant is a condition that is necessarily true before and after each iteration of a loop. It characterizes the behaviour of a loop and plays an important role to verify programs statically. Both program specifications and loop invariants could be complex and laborious to specify.

Static program analysis is emerged to solve the specification problem. Static program analysis is a systematical method which executes program code symbolically to derive properties of program and verify the absence of errors at compile-time. The core algorithm of the analysis is to simulate all the program behaviours arising dynamically at run-time when executing a program on a real computer. It intends to conceptually derive all possible set of states the program may reach and to predict the safety of the reachable states before deploying the program code on its running architectures. Since the states of a program may be infinite, program analysis often need to choose reasonable and computable approximations to the program states (Nielson et al., 1999). This thesis mainly focuses on using static program analysis

techniques to infer loop invariants and program specification in a combined domain. Next section discusses the motivation of this research.

1.2 Motivation

As mentioned previously, the history of program verification and analysis could be backtracked to 1960s. Due to the growing complexity of programs, it is still a challenging task to analyse programs written in mainstream imperative heap-manipulating programming languages, such as C, Java and so on. The problem is due to the wide use of recursive shared mutable data structures which are dynamically allocated in memory. "Shared" means one data structure could be pointed/referred to by multiple pointers/references, i.e. these pointers/references are aliased, and "Mutable" means one data structure could be altered after it is created by any access path, which both make it hard to keep track of the properties of data structures statically in a precise and concise way.

Since the emergence of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002), a number of tools have been developed to analyse and verify the correctness of heap-manipulating programs. As an extension of Hoare logic, it presents a framework to reason about these programs by modelling the program memory (both stack and heap) in a natural and accurate manner. In the pure spatial domain (or called shape domain), Berdine et al. (2005a) introduce the Smallfoot tool for automatic verification of separation logic

specifications of programs that manipulate dynamically allocated recursive data structures. As a successor of Smallfoot, the SpaceInvader tool (Distefano et al., 2006; Yang et al., 2008; Calcagno et al., 2009) improves the automatization and infers loop invariants and method specifications on pointer safety for list-manipulating programs. Other works such as SLAyer (Berdine et al., 2007) and jStar (Distefano and Parkinson, 2008) are also focused on the shape domain.

In the shape domain, data structure shapes portray the spatial relationship amongst components of the data structures and their aliasing. However, the properties of a large number of important recursive data structures, such as sorted lists, priority queues, height-balanced trees, and so on, do not only relate to shape domain, but to a combination of shape and pure domains. The pure domain captures the numerical properties such as length/height of data structures, the values stored in the data structures and relationship amongst them. The THOR tool (Magill et al., 2007, 2008, 2010) proposes an analysis mechanism which incorporates with relatively simple numerical information to capture properties like length of list. To capture more general properties, one state-of-the-art verification system Hip/Sleek (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010) has been built over a combined shape and pure domain to verify richer properties like linked lists and trees with relevant quantitative information such as list length and sortedness, tree height and balanced property, and so on.

Compared with previous works, one advantage of Hip/Sleek is the expressiveness of its specification language to allow users to specify their preferred

level of program correctness by defining predicates to depict the data structures used in their programs. Users can define different predicates to describe the spatial relationship amongst components of their data structures and relevant quantitative features that they care about. For example, users may define a linked list predicate with only length information if they only care about the length of lists, or a linked list with sortedness restriction over the content of the list elements if they care about the sortedness of lists. Users may also define the minimal or maximal boundary of the values of list elements if they care about the interval of list elements. Using Hip/Sleek to reason about the program with these predicates allows users to control their preferred level and aspect of verification of both memory safety and functional correctness of heap-manipulating programs.

Besides its power and benefit, Hip/Sleek also has its own limitations. One restriction of Hip/Sleek is that it requires user specified loop invariants. Loop invariants are key conditions of verification of loops, and can be notoriously complex to provide for sophisticated analysis domain and can significantly restrict the scalability of the verification system. It will be good if one can discover the loop invariants automatically for verification of loops in the combined shape and pure domain.

Another motivation of my thesis is to infer the full specifications for any given program code. Previous analysis researches over the combined domain often require preconditions to be given, and Hip/Sleek also requires post-conditions. Providing such information by hand is also cumbersome and error-prone and restricts the scalability of the systems. Therefore, it is very

useful if one can infer the full specification for given program in the combined domain.

1.3 Objectives

The main objective of this thesis is to increase the level of automation of program verification and free programmers' labour, i.e. to reduce the need of user annotations and increase the scalability of a verification system over a combined shape and pure abstract domains, based on existing verification system Hip/Sleek. The investigated aspects of programs are the functional correctness and memory safety of heap-manipulating programs. The reduced annotations are loop invariants and full program specifications. More specifically, this thesis work has the following two aims:

- **Loop Invariant Synthesis** The first goal is to automatically infer the loop invariants over the combined shape and pure domain. The inferred loop invariants should make use of the user-specified recursive data structures to characterize the loops and assist the verification of the loops to be successful.
- **Full Specification Discovery** The second objective is to infer the pre-/post-condition of program methods. The generated specification should be precise and generic enough for further usage, and satisfy the requirements for users. For example, by analysing an insertion sort

method, if the user is only interested in length properties, sortedness properties can be ignored; if the user is interested in sortedness properties, the length information can also be ignored. The user's potential requirements are guided by the definition of data structure predicates.

The inference mechanism is designed for generic propose. Different kinds of data structure and different user-defined predicates could be handled by the same framework. The meaning of generic contrasts with specialised approaches, such as Rugina (2004) and Habermehl et al. (2010), which are designed to work with restricted "built-in" data structures.

1.4 Challenges

Several challenges are on the way of the research of inference mechanisms for loop invariants and full specifications. The major problems of both loop invariant and full specification inference are the shared mutable data structures, abstraction strategics, the combination of multiple domains/theories, and fixed-point calculation. Besides of these difficulties, the precise precondition synthesis and the abduction method in the combined domain are additional tasks for full specification inference.

More specifically, for both loop invariant synthesis and full specification discovery of programs which manipulate shared and mutable sophisticated data

structures, the problems that need to be solved are followed:

- **Sharing (Aliasing) and Mutability** Aliasing refers to the situation where the same memory location/object is shared by multiple pointers/references, namely the memory location/object can be accessed by different access paths. Aliases manifest themselves in many different forms: variable aliasing, parameter aliasing, array subscript aliasing, and so on. The problem of aliasing is highlighted when mutability is presented, which means one data structure could be altered after it is created by any access path. The major difficulty of sharing/aliasing and mutability is that updates one component may affect other seemingly unrelated components.

- **Abstraction Strategies** Analysing about programs in their full details is infeasible due to the complexity of the programs, especially, when handling infinite structures in programs, such as recursive data structures, loops, recursion and so on. The irrelevant or uninteresting details should be eliminated to keep the analysis within the reach of automated tools and also guarantee the termination of the analysis. Abstractions filter out the unnecessary information and improve efficiency of the analysis. It is a non-trivial problem to choose right abstraction strategies that keep as many relevant details and drop as many irrelevant details as possible.

- **Combination of Multiple Domains/Theories** More often than not, programs operate on many different domains and data structures. For example, one single program may change the shape of a list and compares numbers stored in the list at the same time. To verify the

correctness of the program, the analysis need to work over a combined shape and numerical domain to fully capture the behaviour of the program. Combining provers and decision procedures for a range of theories is a challenging problem. When we analyse structural and quantitative aspects of data structures at the same time, we need approaches to combine different theories.

- **Fixed-point calculation** Fixed-point calculation plays a central role in abstract interpretation of the analysis of loops and recursive methods. Widening can be used to guarantee the termination of the analysis in infinite lattices. New widening operators need to be carefully designed to guarantee the termination of the analysis over the combined domain and also not lose too much information.

The following challenges are added to previous tasks for full specification inference mechanism:

- **Precondition Discovering** Abstract interpretation usually needs preconditions as the program context to reason about the program code. It is very tedious to specify the preconditions for every method in sizable programs. An appropriate approach is needed to discover precise preconditions with respect to the demand of users and based on the behaviour and footprint of the unannotated code.

- **Abduction** Abduction is a technique used in the generation of pre-/post-specification for unannotated code. Calcagno et al. (2009) have

applied this technique to shape domain for compositional shape analysis. To extend the abduction technique to the combined shape and pure domain is a challenging task.

1.5 Contributions

The main contributions of this dissertation are the analysis frameworks for automated loop invariants synthesis and full specification discovery for programs manipulating shared and mutable sophisticated data structures. I have demonstrated that it is possible and practical to analyse such programs automatically and precisely by using abstract interpretation in a combined shape and pure domain. More concretely, the main contributions of this dissertation are followed below.

- **Loop Invariant Synthesis** An automated analysis system based on abstract interpretation for loop invariant synthesis is proposed. The analysis system is based on fixed-point computation with specifically designed novel join and widening operators over the combined domain. An abstraction function is defined to concentrate on concrete states for the combined domain. The termination of the system is proved based on the fixed-point algorithm and the widening operator. The soundness of the analysis is proved with respect to the concrete program semantics. The inference framework has been integrated with Hip/Sleek verification system. The experimental results confirm the

viability of the system and show that the need of user supplied loop invariants can be effectively eliminated.

- **Full Specification Discovery** An automated analysis framework for full specification discovery is presented. The framework can discover program's pre-/post-conditions without requiring any specification to be given for the program. The analysis is based on abstract interpretation techniques and is a compositional analysis in the combined shape and pure domain. A novel abstraction with bi-abduction techniques over the combined domain is defined. The framework has been implemented. The experiment results confirm the viability and precision of the framework in finding interesting properties of non-trivial programs.

1.6 Thesis Outline

This thesis is constructed by 7 chapters including this introduction. Chapter 2 provides a literature survey of program analysis and verification techniques, and mainly focuses on numerical domain and separation logic domain. Chapter 3 presents the target programming language of the analysis, and its operational semantics. The specification language used to express our combined abstract domain is also introduced. Chapter 4 describes the synthesis of loop invariants in the combined abstract domain as the first contribution of this thesis. Specially designed abstraction, join and widening operators are explained. Chapter 5 presents the second contribution of the work, a compositional analysis framework to infer both pre-/post-

conditions of heap-manipulating programs with bi-abduction techniques in the combined abstract domain. Chapter 6 shows the implementation, the experimental results and the evaluation of the proposed methods. Chapter 7 concludes and summarises the contributions of this dissertation, and proposes some possible directions for future works.

Chapter 2

Literature Review

Since 1960s and 1970s, formal program analysis and verification have been proposed by pioneers (Floyd, 1967; Hoare, 1969; Dijkstra, 1976) to reason about programs and guarantee the quality of software formally. This area has been continually developed over the past 40 years due to the growing requirements of software engineering. Various techniques have been invented to fit for different scenarios. As this thesis mainly focuses on the functional correctness and memory safety of pointer-based programs, this chapter gives a brief survey of numerical related and memory related works in this field. Many inspirations of this research are drawn among them.

2.1 Hoare Logic and Program Verification

The approaches to describe program behaviours mathematically are the foundation of program analysis and verification. Floyd (1967) and Hoare (1969) took the lead in the race of the use of logical formulae to depict the behaviours of programs. They proposed a formal system that uses a set of axioms and logic rules for rigorously reasoning about the correctness of computer programs. This system has been called Floyd-Hoare logic or Hoare logic. The central feature of Hoare logic is the Hoare triple.

$$\{P\} \quad C \quad \{Q\}$$

The precondition P and postcondition Q specify the behaviours of a program command C. Standard Hoare logic proves only partial correctness, namely program termination needs to be proved separately. Total correctness is built upon the partial correctness and termination proof. A later work (Dijkstra, 1976) presents the notion of the weakest precondition which is proven as equivalent as the former. Burstall (1974) integrates operational semantics for programs with the formal verification method.

Since then a large number of publications is devoted to Hoare logic. The total correctness version of Hoare calculus presented in Manna and Pnueli (1974), is capable of proving that a program can terminate and is logically correct, which extends Hoare's method by proving correctness and termination at once. The notions of expressiveness and relative completeness introduced in Cook (1978), which finds that Hoare logic is only complete in certain cases, are relative to his interpretive semantics. Clarke (1979) researches on

the expressiveness of finite interpretations, with the result that certain programming languages cannot possess a sound and relatively complete Hoare calculus because the halting problem is undecidable for the languages, even if the underlying interpretation is finite. Lipton (1977) claims that the only expressive interpretations should be the standard interpretation of Peano arithmetic and the finite interpretation.

The verification and analysis presented in this thesis are essentially founded on the basis of Hoare logic. As it will be presented in later chapters, our abstract program semantics used for symbolic executions of programs are based on Hoare logic, or more specifically, *separation logic* (as an extension of Hoare logic) which will be surveyed in a later section.

2.2 Abstract Interpretation

If one designs an abstraction or representation of the program states, then it is possible to capture the interesting facts of the program and infer the loop invariants and program specifications by extracting information from the program code. This premise is the basis of data-flow analysis (Kildall, 1973) and abstract interpretation. The framework of abstract interpretation was introduced in Patrick Cousot's and his collaborations' series of foundational works (Cousot and Cousot, 1976, 1977a; Cousot and Halbwachs, 1978). Cousot and Cousot (1976) introduce the basic way to use "abstract" (symbolic) values associated with variables instead of the "concrete" values

during real execution of the program. For example, if we choose two domains, one is natural numbers $\mathbb{N}$ as the concrete domain recording concrete values of variables, and the other is the set S of integer intervals as the abstract domain, then we define two functions α and γ as

$$\begin{aligned} \alpha(N) &= [\mathsf{min}(N), \mathsf{max}(N)], N \subseteq \mathbb{N} \\ \gamma(s) &= s, s \subseteq \mathbb{N} \end{aligned}$$

where we write $[n, +\infty)$ as $[n, +\infty]$ for expression convenience. Here we call α the abstraction function and γ the concretisation function. The abstraction function maps a set of concrete values to an abstract value, and the concretisation function runs in the reversed way. Note that we have the relationship $\forall N \in \mathcal{P}(\mathbb{N}) \cdot N \subseteq \gamma(\alpha(N))$ and $\forall s \in S \cdot s = \alpha(\gamma(s))$, these two functions hence create a *Galois connection* to link both the concrete and abstract domains together. In an analysis when we are confronted with an infinite increasing chain $1, 2, 3, \ldots$ as the value sequence of some variable, we can condense it as an abstract value $[1, +\infty]$ in S to force convergence.

On the basis of the work (Cousot and Cousot, 1976), Cousot and Cousot (1977a) propose an approach to an approximation of fixed-point to construct a unified lattice model for static program analysis. Its general idea is to have some ordering over both concrete and abstract states and an induced (complete) semi-lattice over them, and regard the (recursive) program being analysed as a transition function f, which is monotonic over the concrete domain (and lifted to the abstract domain). Then the least fixed-point of f ($\mathsf{lfp}\, f$) can be considered as the semantics of f, which may be computed with a fixed-point iteration process.

Cousot and Halbwachs (1978) apply the above mentioned two pieces of work to discover the assertions (of linear type) that can be deduced from the semantics of the program. It can often discover relations which are never stated explicitly in the program.

After that, Cousots still have sequent works to make the framework of abstract interpretation more complete. Cousot and Cousot (1979) exhibit a systematic way to design program analysis frameworks. It shows a (both forward and backward) deductive semantics of programs as the standard of soundness, based on which it studies the design of a space of approximate assertions, and the design of the approximate predicate transformer induced by such assertions. In this way, it brings forward some global program analysis methods. This framework is an excellent foundation for other program analysis practice, while its semantics is rectified again in a later work (Cousot, 1981).

For the approximation methods in the fixed-point calculation of abstract interpretation, Cousot and Cousot (1977a) also introduce some initial ways which are still frequently referenced today. One is static in that it can be understood as the simplification of the equation involved in the concrete semantics into an approximate abstract equation. Galois connections are used in this method to formalise this discrete approximation process. The second is dynamic in that it takes place during the iterative resolution of the abstract equation (or system of equations). This separation introduces additional flexibility which allows for both expressiveness and efficiency. It also introduces the idea of using widening and narrowing operators (∇ and $\triangle$) to

accelerate/force convergence for fixed-point approximation (especially when the lattice is infinite and does not satisfy the ascending chain condition). An instance of this follows the previous example of natural real numbers and intervals. In this example, we may have a widening operator ∇ by choosing a finite ramp

$$0 = r_0 < r_1 < \cdots < r_k = +\infty$$

and the widening's definition is

$$\begin{aligned} \emptyset \nabla [l', h'] &= [l', h'], \text{ or} \\ [l, h] \nabla [l', h'] &= \begin{array}{l} [\text{ if } l' < l \text{ then } \mathsf{max}\{r_i | r_i \leq l'\} \text{ else } l, \\ \quad \text{if } h' > h \text{ then } \mathsf{min}\{r_i | h' \leq r_i\} \text{ else } h\,] \end{array} \end{aligned}$$

so that if we make $r_i - r_{i-1} = 1$, then we will have an infinite ascending chain during the analysis $[0,0], [0,1], [0,2], \cdots$. We can use ∇ to widen each state with its consecutive state in order to make the chain converge as $[0,0], [0,1]$, $[0, +\infty]$. This idea is essential in the work because it offers a way to deal with ascending chains in infinite lattices or to speed up convergence in case of a combinatorial explosion.

To automate the verification process, program analysis techniques are widely applied on the basis of abstract interpretation. More recently, Müller-Olm and Seidl (2004) apply linear algebra techniques to precise dataflow analysis to describe analyses which are determined for each program point identities that are valid among the program variables whenever control reaches that program point (as their main approach). They fully interpret assignment statements with affine expressions on the right hand side to compute the set of all affine relations and polynomial relations of bounded degree. Their complexity is worth noting to be linear to program size and polynomial to

the number of variables. Popeea and Chin (2006) also introduce the notion of affinity to characterise how closely related two polyhedra are. Then they try to find related elements in the polyhedron (base) domain to allow the formulation of precise hull and widening operators to lift to the disjunctive (powerset extension of the) polyhedron domain. In this way, they effectively prevent the original convex-hull's loss of precision. Gulwani and Tiwari (2007) use a backward analysis to propagate information with "generic assertions" analysis and to simplify such assertions with unification. Their analysis and implementation are constructed on this technique. The above mentioned works are focused mostly on the numerical domains. Pham et al. (2011) build an abstract interpretation tool for quantified bag constraints. They also make use of affinity-based hulling and widening techniques to calculate precise disjunctive fixpoints over bag domain. In this thesis, their achievements can be utilised as solvers to the pure part of our combined domain.

2.3 Shape property and Separation Logic

For the modelling of program's memory state, we use the technique of separation logic (O'Hearn and Pym, 1999; Reynolds, 2000; O'Hearn et al., 2001; Reynolds, 2002). In this section, a brief background of it is introduced.

As a pioneer of separation logic, a logic of bunched implications (BI) is introduced by O'Hearn and Pym (1999), which is merged from two parts: additive

intuitionistic logic and multiplicative intuitionistic linear logic. Models of propositional BI's proofs are given by bi-cartesian doubly closed categories, combining freely semantics from both logic families. This work also develops a first-order predicate version of BI with newly invented universal and existential quantifiers.

However, BI is no more than a theoretical logic model until Reynolds has presented his work (Reynolds, 2000) to reason about resource-sensitive programs, and his logic model is analogous to BI's. Generally, it is an extension of Hoare's approach to prove the correctness of imperative programs that perform destructive updates to data structures, and that contain more than one pointer to the same location. It invents an "independent conjunction" $P \wedge Q$ that holds only when P and Q are both true and depend upon distinct areas of storage, whose semantics is exactly the same as the linear conjunction of BI. It is a nice coincidence that they come to the same point from two different ways, which happened several times in the history of computer science such as Turing's computing machine and Church's λ-calculus.

After that, these two branches of research group have cooperated to deliver a series of works (O'Hearn et al., 2001; Reynolds, 2002) to set up the foundation of separation logic which can be used to reason about heap memory state. In separation logic, two more connectives are added to classical logic: separation conjunction $*$ and spacial implication $-\!\!*$. The formula $P * Q$ asserts that two heaps described by P and Q are domain-disjoint. Meanwhile, $P -\!\!* Q$ asserts that if the current heap is extended with a disjoint heap which is described by P, then Q holds in the extended heap. Such connectives are

supported by a low-level storage model based on both the stack and the heap memory. The model supports the basic program operations such as lookup, mutation (update), allocation and deallocation with a series of Hoare logic style reasoning rules. It also provides unrestricted memory address arithmetic. The frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

where no variable occurring free in R is modified by C.

is emphasised as the base of *local reasoning* because it allows the reasoning of programs to concentrate on programs' *footprint* (the heap that the program actually manipulates), instead of a large global heap state. This is significant as it entitles the reasoning with the potential to scale up. Separation logic's assertion language is also formalised on a possible world model of BI. The soundness and relative completeness are also discussed in the paper, as well as latest results of separation logic with the illustration of its possible applications in the field of program reasoning.

For separation logic itself, there are some other works to address. Yang and O'Hearn (2002) present a semantic analysis of the soundness and relative completeness of separation logic for the frame axiom to be inferred automatically, with the result that it can be avoided when writing specifications. Calcagno et al. (2001) discuss on some computability and complexity results of separation logic, where it points out that the validity of separation logic formulae is not decidable; however, the validity over a restricted subset of separation logic formulae is fortunately decidable with certain complexity. Following it, Berdine et al. (2004) provide a fragment of separation logic whose entailment checking problem is decidable with a sound and complete

algorithm to solve it, which plays an important theoretical role in their later works of program analysis. Calcagno et al. (2007) study the semantic structures lying behind separation logic by the concept of local action, which is a state transformer that mutates the state in a local way. It formulates local actions for a class of models called separation algebras, abstracting from the memory and other specific concrete models used in work on separation logic. Local actions provide a semantics for a generalised form of (sequential) separation logic, and allow a general soundness proof for a separation logic for concurrency.

Smallfoot (Berdine et al., 2005a) is a verification tool based on separation logic. It makes use of a symbolic execution designed to work with a fixed set of shape predicates, most notably, the list segment predicate (Berdine et al., 2005b). It is the first attempt to use separation logic in the verification of pointer safety and simple shape properties. The data structures and properties that they verify are quite simple compared with the user-defined predicates.

This thesis work is based on the improvement of another state-of-the-art program verifier Hip/Sleek (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008). Its overview is given in Figure 2.1. The front-end of the system is a standard Hoare-style forward verifier Hip, which invokes the separation logic prover Sleek. The Hip verifier comprises a set of forward verification rules to systematically check that the precondition is satisfied at each call site, and that the declared postcondition is successfully verified (assuming the given precondition) for each method definition. Given two

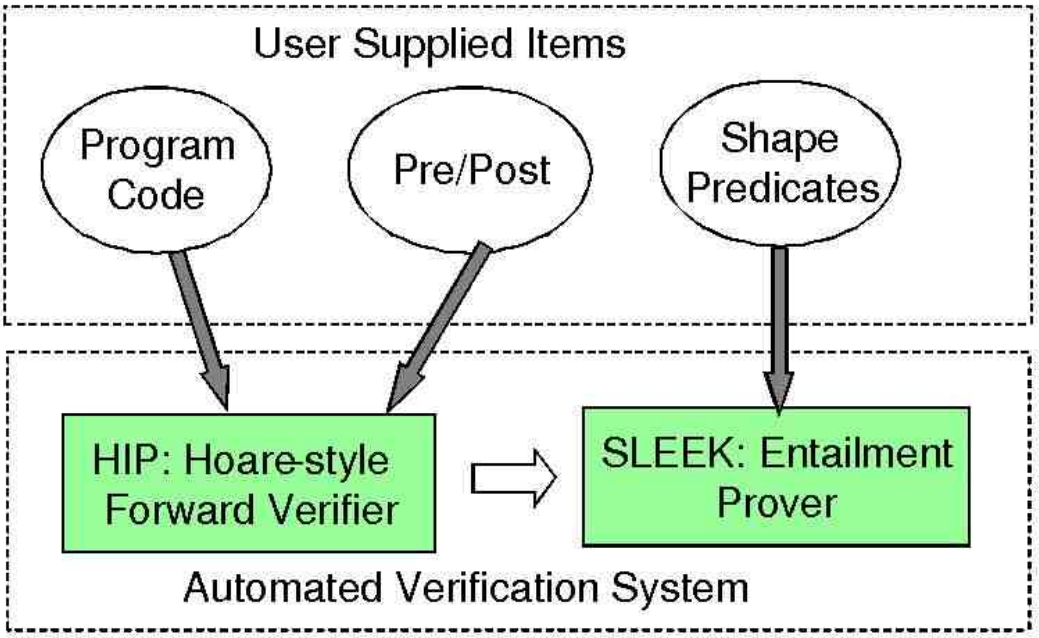

Figure 2.1: The HIP/SLEEK verification system.

states Δ_1 and Δ_2, the separation logic prover SLEEK attempts to prove that Δ_1 entails Δ_2; if it succeeds, it returns a frame Δ_R so that $\Delta_1 \vdash \Delta_2 * \Delta_R$. As discussed previously, it can express and process multiple types of program properties such as shape, quantitative and content ones in states Δ. We want to keep all of its merits and improve it by discovering loop invariant and full specification automatically. Meanwhile, we use the SLEEK tool as our main solver for entailment checking.

This research will be different from the aforementioned ones in the employed techniques (like separation logic and entailment checking) in that we can describe the shape properties in a more natural way from the user's perspective and will still remain expressive and computationally feasible.

2.4 Model Checking

Model checking (Clarke and Emerson, 1981) represents a fairly different approach to the proof of software correctness. It was originally designed to verify finite-state systems by exhausting the whole set of computation states according to some specification described in temporal logic, and it achieved great success on circuit design and implementation (McMillan, 1992). Such success intrigued researchers' interest in applying model checking to the field of software. The key techniques for such application are abstraction (like predicate abstraction (Ball et al., 2001) and counterexample-guided abstraction refinement (Clarke et al., 2000, 2003)), as software usually has infinite computation states which are beyond the capability of model checking. These abstractions are even borrowed into some analysis works, such as in Sagiv et al. (2002) and Balaban et al. (2005). With appropriate abstraction techniques, model checking tools are generally automatic and thus requires no user intervention. Some representatives of such tools include the general framework of model checkers SPIN (Holzmann, 2004), the SLAM model checker for drivers (Ball and Rajamani, 2001), the BLAST model checker for C programs with lazy counterexample-guided abstraction refinement (Henzinger et al., 2003) and Java PathFinder for Java programs by NASA (Visser et al., 2003).

2.5 Summary

This chapter has surveyed the state-of-the-art in the field of software quality assurance, especially program verification and analysis. It mainly covers two types of topics, one is the foundational techniques that we use in this thesis, and the other is some related works by peers in the similar area of program verification and analysis.

Chapter 3

Language And Semantics

In this chapter, both target programming language and specification language are introduced with their semantics. The target programming language, i.e. the language is used to be verified, is a standard object-based language, and the specification language is a predicate-based specification language which is used to express the program specifications and abstract program states.

Predicates are the core of the specification language. The predicates are inductively defined as they are in HIP/SLEEK (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010), and are able to capture recursive data structures with sophisticated program properties involving not only structural aspects but also quantitative aspects and content of data

structures. In this chapter, firstly, the programming language is specified. Then the base of our specification language, separation logic, is depicted. Finally, the specification language itself is described.

3.1 Target Programming Language

3.1.1 Grammar

The target programming language used in our system is a fully typed object-based language which may be viewed as a subset of popular type-safe programming languages, such as Java or C♯. Its grammar is formally defined in Figure 3.1.

A program *Prog* in our language consists of a list of type declarations *tdecl* and a list of method definitions *meth*. The type declarations include class types *classt* used in programs, user-defined predicates *spred* for specifications. *spred* is written in our specification language, thus we leave them until section 3.3. Compared with those fully-fledged object-oriented languages, our language has omitted some features which are orthogonal to this thesis' interest, such as inheritance, dynamic dispatch, concurrency, array, exception, and so on. The semantics of most constructs of the language are understood in the usual sense that one would find in languages such as Java or C♯, except for the class declaration, which declares a class type without instance

Program	*Prog*	::=	$t\vec{d}ecl\ m\vec{e}th$	
Type declaration	*tdecl*	::=	*classt* | *spred*	
Class declaration	*classt*	::=	`class` c { $fi\vec{e}ld$ }	
Field declaration	*field*	::=	$t\ v$	
Type	t	::=	$c \mid \tau$	
Procedure declaration	*meth*	::=	$t\ mn\ ((t\ \vec{v}); (t\ \vec{v}))\ ms\vec{p}ec\ \{e\}$	
Built-in type	τ	::=	`int` | `bool` | `void`	
Expression	e	::=	d	heap-insensitive atomic
		|	$d[v]$	heap-sensitive atomic
		|	$v{=}e$	assignment
		|	$e_1; e_2$	sequence
		|	$t\ v;\ e$	local variable
		|	`if` $(v)\ e_1$ `else` e_2	
		|	`while` $v\ \{e\}$ `where` $ms\vec{p}ec$	
Heap-insensitive atomic	d	::=	-	skip
		|	`null`	null reference
		|	k^τ	constant
		|	v	variable
		|	`new` $c(\vec{v})$	allocation
		|	$mn(\vec{u}; \vec{v})$	method call
Heap-sensitive atomic	$d[v]$	::=	$v.f$	field read
		|	$v.f{=}w$	field write
		|	`free`(v)	deallocation

Figure 3.1: A target programming language.

methods or dynamic dispatch. Other than that, they behave like normal classes: instances of a class type can be (dynamically) allocated (`new` $c(\vec{v})$), their fields can be read (`v.f`) and updated (`v.f` = `w`), references to them can be passed to and from methods, and so on. A type t can either be a class type c or a primitive built-in type τ.

To express program specifications, users are allowed to express the specifications *mspec* for each method or loop in our language. Like in C♯, we support both pass-by-value and pass-by-reference parameters, which are separated with a semicolon (;) where the ones before ';' are pass-by-value and the ones after are pass-by-reference. The grammar for these annotations will be presented in section 3.3. The meaning of a pair of precondition and postcondition is that if the method is invoked in a program state that satisfies its precondition, the method will not have any memory faults such as null or dangling pointer dereferences. Furthermore, if the method terminates, it terminates in a state that satisfies the postcondition. Otherwise, if the program state does not satisfy the precondition, then the verification fails and a catastrophic error is reported with its location in the program. In other words, we adopt the partial correctness semantics of Hoare triples with tight interpretation (Yang and O'Hearn, 2002).

Without loss of generality, our language is expression-oriented, the body of a method hence is an expression composed of standard instructions and constructors of the language. e is the (recursively defined) program constructor, and d and $d[v]$ are atomic instructions. Here $d[v]$ has some specific requirement over the memory state (such as v must be allocated at a valid part of

heap memory) and is hence named heap-sensitive atomic instruction, whereas d does not have such requirements and is called heap-insensitive atomic instruction. As these will be seen in later chapters, the two sorts of instructions are treated differently during the analysis of a program.

We have some further assumptions over the programs, and they are well-formed according to the following rules. Each program is type-safe. Classes, predicates and methods should have distinct names. Local variables in the same scope are distinct. Meanwhile, we do not allow the syntactic sugar for local variables to hide variables from outer scopes or method parameters.

3.1.2 Operational Semantics

This section defines the operational semantics of our programming language. Before doing that, we firstly define the semantic domains. Locations in our system correspond to object identifiers (which can be practically regarded as memory locations). Values include primitive values, locations, and the special value `null` which does not correspond to any object identifier. Objects are finite partial maps that map from field names to values. Primitive values include integer numbers and boolean values. Return types of methods could be `Void`.

We define a small-step operational semantics for our language as transitions between machine configurations. Each machine configuration is a triple con-

OS-VAR $$\langle s, h, v\rangle \hookrightarrow \langle s, h, s(v)\rangle$$

OS-CONST $$\langle s, h, k\rangle \hookrightarrow \langle s, h, k\rangle$$

OS-SEQ $$\langle s, h, \text{-}; e\rangle \hookrightarrow \langle s, h, e\rangle$$

OS-ASSIGN-1 $$\langle s, h, v{=}k\rangle \hookrightarrow \langle s[v{\mapsto}k], h, \text{-}\rangle$$

OS-FIELD-READ $$\langle s, h, v.f\rangle \hookrightarrow \langle s, h, h(s(v))(f)\rangle$$

OS-LOCAL $$\langle s, h, \{t\ v;\ e\}\rangle \hookrightarrow \langle [v{\mapsto}\bot]{+}s, h, \texttt{ret}(v, e)\rangle$$

OS-RET-1 $$\langle s, h, \texttt{ret}(\vec{v}, k)\rangle \hookrightarrow \langle s{-}\{\vec{v}\}, h, k\rangle$$

OS-PROG $$\frac{\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle}{\langle s, h, e_1; e_2\rangle \hookrightarrow \langle s_1, h_1, e_3; e_2\rangle}$$

OS-ASSIGN-2 $$\frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, v{=}e\rangle \hookrightarrow \langle s_1, h_1, v{=}e_1\rangle}$$

OS-RET-2 $$\frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, \texttt{ret}(\vec{v}, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(\vec{v}, e_1)\rangle}$$

OS-FIELD-WRITE $$\frac{r = h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1 = h[s(v_1){\mapsto}r]}{\langle s, h, v_1.f = v_2\rangle \hookrightarrow \langle (s, h_1)\rangle}$$

OS-IF-1 $$\frac{s(v){=}\texttt{true}}{\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle}$$

OS-IF-2 $$\frac{s(v){=}\texttt{false}}{\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_2\rangle}$$

OS-NEW $$\frac{\begin{array}{c}\texttt{data}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \\ \iota{\notin}dom(h) \quad r{=}c[f_1{\mapsto}s(v_1), .., f_n{\mapsto}s(v_n)]\end{array}}{\langle s, h, \texttt{new}\ c(\vec{v})\rangle \hookrightarrow \langle s, h[\iota \mapsto r], \iota\rangle}$$

OS-WHILE-1 $$\frac{s(b){=}\texttt{true}}{\langle s, h, \texttt{while}\ (b)\ \{e\}\rangle \hookrightarrow \langle s, h, e;\ \texttt{while}\ (b)\ \{e\}\rangle}$$

OS-WHILE-2 $$\frac{s(b){=}\texttt{false}}{\langle s, h, \texttt{while}\ (b)\ \{e\}\rangle \hookrightarrow \langle s, h, \text{-}\rangle}$$

OS-CALL $$\frac{\begin{array}{c}s_1{=}s[w_i{\mapsto}s(v_i)]_{i=1}^{m-1} \\ t_0\ mn((t_i\ w_i)_{i=1}^{m-1}; (t_i\ w_i)_{i=m}^{n})\ \{e\}\end{array}}{\langle s, h, mn(\vec{v})\rangle \hookrightarrow \langle s_1, h, \texttt{ret}(\{w_i\}_{i=1}^{m-1}, [v_i/w_i]_{i=m}^{n}e)\rangle}$$

Figure 3.2: Operational semantics.

sisting of:

- Stack s. Stacks are modelled as total maps from variables to values and locations. Note that it is viewed as a "stackable" mapping, where a variable v may occur several times, and $s(v)$ always refers to the value of the variable v which was the most recently popped one.[1]

$$s :: \mathsf{Var} \to \mathsf{Val} \cup \mathsf{Loc}$$

- Heap h. We model heaps as finite partial maps from locations to objects. Objects are expected to conform to their defined class types.

$$h :: \mathsf{Loc} \rightharpoonup_{fin} \mathsf{ObjVal}$$

- Current program code e. Program execution terminates when e is -, a value of type `void`.

Each reduction step can then be formalised as a small-step transition of the form:

$$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

A configuration is *final* if e is a value or -. A configuration is *stuck* if it is not *final* and there is no applicable transition. The full set of transitions is given in Figure 3.2. The operation $[v \mapsto \nu] + s$ "pushes" the variable v to s with the value ν, and $([v \mapsto \nu] + s)(v) = \nu$. The operation $s - \vec{v}$

[1] A more formal definition for s would mark different occurrences of the same variable with different "frame" numbers. We omit the details here.

"pops out" variables $\vec{v}$ from the stack s. $s[v \mapsto k]$ is a mapping which keeps all the mappings in s except that of v (which is now specified to be mapped to k). We also abuse this notation for a class type identifier c to denote a region of heap (mappings) in the form $c[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)]$, which is essentially a heap location where fields f_i are further mapped to values $s(v_i)$, $i = 1, \ldots, n$. $\perp$ represents an arbitrary value. We also introduce an intermediate construct as result which is returned by expression/method call $\texttt{ret}(\vec{v}, e)$, where $\vec{v}$ will be dropped from s after the evaluation/invocation of e as to simulate the behaviour of stack. Whenever such a result is yielded, we assume it is stored in a special logical variable `res` although `res` is never explicitly put in the stack s.

3.2 Separation Logic

Our specification language is built on top of separation logic (O'Hearn and Pym, 1999; Reynolds, 2000; Ishtiaq and O'Hearn, 2001; Reynolds, 2002), designed for reasoning about programs that manipulate shared mutable pointer-based data structures. As mentioned in section 2.3, the distinguished feature of separation logic is its *local reasoning* about data structures linked by pointers and allocated in heap (Distefano et al., 2006). It means that reasoning about a command concerns only the part of the heap that the command accesses, a.k.a. the command's footprint. Note that local reasoning is not only a registered patent for separation logic; it also exists in the original formulation of Hoare logic (Hoare, 1969) with the substitution treatment

in assignment. However, such local reasoning will be lost if heap-based data structure and aliasing are introduced to the programming language. This loss of locality is noted as the pointer swing problem by Hoare and He (1999). In this scenario, separation logic restores the capability to reason locally by means of two technical novelties: 1) the separation conjunction $*$ and 2) tight interpretation of Hoare triples (Yang and O'Hearn, 2002).

A key insight leading to separation logic is that program logics for reasoning about heap-manipulating programs should be explicit for the heap. In other words, program heaps should be one part of the model of a program logic. The satisfiability of a separation logic formula Δ in a program state is thus typically enforced by the semantics relation

$$s, h \models \Delta$$

where s is a model of the program stack, and h is the program heap.

Separation logic introduces a relation points-to $\mapsto$, and three new operators: empty heap `emp`, separating conjunction $*$, and separating implication $-\!\!*$. A points-to formula $\texttt{x} \mapsto \texttt{y}$ describes a singleton heap with only one cell at address `x` that stores value `y`. Formula `emp` holds on empty heap, namely, no data/object is allocated on heap. Formula $\Delta_1 * \Delta_2$ describes a heap that can be partitioned into two domain-disjoint heaps described by Δ_1 and Δ_2. Formula $\Delta_1 -\!\!* \Delta_2$ describes a heap that if it is extended with a disjoint heap represented by Δ_1, then Δ_2 holds in the extended heap. In other words, $\Delta_1 -\!\!* \Delta_2$ captures the heap described by Δ_2, where the heap corresponding to Δ_1 is "taken away". The formal semantics of these operators will be defined formally in subsection 3.3.4.

Tight interpretation is another key aspect of separation logic, which ensures "well-specified programs do not go wrong" (Reynolds, 2005). Under this interpretation, a valid Hoare triple $\{\Delta_1\}\ e\ \{\Delta_2\}$ guarantees that command e should never encounter a memory fault if it is started in a program state that satisfies Δ_1. One significant prerequisite of this interpretation requires the precondition Δ_1 of a command to guarantee that all memory locations accessed by the command, except for the freshly allocated ones, are allocated beforehand. In the setting of separation logic, a memory location `x` is considered allocated if the points-to fact $\texttt{x} \mapsto \_$ is presented. More specifically, Hoare triples for heap-accessing commands in separation logic are described as follows:

- Field read:

$$\begin{aligned} &\{x \mapsto [v_1, \ldots, v_i, \ldots, v_n]\} \\ &\qquad x.f_i \\ &\{x \mapsto [v_1, \ldots, v_i, \ldots, v_n] \wedge \texttt{res}{=}v_i\} \end{aligned}$$

  where `res` is the special variable which denotes the resulted value of an expression.

- Field write:

$$\begin{aligned} &\{x \mapsto [v_1, ..., v_{i-1}, v_i, v_{i+1}, ..., v_n]\} \\ &\qquad x.f_i{=}v \\ &\{x \mapsto [v_1, ..., v_{i-1}, v, v_{i+1}, ..., v_n]\} \end{aligned}$$

The above axioms illustrate the main characteristics of separation logic. In order to analyse a heap-accessing command, it must be explicitly proved that

the heap location under consideration is allocated. Meanwhile, the reward is that any other heap locations can be ignored safely.

The interplay of separation conjunction and tight interpretation makes local reasoning possible, which is formalised by the frame rule in separation logic:

$$\frac{\{\Delta_1\}\ e\ \{\Delta_2\}}{\{\Delta_1 * \Delta_3\}\ e\ \{\Delta_2 * \Delta_3\}}\ \mathsf{mods}(e) \cap \mathsf{fv}(\Delta_3) = \emptyset$$

where $\mathsf{mods}(e)$ returns the set of variables modified by command e. Note that $\mathsf{mods}(e)$ includes neither modified fields, nor the variables used to reach these fields. $\mathsf{fv}(\Delta_3)$ returns the set of free variables which occurs in formula Δ_3. The crucial power of the frame rule is that it allows a global property to be derived from a local one, without necessity to look at other parts of the program.

3.3 Specification Language

The specification language is on the basis of a predicate-based specification methodology, wherein the main annotation construct is the shape predicate, each of which describes a data structure. The aim of using this scheme is to allow users to design their own predicates for shapes and relevant properties (numerical and content ones), in order to capture the desired level of program correctness. The advantages of this methodology include that it unifies heterogeneous techniques and annotations in a homogeneous way for the verification of linked data structures. Predicates also eliminate the need

for an explicit ownership scheme; they capture sufficient information for us to perform verification of properties that involve closures. Finally, it permits us to easily decompose the properties to be verified for a shape predicate, which is beneficial to verify various levels of program correctness.

The grammar for the specification language is given in Figure 3.3. Each shape predicate *spred* has a name c, a list of parameters $\vec{v}$, and a body Φ. Each predicate also has a parameter `root`, written to the left of the predicate name c, which denotes a `root` pointer to the data structure captured by the predicate. A `root` pointer is one from which all objects in the data structure can be reached. `root` is a reserved identifier used only in predicate definitions. Δ is the abstract state, and Φ is the normalised state which is a separation logic formula essentially in disjunctive normal form. The method specifications *mspec* are written in these states where Φ_{pr} and Φ_{po} denote the precondition and postcondition respectively. Each disjunct σ consists of a heap formula κ in the shape domain and a pure formula π in disjunctive convex polyhedra domain and bag (multi-set) domain, i.e. the pure (numerical) domain. The heap formula κ consists of $*$-conjoined atomic heap formulae $\mathtt{p}{::}c\langle\vec{v}\rangle$. Such atomic heap formula $\mathtt{p}{::}c\langle\vec{v}\rangle$ can denote either (i) a points-to fact $\mathtt{p} \mapsto c[\vec{v}]$ if c is a class name, or (ii) a predicate instance $c(\mathtt{p}, \vec{v})$ if c is a predicate name. The pure part π consists of heap-independent formulae, such as formulae for Presburger arithmetic, formulae for pointer equality/disequality and formulae in multiset theory. As shown in the figure, Presburger arithmetic formula (s) is made up of integer constraints, variables, addition, subtraction, scalar multiplication, maximum/minimum values and cardinality of multiset. As for multiset (bag) theory, we allow expression of (quantified) value membership, subset relationship and bag arithmetic (such

as union, intersection and subtraction). To make automated verification possible, we require that there is a sound and terminating method to decide the validity of heap-independent logic.

Shape predicate	*spred*	$::= \texttt{root}::c\langle\vec{v}\rangle \equiv \Phi$
Specification	*mspec*	$::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po}$
Abstract state	Δ	$::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta$
Normalised state	Φ	$::= \bigvee \vec{\sigma}$
Conjunctive state	σ	$::= \exists \vec{v} \cdot \kappa \wedge \pi$
Heap formula	κ	$::= \texttt{emp} \mid \texttt{v}::c\langle\vec{v}\rangle \mid \kappa_1 * \kappa_2$
Pure formula	π	$::= \gamma \wedge \phi \mid \pi_1 \wedge \pi_2$
Aliasing	γ	$::= v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \gamma_1 \wedge \gamma_2$
Pure constr.	ϕ	$::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$
Boolean	b	$::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2$
Numerical constr.	a	$::= s_1 = s_2 \mid s_1 < s_2$
Presburger arith.	s	$::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid \max(s_1, s_2)$
		$\mid \mathit{min}(s_1, s_2) \mid \lvert\texttt{B}\rvert$
Bag constr.	φ	$::= v \in \texttt{B} \mid \texttt{B}_1 = \texttt{B}_2 \mid \texttt{B}_1 \sqsubset \texttt{B}_2 \mid \forall v \in \texttt{B} \cdot \phi \mid \exists v \in \texttt{B} \cdot \phi$
Bag arith.	B	$::= \texttt{B}_1 \sqcup \texttt{B}_2 \mid \texttt{B}_1 \sqcap \texttt{B}_2 \mid \texttt{B}_1 - \texttt{B}_2 \mid \{\vec{v}\}$
	k	$\in$ Integer constans
	c	$\in$ Class or predicate names
	v	$\in$ Variables

Figure 3.3: The specification language.

For the verification of programs, we regard σ as a *conjunctive abstract program state*, and use SH to denote a set of such conjunctive states as symbolic heaps. During a verification process, the *abstract program state* at each pro-

gram point will be a disjunction of σ's which is denoted as Δ, and we name the set of such formulae as $\mathcal{P}_{\mathsf{SH}}$ which describes our analysis domain, i.e. the combined shape and pure (numerical) domain.

A Δ can always be normalised into the Φ form. The normalization rules for separation constraints are given in Figure 3.4.

$$(\Delta_1 \vee \Delta_2) \wedge \pi \rightsquigarrow (\Delta_1 \wedge \pi) \vee (\Delta_2 \wedge \pi)$$
$$(\Delta_1 \vee \Delta_2) * \Delta \rightsquigarrow (\Delta_1 * \Delta) \vee (\Delta_2 * \Delta)$$
$$(\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2) \rightsquigarrow (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2)$$
$$(\kappa_1 \wedge \pi_1) \wedge (\pi_2) \rightsquigarrow \kappa_1 \wedge (\pi_1 \wedge \pi_2)$$
$$(\gamma_1 \wedge \phi_1) \wedge (\gamma_2 \wedge \phi_2) \rightsquigarrow (\gamma_1 \wedge \gamma_2) \wedge (\phi_1 \wedge \phi_2)$$
$$(\exists x \cdot \Delta) \wedge \pi \rightsquigarrow \exists y \cdot ([y/x]\Delta \wedge \pi)$$
$$(\exists x \cdot \Delta_1) * \Delta_2 \rightsquigarrow \exists y \cdot ([y/x]\Delta_1 * \Delta_2)$$

Figure 3.4: Normalization Rules for Separation logic formula

Finally, when we write abstract program states or program specifications, we use three kinds of variables: program variables, logical variables as parameters of predicates related to program variables (such as xn in $\mathtt{x}{::}\mathtt{c}\langle xn\rangle$), and logical variables to record intermediate states. We denote a program variable's initial value as unprimed, and its current (and hence latest) value as primed (Nguyen et al., 2007; Chin et al., 2007). For instance, for a code segment $\mathtt{x} = \mathtt{x}{+}\mathtt{1};\ \mathtt{x} = \mathtt{x}{-}\mathtt{2}$ starting with state $\{\mathtt{x}{>}\mathtt{1}\}$, we have the following

reasoning method:

$$\{\texttt{x}'{=}\texttt{x} \wedge \texttt{x}{>}1\}\ \texttt{x=x+1}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}{+}1\}\ \texttt{x=x-2}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}_1{-}2 \wedge \texttt{x}_1{=}\texttt{x}{+}1\}$$

where the final value of $\texttt{x}$ is recorded in variable $\texttt{x}'$, and $\texttt{x}_1$ keeps an intermediate state of $\texttt{x}$.

3.3.1 Shape Predicates

Our specification language allows the user to describe both the shape of data structures as well as their quantitative properties and contents, and to use them to capture the desired level of program correctness. Shape constraints of the data structures are described by separation logic. Quantitative constraints, such as numerical properties and content of collections, are described by arithmetic or multiset formulae. For example, with a singly linked list node

$$\texttt{class Node } \{\ \texttt{int val};\ \texttt{Node next};\ \}$$

as data structure, a user interested in pointer-safety may define a predicate to depict the list shape as in Distefano et al. (2006); Calcagno et al. (2009):

$$\texttt{root::list}\langle\rangle \ \equiv\ (\texttt{root=null}) \vee (\exists \texttt{i}, \texttt{q} \cdot \texttt{root::Node}\langle \texttt{i}, \texttt{q}\rangle * \texttt{q::list}\langle\rangle)$$

The sole parameter `root` for the predicate `list` is the root pointer that refers to the list. As mentioned earlier, we use a uniform notation $\texttt{p::c}\langle\vec{v}\rangle$ to denote either a singleton heap or a predicate. If `c` is a class type node, the notation represents a singleton heap, $p{\mapsto}c[\vec{v}]$, e.g. the $\texttt{root::Node}\langle\texttt{i},\texttt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape

c with parameters $\vec{v}$, e.g., the q::list$\langle\rangle$ above. In the inductive case, the separation conjunction $*$ ensures that two heap portions (representing the head node and the tail list respectively) are domain-disjoint. Our predicates use existential quantifiers for local values and pointers, such as i and q.

Yet another user may be interested to track the length of a list to analyse quantitative measures, such as heap/stack resource usage. Therefore, the predicate can be defined in a similar manner as in Magill et al. (2008):

$$\texttt{ll}\langle\texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node}\langle\_, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{ll}\langle\texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

where we use the following syntax sugar: (i) default root parameter in LHS may be omitted, (ii) unbound variables, such as q and m, are implicitly existentially quantified, and (iii) _ denotes existentially quantified anonymous variable. The parameter n of the predicate represents an abstract value. Such value is not taken from a concrete heap location, but rather computed from the pure formulae, which are usually based on the structure of the underlying heap. This value is derived automatically by entailment when a predicate is proved from a program state during the verification.

Meanwhile, this predicate may still be extended to support a higher-level of correctness with multiset (bag) property to capture the list's content:

$$\texttt{llB}\langle\texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root}{::}\texttt{Node}\langle\texttt{v}, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{llB}\langle\texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1)$$

which also implicitly suggests the list's length with |S|. This predicate can be strengthened furthermore if it is necessary, so as to verify a sorting algorithm:

$$\begin{aligned}\texttt{sllB}\langle\texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee \\ & (\texttt{root}{::}\texttt{Node}\langle\texttt{v}, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{sllB}\langle\texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1 \wedge (\forall \texttt{x}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{x}))\end{aligned}$$

The constraint $\forall \mathtt{x} \in \mathtt{S}_1 \cdot \mathtt{v} \leq \mathtt{x}$ guarantees the sortedness property, which is adhered in the predicate. Therefore, it can be seen that the user is expected to provide predicate definitions in terms of their required correctness level and program properties. These predicates may be non-trivial but can be reused multiple times for specifications of different methods.

As an example, a sorted doubly linked list segment with multiset property can be described by:

$$\begin{aligned}\mathtt{dlls}\langle \mathtt{pr}, \mathtt{bo}, \mathtt{bi}, \mathtt{S} \rangle \equiv\ & (\mathtt{root}{=}\mathtt{bo} \wedge \mathtt{pr}{=}\mathtt{bi} \wedge \mathtt{S}{=}\emptyset) \vee \\ & (\mathtt{root}{::}\mathtt{Node2}\langle \mathtt{v}, \mathtt{pr}, \mathtt{nx} \rangle * \mathtt{nx}{::}\mathtt{dlls}\langle \mathtt{root}, \mathtt{bo}, \mathtt{bi}, \mathtt{S1} \rangle \\ & \wedge \mathtt{S}{=}\{\mathtt{v}\} \sqcup \mathtt{S}_1 \wedge (\forall \mathtt{x} \in \mathtt{S}_1 \cdot \mathtt{v} \leq \mathtt{x}))\end{aligned}$$

where `Node2` is declared as:

```
class Node2 { int val; Node2 prev; Node2 next }
```

The `dlls` shape predicate captures a chain of nodes that are to be traversed via the `next` field (starting from the current node `root`). The parameters `pr`, `bo` and `bi` represent the front output pointer (the `prev` field of the first node of the doubly linked list), the back output pointer (the `next` field of the last node) and the back input pointer (the last node of the list) respectively.

3.3.2 Well-Formedness and Well-Foundedness

Our predicate-based specification language is a combination of separation logic and pure (heap-independent) logics. A tailored entailment prover SLEEK

(Nguyen et al., 2007) has been built to generate the entailment relation of formulae in the language. Given an antecedent Δ_1 and a consequent Δ_2, the entailment checks that

$$\Delta_1 \vdash \Delta_2 * \Delta_R$$

The entailment proves whether heap nodes in Δ_1 are sufficiently precise to cover all nodes in Δ_2, and whether the pure formulae in Δ_1 entails pure formula in Δ_2. During the proof, a frame Δ_R is computed. The key steps that may be used in such an entailment proof are 1) matching up heap node from the antecedent and the consequent, 2) unfolding a shape predicate in the antecedent, 3) folding against a shape predicates in the consequent.

In order to ensure the soundness and termination of such processes by avoiding problem stemming from the interaction between garbage and program logics, the shape predicates and specifications of programs are required to be $well-formed$.

To define this concept, we firstly need to clarify the accessible variables and the reachability of a heap constraint node from a variable:

Definition 1 (Accessible) *A variable in a predicate or specification is accessible if it is one of the current method parameter or* `root` *or* `res`.

Definition 2 (Reachability) *Given a heap formula* $\kappa = \mathtt{p}::\mathtt{c}\langle\vec{v}\rangle * \kappa_1$, *atomic heap* $\mathtt{p}::\mathtt{c}\langle\vec{v}\rangle$ *is* reachable *from a variable* `q` *if and only if the following recursively defined relation holds:*

$$\begin{aligned}\mathtt{reach}(\kappa, \mathtt{q}, \mathtt{p}::\mathtt{c}\langle\vec{v}\rangle) \;=_{df}\; & (\mathtt{p} = \mathtt{q}) \vee \\ & (\kappa_1 = \mathtt{q}::\mathtt{c}_{\mathtt{q}}\langle\ldots, \mathtt{r}, \ldots\rangle * \kappa_2 \wedge \mathtt{reach}(\kappa_2, \mathtt{r}, \mathtt{p}::\mathtt{c}\langle\vec{v}\rangle))\end{aligned}$$

On the basis of definitions of `Accessible` and `Reachability`, we define the well-formedness of formulae:

Definition 3 (Well-Formed Formulae) *A separation formula* Φ *is well-formed if*

- (`i`) *every class node and shape predicate in* Φ *is reachable from its accessible variables, and*
- (`ii`) Φ *is in a disjunctive normal form* $\bigvee \vec{\sigma}$*, and* σ *is a conjunctive state* $\exists \vec{v} \cdot \kappa \wedge \pi$*, where* κ *is for heap formula,* π *is for (i.e. heap-independent) formula.*

The *well-formed* condition is significant in the light of that all heap nodes of a heap formula must be reachable from accessible variables, which allows the entailment checking procedure to correctly match nodes from the consequence with nodes from the antecedent. It is also required that `root` can appear only in predicate definitions, `res` in procedure postconditions.

Another potential problem during the reasoning is that arbitrary recursive shape relation can lead to non-termination entailment checking. To avoid that problem, we propose to use only *well-founded* shape predicates in our framework:

Definition 4 (Well-Founded Predicate) *A shape predicate* $\mathtt{p}::\mathtt{c}\langle\vec{v}\rangle = \Phi$ *is* well-founded *if*

- *(i)* Φ *is a well-formed formula,*

- *(ii) the parameter* `root` *can only be bound to a class node and not a predicate,*
- *(iii) each disjunct in the disjunctive body of* Φ *can contain at most one class node and only* `root` *is allowed to be bound to that class node, and*
- *(iv) every predicate is reachable from* `root`*.*

The shape predicates given in the last section are all well-founded. In contrast, the following three shape definitions are *not* well-founded:

$$\texttt{foo}\langle\texttt{n}\rangle \equiv \texttt{root}{::}\texttt{foo}\langle\texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1$$

$$\texttt{goo}\langle\rangle \equiv \texttt{root}{::}\texttt{Node}\langle\_, \_\rangle{*}\texttt{q}{::}\texttt{goo}\langle\rangle$$

$$\texttt{hoo}\langle\rangle \equiv \texttt{root}{::}\texttt{Node}\langle\_, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{Node}\langle\_, \_\rangle$$

For `foo`, the `root` identifier is bound to a shape predicate. For `goo`, the heap node pointed by `q` is *not* reachable from variable `root` (thus it is even not well-formed). For `hoo`, an extra object node is bound to a non-`root` variable. The first example may cause non-termination of entailment proof. When we want to rearrange a heap part of `foo` to expose an object from it, we simply get another `foo` which requires another unfolding that leads to non-termination. The second example `goo` captures an unreachable (junk) heap that cannot be located by our entailment method. The last example `hoo` shows the syntactic restriction imposed to facilitate termination of proof of entailment checking, and can be easily overcome by introducing intermediate predicates.

A few more examples of well-founded shape predicates are given below:

$$\begin{aligned}\texttt{root::treep}\langle\texttt{p}\rangle \equiv\ & \texttt{root=null} \\ & \vee\ \texttt{root::Node3}\langle\_, \texttt{l}, \texttt{r}, \texttt{p}\rangle * \texttt{l::treep}\langle\texttt{root}\rangle * \texttt{r::treep}\langle\texttt{root}\rangle\end{aligned}$$

$$\begin{aligned}\texttt{root::avl}\langle\texttt{h}, \texttt{S}\rangle \equiv\ & \texttt{root=null} \wedge \texttt{h=0} \wedge \texttt{S=}\emptyset \\ & \vee\ \texttt{root::Node2}\langle\texttt{v}, \texttt{l}, \texttt{r}\rangle * \texttt{l::avl}\langle\texttt{h}_\texttt{l}, \texttt{S}_\texttt{l}\rangle * \texttt{r::avl}\langle\texttt{h}_\texttt{r}, \texttt{S}_\texttt{r}\rangle \\ & \wedge\ \texttt{h=1+max}(\texttt{h}_\texttt{l}, \texttt{h}_\texttt{r}) \wedge -1 \leq \texttt{h}_\texttt{l} - \texttt{h}_\texttt{r} \leq 1 \\ & \wedge\ \texttt{S=S}_\texttt{l} \sqcup \texttt{S}_\texttt{r} \wedge (\forall \texttt{x} \in \texttt{S}_\texttt{l} \cdot \texttt{x} \leq \texttt{v}) \wedge (\forall \texttt{x} \in \texttt{S}_\texttt{r} \cdot \texttt{v} < \texttt{x})\end{aligned}$$

where `Node3` is declared as:

```
class Node3 { int val; Node3 l; Node3 r; Node3 p }
```

and `treep` is a binary tree where each node has a pointer which points to its parent, and `avl` is a self-balancing binary search tree with near balanced height.

3.3.3 Precondition and Postcondition

To verify certain properties of methods, users are allowed to annotate each method with preconditions and postconditions in our language to specify the behaviours of the method. A precondition is an assertion that should be satisfied when a method is called, thus the method body can assume it when the method starts. A postcondition is an assertion that should be established when the method exits, thus the caller can assume it after the call if the method is successfully verified. According to separation logic semantics, a

precondition further guarantees the existence of all memory locations that the method accesses, and hence guarantees free of memory errors during execution.

```
1  class Node { int val; Node next; }
2  Node insert_sort(Node x)
3  requires  x::llB⟨S⟩ ∧ |S|≥1
4  ensures   res::sllB⟨T⟩ ∧ S=T  {
5    if (x.next == null) return x;
6    else { Node s = x.next;
7      Node r = insert_sort(s);
8      return insert(r, x);
9    }
10 }
11 Node insert(Node r, Node x)
12 requires  r::sllB⟨S⟩*x::Node⟨v, _⟩
13 ensures   res::sllB⟨T⟩ ∧ T=S⊔{v}  {
14   if (r == null) {
15     x.next = null; return x;
16   } else if (x.val <= r.val) {
17     x.next = r; return x;
18   } else {
19     r.next = insert(r.next, x);
20     return r;
21   }
22 }
```

Figure 3.5: Insertion sort for singly linked list.

For example, using the `llB` and `sllB` predicates, we can specify insertion sort algorithm that operates on linked lists. The algorithm recursively sorts the tail of the input list, and inserts the first element into a sorted list so that the order is maintained. Its code is in Figure 3.5.

From the code, we can see the `insert_sort` method sorts a singly linked list. As its precondition $\texttt{x::llB}\langle\texttt{S}\rangle \wedge |\texttt{S}|{\geq}\texttt{1}$ suggests, the method takes in an unsorted list which starts from `x` with content `S`, and whose length should be at least one (this constraint is equivalent to $\texttt{x}{\neq}\texttt{null}$ and $\texttt{S}{\neq}\emptyset$, which, if the user specifies, can be captured by our entailment checker). Upon successful return, it gives a sorted list with the same content, as captured by the postcondition $\texttt{res::sllB}\langle\texttt{T}\rangle \wedge \texttt{S}{=}\texttt{T}$.

The method `insert` inserts an object pointed to by `x` into a sorted list referenced by `r`. The separation conjunction $*$ constrains the object `x` not to belong to the list `r`, thereby the resulting list has one more element. Meanwhile, the returned pointer `res` points to a sorted list whose content is the union of the two inputs as the postcondition indicates.

3.3.4 The Semantic Model

The semantics of our specification formulae is adapted from the "early versions" of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002), except that we have extensions to handle user-defined shape predicates and related

pure properties. We assume a number of sets: Loc of memory locations, Val of primitive values, with $0 \in \mathsf{Val}$ denoting `null`, Var of variables (program and logical variables), and ObjVal of object values stored in the heap, with $c[f_1 \mapsto \nu_1, \ldots, f_n \mapsto \nu_n]$ denoting an object value of class c where $\nu_1, \ldots, \nu_n$ are current values of the corresponding fields $f_1, \ldots, f_n$. Let $s, h \models \Delta$ denote the model relation, i.e. the stack s and heap h satisfy Δ, with h, s from the following concrete domains:

$$h \in \mathit{Heaps} \quad =_{df} \quad \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{ObjVal}$$

$$s \in \mathit{Stacks} \quad =_{df} \quad \mathsf{Var} \to \mathsf{Val} \cup \mathsf{Loc}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping as in the classical separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002). The detailed model definition is in Figure 3.6. Function $\mathsf{dom}(f)$ returns the domain of function f. Note that we use $\mapsto$ to denote mappings, not the points-to assertion in separation logic, which has been replaced by $\mathtt{p}::\mathtt{c}\langle\vec{\mathtt{v}}\rangle$ in our notation.

We use $h_1 \perp h_2$ to denote that the heaps h_1 and h_2 have disjoint domains, i.e. $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$, and $h_1 \cdot h_2$ to indicate the union of such heaps. The test $\mathit{IsObj}(c)$ returns `true` only if c is a data node and $\mathit{IsPred}(c)$ returns `true` only if c is a shape predicate. The definition for $s, h \models \mathtt{p}::\mathtt{c}\langle\vec{\mathtt{v}}\rangle$ is split into two cases: (1) `c` is a data node; (2) `c` is a shape predicate. In the first case, h has to be a singleton heap. In the second case, the shape predicate `c` may be inductively defined. For pure formulae π, as noted in the last line of Figure 3.6, their semantics are defined with a specific notation $\models_A$, which

$$
\begin{array}{lll}
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s,h \models \Phi_1 \texttt{ or } s,h \models \Phi_2 \\
s,h \models \exists \vec{v} \cdot \kappa \wedge \pi & \texttt{iff} & \exists \vec{\nu} \cdot s[\vec{v} \mapsto \vec{\nu}], h \models \kappa \texttt{ and } s[\vec{v} \mapsto \vec{\nu}] \models \pi \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot h_1 \perp h_2 \texttt{ and } h = h_1 \cdot h_2 \texttt{ and} \\
 & & s, h_1 \models \kappa_1 \texttt{ and } s, h_2 \models \kappa_2 \\
s,h \models \texttt{emp} & \texttt{iff} & \mathsf{dom}(h) = \emptyset \\
s,h \models \texttt{p::c}\langle \texttt{v}_1, \ldots, \texttt{v}_\texttt{n} \rangle & \texttt{iff} & \mathit{IsObj}(\texttt{c}) \texttt{ and } s(p) \in \mathsf{Loc} \texttt{ and } h = [s(p) \mapsto r] \\
 & & \texttt{and } r = \texttt{c}[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)] \\
 & \texttt{or} & \mathit{IsPred}(\texttt{c}) \texttt{ and } \texttt{c}\langle \texttt{v}_1, \ldots, \texttt{v}_\texttt{n} \rangle \equiv \Phi \\
 & & \texttt{and } s,h \models [p/\texttt{root}]\Phi \\
s \models \pi_1 \wedge \pi_2 & \texttt{iff} & s \models \pi_1 \texttt{ and } s \models \pi_2 \\
s \models \pi & \texttt{iff} & s \models_A \pi
\end{array}
$$

Figure 3.6: The semantic model.

is preserved by the pure constraint provers that we use for soundness purpose. Its definition is given in Figure 3.7. Note that $s(v)$ returns the value of the variable v that was the most recently popped one, and $s(\mathtt{B})$ returns the most recently popped bag of values of the variable $\mathtt{B}$.

$$
\begin{array}{lll}
s \models_A \gamma_1 \wedge \gamma_2 & \texttt{iff} & s \models_A \gamma_1 \texttt{ and } s \models_A \gamma_2 \\
s \models_A p_1 \bowtie p_2 & \texttt{iff} & s(p_1) \bowtie s(p_2), \texttt{ where } \bowtie \in \{=, \neq\} \\
s \models_A p \bowtie \texttt{null} & \texttt{iff} & s(p) \bowtie 0, \texttt{ where } \bowtie \in \{=, \neq\} \\
s \models_A \texttt{true} & \texttt{always} & \\
s \models_A \texttt{false} & \texttt{never} & \\
s \models_A v & \texttt{iff} & s(v) = \texttt{true} \\
s \models_A b_1 = b_2 & \texttt{iff} & s(b_1) = s(b_2) \\
s \models_A v_1 = v_2 & \texttt{iff} & s(v_1) = s(v_2) \\
s \models_A v_1 \leq v_2 & \texttt{iff} & s(v_1) \leq s(v_2) \\
s \models_A \phi_1 \wedge \phi_2 & \texttt{iff} & s \models_A \phi_1 \texttt{ and } s \models_A \phi_2 \\
s \models_A \phi_1 \vee \phi_2 & \texttt{iff} & s \models_A \phi_1 \texttt{ or } s \models_A \phi_2 \\
s \models_A \neg\phi & \texttt{iff} & s \models_A \phi \texttt{ does not hold} \\
s \models_A \exists v \cdot \phi & \texttt{iff} & s \models_A [k/v]\phi \texttt{ for some } k \\
s \models_A \forall v \cdot \phi & \texttt{iff} & s \models_A [k/v]\phi \texttt{ for all } k \\
s \models_A v \in \mathtt{B} & \texttt{iff} & s(v) \in s(\mathtt{B}) \\
s \models_A \mathtt{B}_1 = \mathtt{B}_2 & \texttt{iff} & s(\mathtt{B}_1) = s(\mathtt{B}_2) \\
s \models_A \mathtt{B}_1 \sqsubset \mathtt{B}_2 & \texttt{iff} & s(\mathtt{B}_1) \subset s(\mathtt{B}_2) \\
s \models_A \mathtt{B}_1 \sqsubseteq \mathtt{B}_2 & \texttt{iff} & s(\mathtt{B}_1) \subseteq s(\mathtt{B}_2) \\
s \models_A \forall v \in \mathtt{B} \cdot \phi & \texttt{iff} & s \models_A [k/v]\phi \texttt{ for all } k \in s(\mathtt{B}) \\
s \models_A \exists v \in \mathtt{B} \cdot \phi & \texttt{iff} & s \models_A [k/v]\phi \texttt{ for some } k \in s(\mathtt{B})
\end{array}
$$

Figure 3.7: The semantic model for pure constraints.

3.4 Summary

This chapter gives the definitions of the target programming language which is a fully typed object-based language, and the specification language which is based on the separation logic and combined with numerical and bag formulae to depict program contracts. For the purpose to prove the soundness of our analysis, we introduce the operational semantics of the programming language and the semantic model for the specification language. Meanwhile, we also illustrate the language settings with several examples.

Chapter 4

Loop Invariants Synthesis

Automated verification of memory safety and functional correctness for heap-manipulating programs has been a challenging task, especially when dealing with complex data structures with strong invariants involving both shape and numerical properties. Existing verification systems usually require users to supply loop invariants for loops to assist the verification, which can be cumbersome and error-prone by hand and can significantly restrict the scalability of the verification system. This chapter presents an automated loop invariants synthesis framework to reduce such demand. The loop invariant synthesis is conducted automatically by a fixed-point iteration process, equipped with newly designed abstraction, join and widening operators over an abstract domain with both shape and numerical information. We also prove the soundness and termination of our approach.

4.1 Introduction

As discussed previously, although research on software verification has a long and distinguished history (dating back to the 1960s), it remains a challenging problem to automatically verify heap manipulating programs written in mainstream imperative languages, such as C/C++ and Java. This is partly due to the shared mutable data structures lying in programs, and the need to track various program properties, such as structural numerical information (e.g. length and height) and relational numerical information (e.g. sortedness and binary search tree properties).

Since the emergence of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002), dramatic advances have been made in automated software verification, e.g. the Smallfoot tool (Berdine et al., 2005a) for the verification on pointer safety (that asserts pointers cannot go wrong), the verification on termination (Berdine et al., 2006), the verification for object-oriented programs (Chin et al., 2008; Parkinson and Bierman, 2008), and Dafny (Leino, 2010) and Hip/Sleek (Chin et al., 2007, 2010; Nguyen and Chin, 2008; Nguyen et al., 2007) for more general properties (both shape and numerical ones) for heap-manipulating programs.

As a key to prove the correctness of loops, *loop invariants* of every loop are required to be provided by users in these verification systems. However, supplying invariants by hand in the sophisticated domain is both cumbersome and error-prone. It also affects the scalability of these tools as a program

may contain many loops.

To conquer this problem, separation logic based shape analysis techniques are brought in. For example, the SpaceInvader tool (Calcagno et al., 2009; Distefano et al., 2006; Yang et al., 2008), as a further step of Smallfoot, can automatically infer loop invariants as well as method specifications for pointer safety properties in the shape domain. Other works such as THOR (Magill et al., 2008) incorporate simple numerical information into the shape domain to automatically synthesise properties like length of list. Their success proves the feasibility to generate loop invariant automatically for shape analysis to help automate the program verification process.

However, the prior loop invariant analyses mainly focus on relatively simple properties, such as pointer safety for lists and list length information. It is difficult to apply them in the presence of more sophisticated program properties, such as:

- More flexible user-defined data structures, e.g. trees;
- Relational numerical properties, like sortedness and binary search property.

These properties can be part of the full functional correctness of heap-manipulating programs. The (aforementioned) Hip/Sleek tool can handle such properties and it allows users to define their own shape predicates to

express their desired level of correctness.

In this chapter, we present a technique to automatically discover loop invariants over the combined shape and numerical domain to improve the level of automation for the HIP/SLEEK verification system. Our approach is based on the framework of abstract interpretation (Cousot and Cousot, 1977a) with fixed-point computation. It makes the following contributions in summary:

- A loop invariant synthesis algorithm is proposed with novel operations for abstraction, join and widening over a combined shape and numerical domain.
- The soundness of the analysis is proven with respect to concrete program semantics, and the termination of the analysis is also proven.

The rest of the chapter is structured as follows. We firstly illustrate our approach informally via a motivating example (Section 4.2), and then discuss the utility of the entailment checker in our analysis (Section 4.3). Formal details about loop invariant synthesis are presented in Section 4.4. More related works and concluding remarks come afterwards.

4.2 The Approach

In this section, we use a motivating example to informally illustrate our analysis approach. The example is an insertion sort algorithm for singly linked list. As introduced in chapter 3, the programming language is a strongly-typed object-based language, and the specification language is based on separation logic to describe heap/shape property with numerical information to specify related pure properties. The only difference here is that we do not require any user annotations for while loops in our language since we are going to infer it.

The class type `Node` used in our example is declared as

$$\texttt{class Node } \{ \texttt{ int val; Node next; } \}$$

As we described formerly, our specification language allows user-defined inductive predicates to specify aspects of both shape and numerical properties about the data structures in the programs that the user is interested in. Based on the class type `Node`, in order to model singly linked list data structure, we assume user-defined predicates `ll` for singly linked list and `lls` for singly linked list segment as follows:

$$\texttt{ll}\langle\texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge\texttt{n}{=}0)\vee(\texttt{root::node}\langle\texttt{v},\texttt{q}\rangle{*}\texttt{q::ll}\langle\texttt{m}\rangle\wedge\texttt{n}{=}\texttt{m}{+}1)$$

$$\texttt{ls}\langle\texttt{p},\texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{p}\wedge\texttt{n}{=}0)\vee(\texttt{root::node}\langle\texttt{v},\texttt{q}\rangle{*}\texttt{q::ls}\langle\texttt{p},\texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

If users want to verify a sorting algorithm, they can incorporate sortedness

property into the above predicates as follows:

$$\texttt{sll}\langle \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root::node}\langle \texttt{mn}, \texttt{null}\rangle \wedge \texttt{n}{=}1 \wedge \texttt{mn}{=}\texttt{mx}) \vee$$
$$(\texttt{root::node}\langle \texttt{mn}, \texttt{q}\rangle * \texttt{q::sll}\langle \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{mn}{\leq}\texttt{k} \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

$$\texttt{sls}\langle \texttt{p}, \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root::node}\langle \texttt{mn}, \texttt{p}\rangle \wedge \texttt{n}{=}1 \wedge \texttt{mn}{=}\texttt{mx}) \vee$$
$$(\texttt{root::node}\langle \texttt{mn}, \texttt{q}\rangle * \texttt{q::sls}\langle \texttt{p}, \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{mn}{\leq}\texttt{k} \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

where `mn` and `mx` denote the minimum and maximum values stored in the sorted list respectively. Such predicates can be used for specifying method specifications and for expressing loop invariants in the example.

4.2.1 Illustrative Example: Insertion Sort

We now illustrate our loop invariant synthesis process via an example. The method `ins_sort` (Figure 4.1) sorts a linked list with the insertion sort algorithm. It is implemented with two nested while loops. The outer loop traverses the whole list `x`, takes out each node from it (line 7), and inserts that node into another already sorted list `r` (which is empty initially before the sorting). This insertion process makes use of the inner while loop in lines 9-11 to look for a proper position in the already sorted list for the new node to be inserted. The actual insertion takes place at lines 12-14.

To verify this program, we need to synthesise appropriate loop invariants for both while loops. Our analysis follows a standard fixpoint iteration process. It starts with the (abstract) program state immediately before the while loop (i.e., the initial state) and symbolically executes the loop body for several

```
1 class Node { int val; Node next; }
2 node ins_sort(node x)
3 requires  x::ll⟨n⟩
4 ensures   res::sll⟨n,mn,mx⟩
5 { int v; node r,cur,srt,prv=null;
6   while (x != null) {
7     cur=x; x=x.next; v=cur.val;
8     srt=r; prv=null;
9     while (srt != null && srt.val <= v) {
10      prv=srt; srt=srt.next;
11    }
12    cur.next=srt;
13    if (prv != null) prv.next=cur;
14    else r=cur;
15  }
16  return r;
17 }
```

Figure 4.1: Loop-based Insertion sort.

iterations, until the obtained states converge to a fixpoint, which is the loop invariant.[1] At the start of each iteration, the obtained state from the previous iteration is joined with the initial state. In addition to this join operator, we have also defined an abstraction function and a widening operator both of which will help the fixpoint iteration to converge. The join and widening operators are specifically designed to handle both shape and numerical information.

As for our example, due to the presence of nested loops, each iteration of the analysis for the outer loop actually infers a loop invariant for the inner loop. We shall now illustrate how we synthesise a loop invariant for the inner loop.

Supposing that in one iteration for the outer loop, the state at line 9 becomes

$$\begin{aligned}&\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle * \mathtt{cur}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \\ &\wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n}\end{aligned}$$

Note that since the inner loop does not mutate the heap part referred to by `cur` and `x` (i.e., $\mathtt{cur}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle$), we can ignore it during the invariant synthesis and add it back to the program state using the frame rule of separation logic (Reynolds, 2002). Therefore, the initial state for loop invariant synthesis becomes

$$\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n} \tag{4.1}$$

[1] The fixpoint iteration converges if one more iteration still yields the same result.

From this state, symbolically executing the loop body once yields the state:

$$\begin{aligned}&\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \wedge \\ &\quad \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r}\end{aligned} \tag{4.2}$$

which says that pointer `srt` moves towards the tail of the list for one node. Then we join it with the initial state (4.1) to obtain

$$\begin{aligned}&(\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n}) \vee \\ &(\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \\ &\qquad \mathtt{prv}{=}\mathtt{r} \wedge \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r})\end{aligned} \tag{4.3}$$

The second iteration over the loop body starts with (4.3) and also exhibits the case that `srt` runs two nodes towards tail, while `prv` goes one node. Its result is then joined with pre-state (4.1) to become the current state:

$$\begin{aligned}&(4.3) \vee \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c_1}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_2}, \mathtt{b}\rangle \wedge \\ &\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2} \wedge \mathtt{c_1}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}2{=}\mathtt{n_r}\end{aligned} \tag{4.4}$$

Executing the loop body the third time returns a post-state where three nodes are passed by `srt`, and two by `prv` as below:

$$\begin{aligned}&(4.4) \vee \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{r_0}\rangle * \mathtt{r_0}{::}\mathtt{node}\langle \mathtt{c_1}, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \\ &\quad \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2}{\leq}\mathtt{c_3} \wedge \mathtt{c_2}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}3{=}\mathtt{n_r}\end{aligned}$$

where we have an auxiliary logical variable $\mathtt{r_0}$. Following this trend, it is predictable that every iteration hereafter will introduce an additional logical variable (referring to a list node). If we indulge in such increase in the subsequent iterations, the analysis will never terminate. Our abstraction process prevents this from happening by eliminating such logical variables as follows:

$$\begin{aligned}&(4.4) \vee \mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \\ &\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2}{\leq}\mathtt{c_3} \wedge \mathtt{c_2}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}3{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}2\end{aligned}$$

Note that the heap part $\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{r}_0\rangle * \mathtt{r}_0{::}\mathtt{node}\langle \mathtt{c}_1, \mathtt{prv}\rangle$ is abstracted as a sorted list segment $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n}_1, \mathtt{a}, \mathtt{c}_1\rangle$ where $\mathtt{n}_1$ denotes the length of the segment and $\mathtt{n}_1{=}2$ is added into the state. This abstraction process ensures that our analysis does not allow the shape to increase infinitely.

The fourth iteration responds with a post-state where four nodes are passed by `srt`, and three by `prv`. Therefore, an abstraction is performed to remove the logical pointer variables. As a simplification of the presentation, we denote σ as $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv},\mathtt{n}_1,\mathtt{a},\mathtt{c}_1\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_2,\mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_s,\mathtt{c}_3,\mathtt{b}\rangle \wedge \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2{\leq}\mathtt{c}_3 \wedge \mathtt{c}_2{\leq}\mathtt{v} \wedge \mathtt{n}_r{+}1{=}\mathtt{n}{-}\mathtt{n}_x$, and the abstracted result (after the fourth iteration) is

$$(4.4) \vee (\sigma \wedge \mathtt{n}_s{+}3{=}\mathtt{n}_r \wedge \mathtt{n}_1{=}2) \vee (\sigma \wedge \mathtt{n}_s{+}4{=}\mathtt{n}_r \wedge \mathtt{n}_1{=}3)$$

from which we have an observation that the last two disjunctions share the same shape part (as in σ). This disjunction will be transferred to the numerical domain as follows:

$$(4.4) \vee (\sigma \wedge (\mathtt{n}_s{+}3{=}\mathtt{n}_r \wedge \mathtt{n}_1{=}2 \vee \mathtt{n}_s{+}4{=}\mathtt{n}_r \wedge \mathtt{n}_1{=}3))$$

This simplifies the abstraction further. After that, our widening operation compares the current state with the previous one, in order to look for the same (numerical) constraints that both states imply and to replace those numerical constraints in the current state with the ones discovered by widening. This operation eventually ensures termination of our analysis. As for the example, some constraints among $\mathtt{n}_s$, $\mathtt{n}_r$ and $\mathtt{n}_1$ can be found to make the widened post-state become:

$$(4.4) \vee (\sigma \wedge \mathtt{n}_s{+}\mathtt{n}_1{=}\mathtt{n}_r{-}1 \wedge \mathtt{n}_1{\geq}2) \tag{4.5}$$

One more iteration of symbolic execution will produce the same result as (4.5), suggesting that it is already the fixpoint (and hence the loop invariant):

$$\begin{array}{l}
\mathtt{r}::\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \vee \\
\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle{*}\mathtt{srt}::\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r} \vee \\
\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle{*}\mathtt{prv}::\mathtt{node}\langle \mathtt{c_1}, \mathtt{srt}\rangle{*}\mathtt{srt}::\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_2}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2} \wedge \mathtt{c_1}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}2{=}\mathtt{n_r} \vee \\
\mathtt{r}::\mathtt{sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle{*}\mathtt{prv}::\mathtt{node}\langle \mathtt{c_2}, \mathtt{srt}\rangle{*}\mathtt{srt}::\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2}{\leq}\mathtt{c_3} \wedge \mathtt{c_2}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}\mathtt{n_1}{=}\mathtt{n_r}{-}1 \wedge \mathtt{n_1}{\geq}2
\end{array}$$

Note that although it is possible to further join the third disjunctive branch with the fourth, our analysis does not do so as it tries to keep the result as precise as possible by eliminating only auxiliary pointer variables.

With the frame part $\mathtt{cur}::\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle{*}\mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle$ added back, the analysis for the outer loop continues. Eventually, the following loop invariant is discovered for the outer loop:

$$\begin{array}{l}
(\mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{n_x}{=}\mathtt{n}) \vee (\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{null}\rangle{*}\mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}1) \vee \\
\quad (\mathtt{r}::\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle{*}\mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}\mathtt{n_r} \wedge \mathtt{n_r}{\geq}2)
\end{array}$$

which allows us to verify the whole method successfully by using automated verification tools, such as Hip/Sleek.

4.3 Entailment Checker

As we introduced in subsection 3.3.2, we use the separation logic prover SLEEK (Nguyen et al., 2007) to prove whether one abstract state Δ_1 entails another one Δ_2: $\Delta_1 \vdash \Delta_2 * \Delta_R$. Along with the proof, SLEEK also computes the residue part Δ_R (a.k.a the frame) which is useful for our inference framework. To prove the heap entailment is to check whether heap nodes in the antecedent Δ_1 are sufficiently precise to cover all nodes from the consequent Δ_2. The entailment checking procedure uses unfold/fold reasoning to deal with user-defined shape predicates with sophisticated numerical properties. During the entailment proof, the frame Δ_R is generated and it contains the nodes which are not consumed from the antecedent after matching up with the formula from the consequent and numerical constraints which convey the relationship between the variables in the antecedent and consequent formulae. For instance, by entailment proof

$$\exists \mathtt{y} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{vx}, \mathtt{y}\rangle * \mathtt{y}{::}\mathtt{ll}\langle \mathtt{n}\rangle \vdash \mathtt{x}{::}\mathtt{ll}\langle \mathtt{m}\rangle * \Delta_R$$

we can generate the residue Δ_R as $\mathtt{m} = \mathtt{n}{+}1$, which says that $\mathtt{x}$ is a list of length $\mathtt{n}{+}1$. Meanwhile, if we try to prove

$$\exists \mathtt{y} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{vx}, \mathtt{y}\rangle * \mathtt{y}{::}\mathtt{node}\langle \mathtt{vy}, \mathtt{z}\rangle \wedge \mathtt{vx} \leq \mathtt{vy} \vdash \mathtt{x}{::}\mathtt{sls}\langle \mathtt{n}, \mathtt{z}, \mathtt{mn}, \mathtt{mx}\rangle * \Delta_R$$

the residue Δ_R can be generated as $\mathtt{n}{=}2 \wedge \mathtt{mn}{=}\mathtt{vx} \wedge \mathtt{mx}{=}\mathtt{vy} \wedge \mathtt{mn} \leq \mathtt{mx}$, which shows that the length of the sorted list from $\mathtt{x}$ to $\mathtt{z}$ is 2, and the minimal value of the list is $\mathtt{vx}$ in node $\mathtt{x}$ and the maximal value is $\mathtt{vy}$ in node $\mathtt{y}$. From above examples, we can see that the SLEEK entailment prover can be used to eliminate quantified pointer variables, to generate more abstract shape views, and to preserve useful numerical information.

Based on the entailment relation, we define a partial order over the abstract states:

$$\Delta_2 \preceq \Delta_1 =_{df} \Delta_1 \vdash \Delta_2 * \Delta_R \qquad \text{for some } \Delta_R$$

We also denote this by $\Delta_1 \succeq \Delta_2$. Based on this partial order, we also have an induced lattice over these states as the base of fixpoint calculation for loop invariants.

4.4 Analysis Algorithm

Our proposed analysis algorithm is given in Figure 4.2. The algorithm takes four input parameters: $\mathcal{T}$ as the program environment with all the method specifications in the program, Δ_{pre} as the precondition for the while loop (i.e. the abstract state that is before the loop starts), the while loop itself `while` b $\{e\}$, and the upper bound n on the number of shared logical variables we keep during the analysis.

Our analysis is based on abstract interpretation (Cousot and Cousot, 1977a) with specifically designed operations (abs, join and widen) over this combined domain.[2] At the beginning, we initialise the iteration variable (i) and two states to begin with (Δ_i and Δ_i'). The starting state of the calculation is Δ_{pre}. Among the two states here, the unprimed version Δ_i denotes the

[2]Note that our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details in Section 4.4.2.

Fixpoint Computation in Combined Domain

Input: $\mathcal{T},\ \Delta_{pre},\ \texttt{while}\ b\ \{e\},\ n\,;$

Local: $i := 0;\ \Delta_i := \Delta_{pre};\ \Delta'_i := \Delta_i;$

1 **repeat**

2 $\quad i := i + 1;$

3 $\quad \Delta_i := \mathsf{widen}^{\dagger}(\Delta_{i-1}, \mathsf{join}^{\dagger}(\Delta_{i-1}, \Delta'_{i-1}));$

4 $\quad \Delta'_i := \mathsf{abs}^{\dagger}([\![e]\!]_{\mathcal{T}}(\Delta_i \wedge b));$

5 $\quad$ **if** $\Delta'_i = \texttt{false} \vee \mathsf{cp\_no}(\Delta'_i) > n$
$\quad$ **then return** fail **end if**

6 **until** $\Delta'_i = \Delta'_{i-1};$

7 **return** Δ'_i

Figure 4.2: Loop invariant synthesis main analysis algorithm .

initial state before the i^{th} execution of the loop body, and the primed one Δ'_i represents the result state after. Each iteration starts at line 1. Firstly, we join together the initial state Δ_{i-1} of the previous iteration with the result state Δ'_{i-1} obtained in the previous iteration, and widen it against the state Δ_{i-1} (line 3). Then we symbolically execute the loop body with the abstract semantics in Section 4.4.1 (line 4), and apply the abstraction operation to the obtained abstract state. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 5). Otherwise we judge whether a fixpoint is already reached by comparing the current abstract state with the previous one (line 6). The fixpoint Δ'_i is returned as the loop invariant.

We will elaborate the key techniques of our analysis: the abstract semantics, the abstraction function, and the join and widening operators.

4.4.1 Abstract Semantics

The abstract semantics is used to execute the loop body symbolically to obtain its post-state during the loop invariant synthesis. Its type is defined as

$$[\![e]\!] \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains all the specifications of all methods (extracted from the program *Prog*). For some expression e, given its precondition, the se-

mantics returns the postcondition.

The foundation of the semantics is the basic transition functions which transfer from a conjunctive abstract state to a conjunctive or disjunctive abstract state as below:

$$\begin{array}{lll} \mathsf{unfold}(x) & : \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & \text{Rearrangement} \\ \mathsf{exec}(d[x]) & : \mathsf{AllSpec} \rightarrow \mathsf{SH}[x] \rightarrow \mathsf{SH} & \text{Heap-sensitive execution} \\ \mathsf{exec}(d) & : \mathsf{AllSpec} \rightarrow \mathsf{SH} \rightarrow \mathsf{SH} & \text{Heap-insensitive execution} \end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of a data node ($\mathtt{x}::\mathtt{c}\langle\mathtt{v}^*\rangle$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{unfold}(x)$ rearranges the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

$$\frac{\mathit{IsObj}(c) \quad \sigma \vdash \mathtt{x}::\mathtt{c}\langle\mathtt{v}^*\rangle * \sigma'}{\mathsf{unfold}(x)\sigma =_{df} \sigma}$$

$$\frac{\mathit{IsPred}(c) \quad \sigma \vdash \mathtt{x}::\mathtt{c}\langle\mathtt{u}^*\rangle * \sigma' \quad \mathtt{root}::\mathtt{c}\langle\mathtt{v}^*\rangle \equiv \Phi}{\mathsf{unfold}(x)\sigma =_{df} \sigma' * [x/\mathtt{root}, u^*/v^*]\Phi}$$

As mentioned earlier, the test $\mathit{IsObj}(c)$ returns `true` only if c is a data node and $\mathit{IsPred}(c)$ returns `true` only if c is a shape predicate.

The symbolic execution of heap-sensitive commands $d[x]$ (i.e. $x.f_i$, $x.f_i := w$, or $\mathtt{free}(x)$) assumes that the rearrangement $\mathsf{unfold}(x)$ has been done previ-

ously:

$$\frac{IsObj(c) \quad \sigma \vdash \mathtt{x}::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma =_{df} \sigma' * \mathtt{x}::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle \wedge \mathtt{res}{=}v_i}$$

$$\frac{IsObj(c) \quad \sigma \vdash \mathtt{x}::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_n}\rangle * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})\sigma =_{df} \sigma' * \mathtt{x}::\mathtt{c}\langle \mathtt{v_1}, .., \mathtt{v_{i-1}}, \mathtt{w}, \mathtt{v_{i+1}}, .., \mathtt{v_n}\rangle}$$

$$\frac{IsObj(c) \quad \sigma \vdash \mathtt{x}::\mathtt{c}\langle \mathtt{u}^*\rangle * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma =_{df} \sigma'}$$

The symbolic execution rules for heap-insensitive commands are shown as follows:

$$\mathsf{exec}(k)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}k$$

$$\mathsf{exec}(x)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}x$$

$$\frac{IsObj(c)}{\mathsf{exec}(\mathsf{new}\ c(v^*))(\mathcal{T})\sigma =_{df} \sigma * \mathtt{res}::\mathtt{c}\langle \mathtt{v}^*\rangle}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t'_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^m \circ [y'_i/v_i]_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x'_i/u'_i]_{i=1}^m \circ [y'_i/v'_i]_{i=1}^n \quad \rho_l = [r_i/y'_i]_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{exec}(mn(x_1, .., x_m; y_1, .., y_n))(\mathcal{T})\sigma =_{df} (\rho_l\sigma') * (\rho_o\Phi_{po})}$$

Note that the first three rules deal with constant (k)[3], variable (x) and data node creation (new $c(v^*)$) respectively, while the last rule handles method invocation. In the last rule, the call site is ensured to meet the precondition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and the post-state of the method call involves mn's postcondition as signified by $\rho_o\Phi_{po}$. The substitution $\rho_2 \circ \rho_1$ works by first applying ρ_1 and then ρ_2.

A lifting function † is defined to lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{unfold}^{\dagger}(x) \bigvee \sigma_i =_{df} \bigvee(\mathsf{unfold}(x)\sigma_i)$$

and this function is overloaded for exec to lift both its domain and range to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T}) \bigvee \sigma_i =_{df} \bigvee(\mathsf{exec}(d)(\mathcal{T})\sigma_i)$$

Based on the transition functions above, we can define the abstract semantics for a program command e as follows:

$$\begin{array}{lll}
[\![d[x]]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{unfold}^{\dagger}(x)\Delta \\
[\![d]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d)(\mathcal{T})\Delta \\
[\![e_1; e_2]\!]_{\mathcal{T}}\Delta & =_{df} & [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}\Delta \\
[\![x := e]\!]_{\mathcal{T}}\Delta & =_{df} & [x'/x, r'/\mathtt{res}]([\![e]\!]_{\mathcal{T}}\Delta) \wedge x{=}r' \quad \textit{fresh logical } x', r' \\
[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}\Delta & =_{df} & ([\![e_1]\!]_{\mathcal{T}}(v \wedge \Delta)) \vee ([\![e_2]\!]_{\mathcal{T}}(\neg v \wedge \Delta))
\end{array}$$

which forms the foundation for us to analyse the loop body.

[3] `null` is treated as a constant.

4.4.2 Abstraction, Join and Widening

This section describes our specifically designed abstraction, join and widening operations employed in our loop invariant synthesis process.

Abstraction operator. During the symbolic execution, we may be confronted with many "concrete" shapes in postconditions of the loop body. As an example of list traversal, the list may contain one node, two nodes, or even more nodes in the list, which the analysis cannot enumerate infinitely. The abstraction operator deals with those situations by abstracting the (potentially infinite) concrete situations into more abstract shapes. Our rationale is to keep only program variables and shared cutpoints; all other logical variables will be abstracted away. As an instance, the first state below can be further abstracted (as shown), while the second one cannot:

$$\begin{array}{l} \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle\_, \mathtt{null}\rangle) \;=\; \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{=}2 \\ \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{y}{::}\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle\_, \mathtt{null}\rangle) \;=\; \text{-} \end{array} \tag{4.6}$$

where both $\mathtt{x}$ and $\mathtt{y}$ are program variables, and $\mathtt{z}_0$ is an existentially quantified logical variable. In the second case $\mathtt{z}_0$ is a shared cutpoint referenced by both $\mathtt{x}$ and $\mathtt{y}$, and thus the state is not changed. As illustrated above, the abstraction transition operator abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

$$\mathsf{abs} \;:\; \mathsf{SH} \rightarrow \mathsf{SH} \qquad \text{Abstraction}$$

which indicates that it takes in a conjunctive abstract state σ and abstracts it as another conjunctive state σ'. Below are its rules.

$$\frac{}{\mathsf{abs}(\sigma \wedge x_0{=}e) =_{df} \sigma[e/x_0]}\ (\mathsf{Subst1})$$

$$\frac{}{\mathsf{abs}(\sigma \wedge e{=}x_0) =_{df} \sigma[e/x_0]}\ (\mathsf{Subst2})$$

$$\frac{x_0 \notin \mathsf{Reach}(\sigma)}{\mathsf{abs}(\mathtt{x_0}{::}\mathtt{c}\langle\vec{\mathtt{v}}\rangle{*}\sigma) =_{df} \sigma{*}\mathtt{true}}\ (\mathsf{Unreach})$$

$$\frac{\begin{array}{c}\mathtt{c_2}\langle\vec{\mathtt{u_2}}\rangle \equiv \Phi \qquad \mathtt{p}{::}\mathtt{c_1}\langle\vec{\mathtt{v_1}}\rangle{*}\sigma_1 \vdash \mathtt{p}{::}\mathtt{c_2}\langle\vec{\mathtt{v_2}}\rangle{*}\sigma_2 \\ \mathsf{Reach}(\mathtt{p}{::}\mathtt{c_2}\langle\vec{\mathtt{v_2}}\rangle{*}\sigma_2) \cap \{\vec{v_1}\} = \emptyset\end{array}}{\mathsf{abs}(\mathtt{p}{::}\mathtt{c_1}\langle\vec{\mathtt{v_1}}\rangle{*}\sigma_1) =_{df} \mathtt{p}{::}\mathtt{c_2}\langle\vec{\mathtt{v_2}}\rangle{*}\sigma_2}\ (\mathsf{Abs})$$

The first two rules Sub1 and Sub2 eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule Unreach is used to eliminate any garbage (heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. As x_0 is already unreachable from and unusable by the program variables, it is safe to treat it as garbage `true`, for example, the $\mathtt{x_0}$ in $\mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{null}\rangle{*}\mathtt{x_0}{::}\mathtt{node}\langle\_, \mathtt{null}\rangle$ where only `x` is a program variable.

The last rule Abs plays the most significant role which intends to eliminate shape formulae led by logical variables (all variables in $\vec{\mathtt{v_1}}$). It tries to fold data nodes up to a predicate node. $\mathtt{c_2}$ is a predicate which is selected from the set of user-defined predicates based on the data type of `p`. We may have multiple predicates that are satisfied this rule, so that "abs" is relational, rather than functional. However, we only choose one satisfied predicate during one analysis, if this predicate is not sufficient to generate

adequate loop invariant, we will try another one. Meanwhile, it also ensures that the latter is a sound abstraction of the former by entailment proof, and the logical parameters of $\mathtt{c_1}$ are not reachable from other part of the heap (so that the abstraction does not lose necessary information). For example: $\mathsf{abs}(\mathtt{x{::}node}\langle\_, \mathtt{z_0}\rangle * \mathtt{z_0{::}node}\langle\_, \mathtt{null}\rangle) = \mathtt{x{::}ll}\langle\mathtt{n}\rangle \wedge \mathtt{n{=}2}$, where x is a program variable and z_0 is a logical variable. The function Reach is defined as follows:

$$\mathsf{Reach}(\sigma) =_{df} \bigcup_{v \in \mathtt{fv}(\sigma)} \mathtt{ReachVar}(\kappa \wedge \pi, v) \ \mathtt{where}\ \sigma ::= \exists \mathtt{u}^* \cdot \kappa \wedge \pi$$

which returns all variables which are reachable from free variables in the abstract state σ. The function $\mathtt{ReachVar}(\kappa \wedge \pi, v)$ returns the minimal set of variables that satisfies the relationship below:

$$\{v\} \cup \{z_2 \mid \exists z_1, \pi_1 \cdot z_1 \in \mathtt{ReachVar}(\kappa \wedge \pi, v) \wedge \pi = (z_1 = z_2 \wedge \pi_1)\} \cup \{z_2 \mid$$
$$\exists z_1, \kappa_1 \cdot z_1 \in \mathtt{ReachVar}(\kappa \wedge \pi, v) \wedge \kappa = (\mathtt{z_1{::}c}\langle .., \mathtt{z_2}, .. \rangle * \kappa_1)\} \subseteq \mathtt{ReachVar}(\kappa \wedge \pi, v)$$

viz. it is composed of aliases of v and variables reachable from v. For example, $\mathtt{ReachVar}(\mathtt{x{::}node}\langle\_, \mathtt{z_0}\rangle * \mathtt{y{::}node}\langle\_, \mathtt{z_0}\rangle * \mathtt{z_0{::}node}\langle\_, \mathtt{p_0}\rangle, \{\mathtt{x}\}) = \{\mathtt{x}, \mathtt{z_0}, \mathtt{p_0}\}$.

We apply the above abstraction rules (following the given order) onto an abstract state exhaustively until it stabilises. Such convergence is confirmed because the abstract shape domain is finite due to the bounded numbers of variables and predicates, which will be discussed later.

Finally, the lifting function is overloaded for abs to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{abs}^{\dagger}(\bigvee \sigma_i) =_{df} \bigvee \mathsf{abs}(\sigma_i)$$

which allows it to be used in the analysis.

Join operator. The operator join is applied over two conjunctive abstract states, trying to find a common shape as a sound abstraction for both:

$$
\begin{array}{l}
\mathsf{join}(\sigma_1, \sigma_2) =_{df} \\
\quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\
\quad\quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists \vec{x_1} \cdot \kappa_1 \wedge \pi_1), (\exists \vec{x_2} \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\
\quad\quad\quad \textbf{if } \kappa_1 \vdash \kappa_2 * \mathtt{true} \textbf{ then } \exists \vec{x_1}, \vec{x_2} \cdot \kappa_2 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\
\quad\quad\quad \textbf{else if } \kappa_2 \vdash \kappa_1 * \mathtt{true} \textbf{ then } \exists \vec{x_1}, \vec{x_2} \cdot \kappa_1 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\
\quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2
\end{array}
$$

where the rename function prevents naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example, it will reassign $\mathtt{x_0}$'s name in both states $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{0}$ and $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{1}$ to make them $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{0}$ and $\exists \mathtt{x_1} \cdot \mathtt{x_1}{=}\mathtt{1}$. After this procedure, it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape from the weaker state (which is more general/abstract) as the shape of the joined state, and performs joining for numerical formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$. The convex hull operator works over numerical domain (Popeea and Chin, 2006). Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case). Then we lift this operator for abstract state Δ as follows:

$$\mathsf{join}^\dagger(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\bigvee_i \sigma_i^1), (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j} \mathsf{join}(\sigma_i^1, \sigma_j^2)$$

which essentially joins all pairs of disjunctions from the two abstract states, and makes a disjunction of them.

Widening operator. The finiteness of the shape domain is confirmed by the abstraction function. To ensure the termination of the whole analysis,

we still need to guarantee the convergence over the numerical domain. This task is accomplished by the widening operator.

The widening operator $\mathsf{widen}(\sigma_1, \sigma_2)$ is defined as

$$
\begin{array}{l}
\mathsf{widen}(\sigma_1, \sigma_2) =_{df} \\
\quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\
\quad\quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists \vec{x_1} \cdot \kappa_1 \wedge \pi_1), (\exists \vec{x_2} \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\
\quad\quad\quad \textbf{if } \kappa_1 \vdash \kappa_2 * \texttt{true} \textbf{ then } \exists \vec{x_1}, \vec{x_2} \cdot \kappa_2 \wedge (\mathsf{widen}_{\pi}(\pi_1, \pi_2)) \\
\quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2
\end{array}
$$

where the rename function has the same effect as above. Generally, this operator is analogous to join; the only difference is that we expect the second operand σ_2 is weaker than the first σ_1, so that the widening reflects the trend of such weakening from σ_1 to σ_2. In this case, it applies the widening operation $\mathsf{widen}_{\pi}(\pi_1, \pi_2)$ over the numerical domain (Popeea and Chin, 2006). Therefore, based on the widening over conjunctive abstract states, we lift the operator over (disjunctive) abstract states:

$$\mathsf{widen}^{\dagger}(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\bigvee_i \sigma_i^1), (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j} \mathsf{widen}(\sigma_i^1, \sigma_j^2)$$

which is similar to its counterpart of the join operator.

The above three operations (abstraction, join and widening) provide termination guarantee while preserving soundness as the following example demonstrates.

Example 5 (Abstraction, join and widening) *Assuming we have two abstract states,* $\Delta_0 = \texttt{x::node}\langle \_, \texttt{x}_0 \rangle * \texttt{x}_0\texttt{::node}\langle \_, \texttt{null} \rangle$ *and* $\Delta_1 = \texttt{x::node}\langle \_, \texttt{x}_0 \rangle *$

$\mathtt{x}_0{::}\mathtt{node}\langle\_, \mathtt{x}_1\rangle * \mathtt{x}_1{::}\mathtt{node}\langle\_, \mathtt{null}\rangle$, *we would like to discover a sound approximation for both states. Firstly, we perform abstractions on both to obtain two abstract states, saying,* $\Delta'_0 = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}2$ *and* $\Delta'_1 = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}3$. *Then these two are joined together according to shape similarity to be* $\Delta''_1 = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge (\mathtt{n}_0{=}2 \vee \mathtt{n}_0{=}3)$, *which transfers disjunction to numerical domain. Finally, based on the first state* Δ'_0, *the joined state is widened to yield a state* $\mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0{\geq}2$. *It is a sound abstraction of both* Δ_0 *and* Δ_1, *and finishes the analysis with one more iteration.* □

4.4.3 Soundness and termination

Soundness

The soundness of our analysis relies on the underlying operational semantics of our programming language, which is a small-step semantics that consists of transitions of the form:

$$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

where s/s_1 and h/h_1 denote respectively the stacks and the heaps, and where e/e_1 denotes the program codes (an empty program code is denoted by -). The transitive closure of the above transition relation is denoted as $\hookrightarrow^*$.

Before proceeding to the soundness definition, as mentioned earlier, for program variables we have both unprimed version (for their initial values in

abstract states) and primed version (for their current values). We realise that the concrete program states should always be linked to the primed ones. Therefore, we have the following definition:

Definition 4.5 (Poststate) *Given an abstract state Δ, Post(Δ) captures the relation between primed variables of Δ. That is,*

$Post(\Delta) =_{df} \rho(\exists V \cdot \Delta)$, *where*
$V = \{v_1, \ldots, v_n\}$ *denotes all unprimed program variables in* Δ, *and*
$\rho = [v_1/v_1', \ldots, v_n/v_n']$.

For example, let $\Delta = \texttt{x}'::\texttt{node}\langle \texttt{v}', \texttt{y}' \rangle \wedge \texttt{v}'=\texttt{v} \wedge \texttt{y}'=\texttt{null}$, *we have*

$$Post(\Delta) = \texttt{x}::\texttt{node}\langle \texttt{v}, \texttt{y} \rangle \wedge \texttt{y}=\texttt{null}.$$

Then we define the soundness of our analysis as follows:

Definition 4.6 (Soundness) *Let Δ denote the loop invariant synthesised by our analysis for a while loop* `while` b $\{e\}$. *The analysis is sound if* $\forall$ $s, h \cdot s, h \models Post(\Delta)$ *and* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$ *(for some s', h'), we have* $s', h' \models Post(\Delta)$.

The crux to prove the soundness of our analysis is to ensure that soundness is preserved during each step of our analysis. That is to say, the abstract semantics, the abstraction of shapes, the join operation and the widening operation used in our analysis are all sound. From Lemma 7 and the later parts, we will see that all these can be reduced to the soundness of entailment proof/checking provided by Sleek .

Lemma 6 (Soundness of entailment proof) *For* Δ_1 *and* Δ_2*, if* $\Delta_1 \vdash \Delta_2$ *holds, then for all* $s, h \models \Delta_1$*, we have* $s, h \models \Delta_2$*.*

Proof. The soundness of the entailment proof is proven by Chin et al. (2010). □

Lemma 7 (Soundness of abstract semantics) *If* $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_1$*, then for all* s, h*, if* $s, h \models Post(\Delta)$ *and* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$*, then there always exists* Δ_0 *so that*

$$s_1, h_1 \models Post(\Delta_0) \qquad and \qquad [\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof. The proof is done by structural induction over program constructors and the details are left in the Appendix. □

Lemma 8 (Soundness of abs) *If* $\mathsf{abs}(\sigma) = \sigma'$*, then* $\sigma \succeq \sigma'$*.*

Proof. The soundness proof of first two substitution rules is trivial. For the Unreach rule, we can easily prove that $\mathtt{x_0}{::}\mathtt{c}\langle\vec{\mathtt{v}}\rangle * \sigma \vdash \mathtt{true} * \sigma$. σ is the frame part of the entailment check. By the Frame Rule of separation logic, we only need to show that $\mathtt{x_0}{::}\mathtt{c}\langle\vec{\mathtt{v}}\rangle \vdash true$, which obviously is true. The result of rule Abs is checked by SLEEK , the soundness of this rule hence is guaranteed by the entailment proof. □

Lemma 9 (Soundness of join) *If* $\mathsf{join}(\sigma_1, \sigma_2) = \sigma_j$*, then we have* $\sigma_1 \succeq \sigma_j$ *and* $\sigma_2 \succeq \sigma_j$*.*

Proof. Let σ_1 be $(\exists \vec{x_1} \cdot \kappa_1 \wedge \pi_1)$, and σ_2 be $(\exists \vec{x_2} \cdot \kappa_2 \wedge \pi_2)$. By the definition

of the join operator, we have three cases:

- If $\kappa_1 \vdash \kappa_2 * \texttt{true}$, we have $\sigma_j = \exists \vec{x_1}, \vec{x_2} \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$. Then we need to show that $\exists \vec{x_1} \cdot \kappa_1 \wedge \pi_1 \vdash \exists \vec{x_1}, \vec{x_2} \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$ and $\exists \vec{x_2} \cdot \kappa_2 \wedge \pi_2 \vdash \exists \vec{x_1}, \vec{x_2} \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$, which are true because $\kappa_1 \vdash \kappa_2 * \texttt{true}$ by condition, $\kappa_2 \vdash \kappa_2$ by separation logic, and $\pi_1 \vdash \mathsf{join}_\pi(\pi_1, \pi_2)$ and $\pi_2 \vdash \mathsf{join}_\pi(\pi_1, \pi_2)$ by soundness of convex hull operator over numerical domains (Popeea and Chin, 2006).
- If $\kappa_2 \vdash \kappa_1 * \texttt{true}$, the soundness proof is similar to the first case.
- Otherwise, we have $\sigma_j = \sigma_1 \vee \sigma_2$. The soundness proof of this case is trivial. □

Lemma 10 (Soundness of widen) *If* $\mathsf{widen}(\sigma_1, \sigma_2) = \sigma_w$, *then we have* $\sigma_1 \succeq \sigma_w$ *and* $\sigma_2 \succeq \sigma_w$.

Proof. The proof of soundness of widen is similar to the soundness proof of join. □

Based on the results above, we have the following theorem.

Theorem 11 (Soundness) *Our analysis is sound with respect to the underlying operational semantics.*

The soundness of our analysis ensures the generated formula is loop invariant

of the input loop, but it cannot guarantee the invariant is "logical adequate", namely, it is strong enough to prove the given postcondition of the methods. The logical adequacy of the synthesised invariant can be judged by comparing the invariant with the postcondition. For example, if the shape predicate of one program variable in the invariant has less information than the shape predicate of the same variable in the postcondition, the synthesised invariant may not be fit to prove the postcondition. During our analysis, we could try to always use the strongest predicates as the abstraction target to make the generated invariant be more adequate.

Termination

Next, we show the termination of our analysis algorithm, which is based on two observations: the finiteness over the shape domain and the termination over the numerical domain guaranteed by our widening operator. The first can be proven by claiming the finiteness of the number of all possible shape formulae. Recalling our analysis algorithm where we set an upper bound n for shared cutpoints (logical variables) that we keep track of, we know that the number of program variables and logical variables preserved in our analysis are finite. Note that the number of all shape predicates are also limited; thus all the shape formulae are finite. The second is proven in the abstract interpretation frameworks for numerical domains (Popeea and Chin, 2006). These two facts guarantee the convergence of our analysis.

Lemma 12 (Finiteness of the abstract shape domain) *With a finite number of program variables, logical variables, data node types and shape predi-*

cates, the abstract shape domain is finite.

Proof. Let the number of program variables, logical variables, data node types and shape predicates be m, n, l_1 and l_2 respectively, and the maximal number of fields of data nodes or arguments of the predicates be k, where $m, n, l_1, l_2, k \in \mathbb{N}$ are finite numbers. Note that the root parameter of a data structure is also counted in k. For example, the number of arguments of predicate ll is 3. Based on the fact that the number of variables is $(m+n)$, and k is the number of holes for variables in one single predicate, we then have the possible number of atomic formulae of this predicate as $(m+n)^k$. The number of shape structure is (l_1+l_2), then the upper bound of the number of all possible different atomic shape formulae is $(l_1+l_2) \times (m+n)^k + 1$, where 1 is for the case when shape formula is `true`. Thus there are at most $2^{(l_1+l_2)\times(m+n)^k+1}$ shape formulae in this shape domain. Since m, n, l_1, l_2 and k are finite natural numbers, the shape domain is finite. □

Definition 4.7 *A (conjunctive) state σ is reducible if and only if $\mathsf{abs}(\sigma) \nvdash \sigma$. If $\mathsf{abs}(\sigma) \vdash \sigma$, then σ is irreducible, in which case we also say σ is stabilised.*

Lemma 13 (Termination of abs) *For all state σ, the procedure of applying the four abstraction operations over σ exhaustively (following the given order) will terminate in finite steps within a finite shape domain.*

Proof. Let us apply abs over σ_0 exhaustively to obtain a sequence σ_1, σ_2, $\cdots$, σ_{n-1}, σ_n, where $n \in \mathbb{N}$. By the soundness of abs, we have $\sigma_0 \vdash \sigma_1, \sigma_1 \vdash \sigma_2, \cdots, \sigma_{n-1} \vdash \sigma_n$, i.e. $\sigma_0 \succeq \sigma_1 \succeq \sigma_2 \succeq \cdots \succeq \sigma_{n-1} \succeq \sigma_n$. Since the shape parts of $\sigma_{0,\cdots,n}$ are in a finite shape domain, and the four abstraction rules

do not alter the numerical parts of these states, there must exist a $\sigma_{i,0\leq i\leq n}$ which is *irreducible/stablised*, i.e., $\sigma_k = \sigma_i$ for all $k \geq i$. □

Lemma 14 (Widening Termination) *Within a finite shape domain, given a sequence σ'_n ($n \in \mathbb{N}$), the sequence σ_n generated by $\sigma_0 = \sigma'_0$ and $\sigma_{n+1} = \mathsf{widen}(\sigma_n, \sigma'_{n+1})$ is ultimately stationary, i.e. $\exists i \cdot \forall k \geq i \cdot \sigma_k = \sigma_i$.*

Proof. The proof follows the idea of the widening termination proof in cofibered domains (Venet, 1996). Similar to the proof of abs termination, the shape part of σ_n will be stationary since the shape domain is finite. The termination of numerical part can be guaranteed by numerical join and widening operations (Popeea and Chin, 2006). Combining them together, σ_n will be stationary. □

Based on the above discussions, we have the following conclusion.

Theorem 15 (Termination) *The iteration of our fixpoint computation will terminate in finite steps for finite input parameters, specifications, and user-defined predicates with a given finite upper bound for the number of logical variables.*

4.8 Related Work

Loop invariants are key component of program verification. Two most widespread frameworks for static invariant inference are abstract interpretation and the constraint-based approach. Constraint-based techniques (Colón et al., 2003; Sankaranarayanan et al., 2004; Kapur, 2005; Cousot, 2005a; Chen et al., 2007) rely on decision procedures over mathematical domains to represent concisely the semantics of loops with respect to certain template properties. The constraint-based approach has the advantage in finding invariants on sophisticated numerical domains (such as polynomials and convex polyhedra), but it lacks an effective method to choose the proper template of invariants.

Cousot and Cousot (1976) introduce abstract interpretation for discovering loop invariants, which symbolically executes programs over the interval domain, and calculates a fixed point as the loop invariants. Based on this work, a number of works have been proposed to work with other numerical abstract domains, such as convex polyhedra abstract domain (Cousot and Halbwachs, 1978), octagon abstract domain (Miné, 2001), polynomial equalities with bounded degree abstract domain (Rodríguez-Carbonell and Kapur, 2007), weighted hexagons abstract domain (Fulara et al., 2010), linear equalities abstract domain (Chen et al., 2010), and so on. These works mainly focus on finding numerical program properties. Compared with their works, our analysis is also founded on the abstract interpretation framework, but tries to discover loop invariants with *both* shape and numerical information. Meanwhile, we can also utilise their techniques of join and widening to reason about the numerical domain, as we make use of the work (Popeea and Chin,

2006) which works over a disjunctive convex polyhedra abstract domain that allows our analysis to keep a number of disjuncts to achieve very precise results.

For memory safety, Cousot and Cousot (1977b) apply abstract interpretation for generating loop invariants in a simple abstract domain which can detect that whether a pointer points to `null` or not. Later on, the local shape analysis (Distefano et al., 2006) infers loop invariants for list-processing programs with separation logic support. To deal with the size information (such as number of nodes in lists/trees), THOR (Magill et al., 2008, 2010) derives a numerical program from the original heap-processing one in a sound way, so that the size information can be obtained with a numerical only loop invariant synthesis. Compared with these works, our approach can handle data structures with stronger invariants such as sortedness and binary search property, which have not been addressed in the previous works. The shape analysis framework TVLA (Sagiv et al., 2002) is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. (2007) report a global shape analysis that discovers inductive structural shape invariants from the code. Compared with these works, our analysis is based on separation logic which can be benefited from its frame rule, and hence supports local reasoning.

4.9 Summary

This chapter presents an analysis framework which allows us to synthesise sound loop invariants over a combined separation and numerical domain. The key components of our analysis include novel operations for abstraction, join and widening in the combined domain. We have built a prototype system and the initial experimental results are reported in chapter 6.

Chapter 5

Full Specification Discovery

Discovering program specifications automatically for heap-manipulating programs is a challenging task due to the complexity of aliasing and mutability of data structures used. In the previous chapter, we present a synthesis approach to generate loop invariants for heap-manipulating programs. In this chapter, we describe a compositional analysis framework for discovering program specifications in a combined abstract domain with shape, numerical and bag (multi-set) information. The framework analyses each method and derives its summary independently from its callers. We propose a novel abstraction method with a bi-abduction technique in the combined domain to discover pre/post-conditions which cannot be automatically inferred before. The analysis does not only prove the memory safety properties, but also finds relationships between pure and shape domains towards full functional

correctness of programs.

5.1 Introduction

In automatic program analysis, certain kinds of program properties have been well explored over the last decades, such as numerical properties in linear abstraction domain, and shape properties for list-manipulating programs in separation domain. However, previous works are not sufficient to automatically analyse program properties in complex mixed domains, especially for programs with sophisticated data structures and strong invariants that involve both structural and pure information. Examples of such properties include: a list remains sorted during the execution of a program, a binary search tree is balanced before and after the execution of a program procedure, the elements of a list remain unchanged after reversing the list. The difficulty is not only due to sharing and mutability of data structures under manipulation, but also the need to track often closely intertwined program properties, such as structural numerical information (length and height), symbolic contents of data structures (bag of values stored in a tree), and relational numerical information (sortedness and balancedness).

In addition to classical shape analyses (e.g. (Bozga et al., 2003; Deutsch, 1994; Jonkers, 1981; Sagiv et al., 2002)), separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002) has been applied to analyse shape properties of heap-manipulating programs in recent years (Calcagno et al., 2009; Distefano et al.,

2006; Yang et al., 2008). These works can automatically infer method specifications for shape properties of programs. A more recent work THOR (Magill et al., 2008, 2010) also incorporates simple numerical information into the shape domain to allow it to automatically synthesise properties like size information of data structures.

However, these previous analyses mainly focus on relatively simple properties, such as pointer safety for lists and list length information. To analyse more complex properties of heap-manipulating programs, such as sortedness and balancedness properties, our recent work (Qin et al., 2011) offers a template-based approach, whereby users supply shape templates in pre-/post-conditions of procedures and the analysis infers the missing pure information to complete the given templates. While that approach does not require an analysis in the combined domain, one of its limitations is that it relies on users to supply the pre-/post-shape templates. If the supplied templates do not cover all the required heap portion, or are not precise enough, or even are essentially unsound for the program (an extreme example being {`true`} Prog {`false`}), it will fail to discover the full specifications.

To overcome these limitations, in this chapter, we propose a direct one-pass analysis in a combined abstraction with *separation*, *numerical* and *bag* information. To the best of our knowledge, this is the first time where such a combination of domains have been used together for inferring pre/post specifications. One advantage of doing so is that we do not only analyse functional correctness and memory safety separately, but also find close relationships between shape and pure (numerical and bag) domains. What

we propose is a compositional analysis by abstract interpretation in such a combined domain. In other words, we analyse a program fragment without any given contextual information, and we analyse each method in a modular way independent of its callers. To generate pre-/postconditions, our analysis adopts a new bi-abduction mechanism over the combined domain, which extends the bi-abduction technique proposed by Calcagno et al. (Calcagno et al., 2009). As in our previous work, our analysis allows users to supply creatively defined shape predicates, while it does not require users to supply partial specifications or annotations for program code to be analysed.

In summary, this chapter makes the following contributions:

- We have designed a program analysis framework which can discover program pre/post-conditions (involving heap, numerical and bag properties) automatically without given any specification about the program.
- For such framework, we have described a compositional analysis for abstract interpretation in a combined pure and shape domain.
- We have defined novel operations for abstraction, join and widening with an extended bi-abduction technique over the combined domain.

Outline. The rest of the chapter is structured as follows. We firstly illustrate our approach informally via three examples (Section 5.2). Formal details

about specification discovery are presented in Section 5.4. Lastly, we present related work and some concluding remarks.

5.2 The Approach

In this section, our approach is informally illustrated by demonstrating three examples. The first example shows a simple program which counts the length of a list. By using this example, we demonstrate our bi-abduction mechanism with simple numerical properties. The second example uses a filter program to show how our analysis works with bag properties. The last one shows the analysis of an insertion sort algorithm by using our extended abstraction with abduction support.

5.2.1 Illustrative Example: Length

Firstly, we illustrate our approach with describing how to discover pre-/post-condition of a recursive method `length` in Figure 5.1. The data structure used in this method is `data Node { Int val; Node next; }`. Based on this data structure, we use a singly linked list segment shape predicate `ls` for this method:

$$\texttt{ls}\langle \texttt{n}, \texttt{p}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node}\langle \_, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{ls}\langle \texttt{m}, \texttt{p}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

The method `length` takes a singly linked list as the input and counts the length of the list recursively until the end of the list, i.e. the method stops when `null` is reached.

```
1 data Node { Int val; Node next; }
2 Int length(Node x)
3 {
4   if (x == null)
5   { return 0; }
6   else {
7     Node t = x.next;
8     Int l = length(t);
9     return r+1;
10  }
11 }
```

Figure 5.1: Counting the length of a list.

Our analysis aims at finding sound and precise specification (summary) (Pre, Post) of the method. Before the analysis, we use a pair (`emp`, `false`) as an initial pre-/post-condition of the method, which means we have no knowledge about the program's requirement or effect yet. During the analysis, we use a pair of states (Pre, Curr) to keep trace of the precondition we have discovered and the current state we have reached respectively. If the current precondition is not sufficient to operate the program command, we use an abductive inference mechanism to synthesize a candidate precondition M as the missing precondition. The starting point of the analysis is (`emp`, `emp`).

We symbolically execute the method body a number of times until the pre-/post-condition are fixed. To ensure the iteration convergence, we apply abstraction, join and widening operations over both shape and pure domains to achieve the fixed point.

First Iteration For the example `length`, in the first iteration, the analysis starts with (`emp`, `emp`) before the condition statement (line 4). The `else` branch from line 6 to 10 now is "short-circuited" since the recursive call is in the code, and the current Post of the method is `false`. To enter the `if` branch (line 5), the condition `x == null` needs to be satisfied with precondition. However, the current Pre is `emp`. To find the missing precondition, we apply abduction mechanism and discover $x = \texttt{null}$ to add to precondition. We now have a paired state ($\texttt{emp} \wedge x = \texttt{null}$, $\texttt{emp} \wedge x = \texttt{null}$) before executing line 5, and after executing `return 0`, we have a summary of the method:

$$\begin{aligned} &(\mathsf{Pre}_1, \mathsf{Post}_1) := \\ &(\texttt{emp} \wedge x = \texttt{null}, \texttt{emp} \wedge x = \texttt{null} \wedge \texttt{res} = 0), \end{aligned} \tag{5.1}$$

where `res` denotes the value returned by the program.

Second Iteration In the second iteration, specification (5.1) is updated as a new summary of method `length`. The starting point of the analysis is reset to (`emp`, `emp`). By executing the `if` branch, we have the same result as summary (5.1). When entering the `else` branch, the state $x \neq \texttt{null}$ is added to the precondition. We have a paired state ($\texttt{emp} \wedge x \neq \texttt{null}$, $\texttt{emp} \wedge x \neq \texttt{null}$) before line 7. The statement `Node t = x.next` tries to access the `next` field of `x`. By abduction, $\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0 \rangle$ is added to Pre, where $\texttt{fv}_0$ and $\texttt{fp}_0$

are fresh logic variables. After line 7, the paired state is ($\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle$, $\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle \wedge \texttt{t} = \texttt{fp}_0$). Now we can use the summary (5.1) for the method call, which requires $\texttt{t} = \texttt{null}$, i.e. $\texttt{fp}_0 = \texttt{null}$ to add to Pre and returns $\texttt{l} = 0$ to add to Curr. Note that $\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle$ as the frame part is discovered by bi-abduction. The frame part is not altered by the method call and passed to the post-state of this call. $\texttt{fp}_0$ is a reachable variable from program variable x, hence we add $\texttt{fp}_0 = \texttt{null}$ to Pre, not $\texttt{t} = \texttt{null}$ to Pre. After line 9, the summary of the else branch is found:

$$(\texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle,\ \texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle \wedge \texttt{res} = 1) \tag{5.2}$$

By joining the formulae (5.1) and (5.2), we update a new summary for the method

$$\begin{aligned}&(\mathsf{Pre}_2, \mathsf{Post}_2) := \\ &(\mathsf{Pre}_1 \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle, \\ &\ \mathsf{Post}_1 \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle \wedge \texttt{res} = 1)\end{aligned} \tag{5.3}$$

Note that the base branch, i.e. the branch without recursive call, is the exit block of a recursive method. It is important to ensure that the precondition allows the program to enter the base branch to guarantee the termination of the method.

Third Iteration In the third iteration, we have specification (5.3) as summary of method length. The result of the if branch is the same as the last iteration. For the else branch, before the recursive call in line 8, we

have a paired state which is the same as the last iteration ($\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle$, $\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle \wedge \texttt{t} = \texttt{fp}_0$). The bi-abduction utility uses updated summary (5.3) and generates the anti-frame part $\texttt{emp} \wedge \texttt{fp}_0 = \texttt{null} \vee \texttt{fp}_0\texttt{::Node}\langle \texttt{fv}_1, \texttt{fp}_1\rangle \wedge \texttt{fp}_1 = \texttt{null}$ and the frame part $\texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle$. The paired state after line 9 is

$$\begin{array}{l}(\texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle * \texttt{fp}_0\texttt{::Node}\langle \texttt{fv}_1, \texttt{null}\rangle, \\ \texttt{x::Node}\langle \texttt{fv}_0, \texttt{null}\rangle \wedge \texttt{res} = 1 \\ \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle * \texttt{fp}_0\texttt{::Node}\langle \texttt{fv}_1, \texttt{null}\rangle \wedge \texttt{res} = 2) \end{array} \quad (5.4)$$

Joining state (5.3) and state (5.4), the new summary of the method is

$$\begin{array}{l}(\mathsf{Pre}_3, \mathsf{Post}_3) := \\ (\mathsf{Pre}_2 \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle * \texttt{fp}_0\texttt{::Node}\langle \texttt{fv}_1, \texttt{null}\rangle, \\ \mathsf{Post}_2 \vee \texttt{x::Node}\langle \texttt{fv}_0, \texttt{fp}_0\rangle * \texttt{fp}_0\texttt{::Node}\langle \texttt{fv}_1, \texttt{null}\rangle \wedge \texttt{res} = 2) \end{array} \quad (5.5)$$

Comparing with summary (5.3), we discover that it is possible that x points to a list with two nodes in the precondition. Note that the auxiliary pointer logical variable $\texttt{fp}_0$ is presented in the summary. If we continue this trend, we will get even longer formulae to cover successive iterations and more additional logical variables. If we do not stop such increasing, the analysis will be an infinite regress. We can interrupt this regress by using abstraction, join and widening operators. By applying abstraction to summary (5.5), we

have:

$$
\begin{array}{l}
(\mathsf{Pre}_3, \mathsf{Post}_3) := \\
(\mathtt{emp} \wedge \mathtt{x} = \mathtt{null} \vee \mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle \\
\quad \vee \mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge \mathtt{xn} = 2, \\
\quad \mathtt{emp} \wedge \mathtt{x} = \mathtt{null} \wedge \mathtt{res} = 0 \vee \mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle \wedge \mathtt{res} = 1 \\
\quad \vee \mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge \mathtt{xn} = 2 \wedge \mathtt{res} = 2)
\end{array} \tag{5.6}
$$

The heap part $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{fp}_0\rangle * \mathtt{fp}_0{::}\mathtt{Node}\langle \mathtt{fv}_1, \mathtt{null}\rangle$ is abstracted as a list $\mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle$ with $\mathtt{xn}$ denoting the length of the list and $\mathtt{xn} = 2$ added into the state. Note that shape parts of first two disjunctive branches, $\mathtt{emp}$ and $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle$, can be entailed by shape formula $\mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle$, and generates numerical information $\mathtt{xn} = 0 \wedge \mathtt{xn} = 1$. We can apply the join operator to the summary (5.6) to have:

$$
\begin{array}{l}
(\mathsf{Pre}_3, \mathsf{Post}_3) := \\
(\mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge 0 \leq \mathtt{xn} \leq 2, \\
\quad \mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge 0 \leq \mathtt{xn} \leq 2 \wedge \mathtt{res} = \mathtt{xn})
\end{array} \tag{5.7}
$$

After join, the widening operator compares summary (5.7) with (5.3). The shape parts of both summaries can be unified. After applying widening operator over the numerical parts, a new summary is updated for the method

$$
\begin{array}{l}
(\mathsf{Pre}_3, \mathsf{Post}_3) := \\
(\mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge 0 \leq \mathtt{xn}, \\
\quad \mathtt{x}{::}\mathtt{ls}\langle \mathtt{xn}, \mathtt{null}\rangle \wedge 0 \leq \mathtt{xn} \wedge \mathtt{res} = \mathtt{xn})
\end{array} \tag{5.8}
$$

Fourth Iteration Following the similar symbolic execution process described above, with the method summary (5.8), we compute a result of this

iteration which is the same as the last one, i.e. the summary (5.8), which means that it is the fixed point and thus the method summary we desired.

The essential steps to terminate the search for suitable preconditions are abstraction and widening. Both operators are tantamount to weakening a state, and they are over-approximation and sound for synthesis of postcondition. However, when such steps are applied to synthesis of precondition, it may make the precondition too weak to be sound. Thus after the analysis, we shall use a forward analysis process to check the discovered summary.

5.2.2 Illustrative Example: Filter

This example `filter` (Figure 5.2) selects elements from a list that satisfy certain condition ($\leq$ `k`), which shows how our approach deals with bag property of data structures. The example is based on the data structure `Node`, and the shape predicate we used is `llB`:

$$\begin{aligned}&\texttt{sllB}\langle \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null}\wedge \texttt{S}{=}\emptyset)\vee\\ &\quad(\texttt{root}{::}\texttt{node}\langle \texttt{v},\texttt{q}\rangle{*}\texttt{q}{::}\texttt{sllB}\langle \texttt{S}_1\rangle\wedge \texttt{S}{=}\texttt{S}_1\sqcup\{\texttt{v}\}\wedge\forall \texttt{u}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{u})\end{aligned}$$

First Iteration The analysis starts with (`emp`, `emp`) as the initial state. The branch from line 5 to line 13 is "short-circuited" in the first iteration because the current Post of the method (`false`) is applied as the effect of the recursive call. To enter line 4, the condition `x == null` needs to be

```
1 Node filter(Node x, int k)
2 {
3   if (x == null) {
4     return x;
5   } else if (x.val <= k) {
6     Node t = x.next;
7     x.next = filter(t, k);
8     return x;
9   } else {
10    Node t = x.next;
11    free(x);
12    x = filter(t, k);
13    return x;
14 } }
```

Figure 5.2: Filtering elements of a list.

satisfied with precondition. We apply our abduction mechanism and discover x=null to add to precondition. After executing return x, we have an initial summary of the method:

$$(\mathsf{Pre}_1, \mathsf{Post}_1) := (\texttt{emp}\wedge\texttt{x=null}, \texttt{emp}\wedge\texttt{res=null}\wedge\texttt{res=x}) \tag{5.9}$$

where res denotes the value returned by the program.

Second Iteration In the second iteration, the specification (5.9) is updated as a new summary of the method. The starting point of the analysis is reset to (emp, emp). By executing the branch in line 4, we have the same result as summary (5.9). To enter the if branch in line 5, we abduct $\texttt{x::Node}\langle\texttt{fv}_0, \texttt{fp}_0\rangle \wedge fv_0 \leq k$ to Pre. The branch from line 5 to 8 accesses the field of x, and recursively calls the method filter. By applying bi-abduction, with the summary that is discovered in last iteration, the summary of this branch is found:

$$\begin{aligned}&(\texttt{x::Node}\langle\texttt{fv}_0, \texttt{null}\rangle\wedge\texttt{fv}_0\leq\texttt{k},\\ &\ \texttt{res::Node}\langle\texttt{fv}_0, \texttt{null}\rangle\wedge\texttt{fv}_0\leq\texttt{k}\wedge\texttt{res=x})\end{aligned} \tag{5.10}$$

Similarly, the summary of line 9 to 14 is calculated as

$$(\texttt{x::Node}\langle\texttt{fv}_0, \texttt{null}\rangle\wedge\texttt{fv}_0>\texttt{k}, \texttt{emp}\wedge\texttt{fv}_0>\texttt{k}\wedge\texttt{res=null}) \tag{5.11}$$

By joining the formulae (5.9), (5.10) and (5.11), and eliminating intermediate logical variables, we update a new summary for the method

$$\begin{aligned}&(\mathsf{Pre}_2, \mathsf{Post}_2) := (\mathsf{Pre}_1 \vee \texttt{x::Node}\langle\texttt{fv}_0, \texttt{null}\rangle,\\ &\ \mathsf{Post}_1 \vee \texttt{emp}\wedge\texttt{fv}_0>\texttt{k}\wedge\texttt{res=null} \vee \texttt{res::Node}\langle\texttt{fv}_0, \texttt{null}\rangle\wedge\texttt{fv}_0\leq\texttt{k}\wedge\texttt{res=x})\end{aligned} \tag{5.12}$$

Third Iteration Based on specification (5.12), a third iteration of symbolic execution is accomplished, and the calculated specification is

$$\begin{aligned}(\mathsf{Pre}_3, \mathsf{Post}_3) := (&\mathsf{Pre}_2 \vee \mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{fp}_0\rangle * \mathtt{fp}_0{::}\mathtt{Node}\langle \mathtt{fv}_1, \mathtt{null}\rangle,\\ &\mathsf{Post}_2 \vee \mathtt{res}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle \wedge \mathtt{fv}_0 \leq \mathtt{k} \wedge \mathtt{fv}_1 > \mathtt{k}\\ &\vee \mathtt{res}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{fp}_0\rangle * \mathtt{fp}_0{::}\mathtt{Node}\langle \mathtt{fv}_1, \mathtt{null}\rangle \wedge \mathtt{fv}_0 \leq \mathtt{k} \wedge \mathtt{fv}_1 \leq \mathtt{k})\end{aligned} \tag{5.13}$$

Comparing with summary (5.12), we discover it is possible that $\mathtt{x}$ points to a list with two nodes in the precondition. If we continue with this trend, the analysis will have termination problem. Therefore, we firstly apply abstraction to the precondition against with the given predicate $\mathtt{llB}$ to eliminate the logical variables, hence the heap part $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{fp}_0\rangle * \mathtt{fp}_0{::}\mathtt{Node}\langle \mathtt{fv}_1, \mathtt{null}\rangle$ is abstracted as $\mathtt{x}{::}\mathtt{llB}\langle \mathtt{n}_1, \mathtt{S}_1\rangle \wedge \mathtt{n}_1{=}2 \wedge \mathtt{S}_1{=}\{\mathtt{fv}_0, \mathtt{fv}_1\}$. Before we join this with Pre_2, the heap formula $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle$ in Pre_2 can be unified as $\mathtt{x}{::}\mathtt{llB}\langle \mathtt{n}_1, \mathtt{S}_1\rangle \wedge \mathtt{n}_1{=}1 \wedge \mathtt{S}_1{=}\mathtt{fv}_0$ and $\mathtt{x}{=}\mathtt{null}$ be $\mathtt{x}{::}\mathtt{llB}\langle \mathtt{n}_1, \mathtt{S}_1\rangle \wedge \mathtt{n}_1{=}0 \wedge \mathtt{S}_1{=}\emptyset$. Then we join the disjunctive formulae if they have the same shape and widen with the Pre_2 to have the precondition as $\mathtt{x}{::}\mathtt{llB}\langle \mathtt{n}_1, \mathtt{S}_1\rangle$. By applying similar operators to postcondition, a new summary is produced:

$$\begin{aligned}(\mathsf{Pre}_3, \mathsf{Post}_3) := (&\mathtt{x}{::}\mathtt{llB}\langle \mathtt{n}_1, \mathtt{S}_1\rangle \wedge 0 \leq \mathtt{n}_1,\\ &\mathtt{res}{::}\mathtt{llB}\langle \mathtt{n}_2, \mathtt{S}_2\rangle \wedge 0 \leq \mathtt{n}_2 \leq \mathtt{n}_1 \wedge (\forall \mathtt{v} \in \mathtt{S}_2 \cdot \mathtt{v} \leq \mathtt{k}) \wedge (\forall \mathtt{v} \in (\mathtt{S}_1 - \mathtt{S}_2) \cdot \mathtt{v} > \mathtt{k}) \wedge \mathtt{S}_2 \subseteq \mathtt{S}_1)\end{aligned} \tag{5.14}$$

Fourth Iteration By the similar symbolic execution process with the method summary (5.14), we compute a result for the fourth iteration to be the same as the last one, namely, (5.14), which is a fixed point desired for our method summary.

This example shows how our analysis works over the combined shape, numerical and bag domain. The fixpoint calculation in bag domain is supported by the tool Fixbag (Pham et al., 2011).

5.2.3 Illustrative Example: Insertion Sort

In this section, we illustrate our approach with another more interesting example insertion sort in Figure 5.3. The method `insert_sort` recursively sorts the tail of the input singly linked list, and inserts the first node into the correct position of the already sorted list and maintains the sortedness properties. The function `insert` takes a list and a node, finds a position of the list, and inserts the node. If the input list is sorted in increasing order, the return list is sorted. The data structure the program used is `Node`. Based on this data structure, two user-defined shape predicates for the analysis are linked list segment `ls`:

$$\texttt{ls}\langle \texttt{n}, \texttt{p}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node}\langle \_, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{ls}\langle \texttt{m}, \texttt{p}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

and sorted linked list segment `sls`:

$$\begin{aligned}\texttt{sls}\langle \texttt{n}, \texttt{s}, \texttt{l}, \texttt{p}\rangle \equiv (&\texttt{root}{::}\texttt{Node}\langle \texttt{s}, \texttt{p}\rangle \wedge \texttt{n}{=}1 \wedge \texttt{l} = \texttt{s}) \vee \\ &\texttt{root}{::}\texttt{Node}\langle \texttt{s}, \texttt{q}\rangle{*}\texttt{q}{::}\texttt{sls}\langle \texttt{m}, \texttt{s1}, \texttt{l}, \texttt{p}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1 \wedge \texttt{s} \leq \texttt{s1} \wedge \texttt{s1} \leq \texttt{l}\end{aligned}$$

One of the challenges of the analysis of `insert` is that the sortedness of input list cannot be detected only based on the footprint of the method. However, if the input list is not sorted, the return list will not be sorted, then the inferred

specification of `insert` cannot be used for `insert_sort` to produce sorted list. In this situation, the user defined predicates can offer some guidance for the analysis. We use this example to show what the problem is, and how we conquer it.

We analyse the method `insert` firstly due to the fact that `insert_sort` depends on it. The initial summary is set to pair (`emp`, `false`), and the starting point of every iteration of the analysis is (`emp`, `emp`).

First Iteration The analysis process of `insert` is analogous to above examples. The path from line 20 to 22 is "short-circuited" in the initial iteration. After the analysis of the first branch in 16, the found summary is

$$\begin{aligned}&(\mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{xp}_0\rangle \wedge \mathtt{xv}_0 \leq \mathtt{rv}_0,\\ &\ \mathtt{res}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{r}\rangle * \mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle \wedge \mathtt{xv}_0 \leq \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{x})\end{aligned} \tag{5.15}$$

After line 19, the summary is

$$\begin{aligned}&(\mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{xp}_0\rangle \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{rp}_0 = \mathtt{null},\\ &\ \mathtt{res}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{null}\rangle \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{r})\end{aligned} \tag{5.16}$$

By joining them, we have

$$\begin{aligned}&(\mathsf{Pre}_1, \mathsf{Post}_1) :=\\ &(\mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{xp}_0\rangle \wedge (\mathtt{xv}_0 \leq \mathtt{rv}_0 \vee \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{rp}_0 = \mathtt{null}),\\ &\ \mathtt{res}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{r}\rangle * \mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle \wedge \mathtt{xv}_0 \leq \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{x}\\ &\quad \vee \mathtt{res}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{null}\rangle \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{r})\end{aligned} \tag{5.17}$$

where Pre_1 means the method terminates in two cases, $\mathtt{xv}_0 \leq \mathtt{rv}_0$ or $\mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{rp}_0 = \mathtt{null}$. Post_1 denotes that if $\mathtt{xv}_0 \leq \mathtt{rv}_0$, the return value will be `x`, and

```
1 data Node { Int val; Node next; }
2 Node insert_sort(Node x)
3 {
4   if (x.next == null)
5   { return x; }
6   else {
7     Node s = x.next;
8     Node r = insert_sort(s);
9     Node t = insert(r, x);
10    return t;
11  }
12 }
13 Node insert(Node r, Node x)
14 {
15  if (x.val <= r.val)
16  { x.next = r; return x;}
17  else if (r.next == null)
18  { r.next = x; x.next = null;
19    return r;   }
20  else {
21    r.next = insert(r.next, x);
22    return r;
23  }
24 }
```

Figure 5.3: Recursive-call based Insertion sort.

next field of x will point to r; if $\mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{rp}_0 = \mathtt{null}$, the return value will be x, next field of x will point to x, and the tail of the list will point to null. It is a very precise description of current state.

Second Iteration In the second iteration, the analysis of the first two branches have the same result as summary 5.17. For the recursive branch, by bi-abduction, we have a paired state

$$\begin{aligned}
&(\mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{rp}_0{::}\mathtt{Node}\langle \mathtt{rv}_1, \mathtt{rp}_1\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{xp}_0\rangle \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \\
&\ (\mathtt{xv}_0 \leq \mathtt{rv}_1 \vee \mathtt{xv}_0 > \mathtt{rv}_1 \wedge \mathtt{rp}_1 = \mathtt{null}), \\
&\ \mathtt{res}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{rp}_0\rangle * \mathtt{rp}_0{::}\mathtt{Node}\langle \mathtt{rv}_1, \mathtt{rp}_1\rangle \\
&\ \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{r} \wedge \mathtt{xv}_0 \leq \mathtt{rv}_1 \vee \\
&\ \mathtt{res}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{rp}_0{::}\mathtt{Node}\langle \mathtt{rv}_1, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{null}\rangle \\
&\ \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{res} = \mathtt{r} \wedge \mathtt{xv}_0 > \mathtt{rv}_1)
\end{aligned} \tag{5.18}$$

Now we need to apply abstraction to this state. The program variable r in the precondition can be easily abstracted to a singly linked list by the technique we used in the previous example. However, the specification is not sufficient for sort algorithm, and the sortedness information is missing to abstract the two nodes $\mathtt{r}{::}\mathtt{Node}\langle \mathtt{rv}_0, \mathtt{rp}_0\rangle * \mathtt{rp}_0{::}\mathtt{Node}\langle \mathtt{rv}_1, \mathtt{rp}_1\rangle$ to a sorted list. The missing information is the numerical relation between rv_0 and rv_1. The guidance of this abstraction comes from the user-defined predicate sls. The user-defined predicates can be viewed as the abstraction strategies which are indicated by the program designer. By applying the abstraction that is equipped with an abduction mechanism to the precondition against the predicate sls, we have $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{rn}_0, \mathtt{rv}_0, \mathtt{rv}_1, \mathtt{rp}_1\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}_0, \mathtt{xp}_0\rangle \wedge \mathtt{xv}_0 > \mathtt{rv}_0 \wedge \mathtt{rn}_0 = 2 \wedge (\mathtt{xv}_0 \leq \mathtt{rv}_1 \vee \mathtt{xv}_0 > \mathtt{rv}_1 \wedge \mathtt{rp}_1 = \mathtt{null})$. By applying join and widening operations with the Pre_1, we have a precondition $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{rn}_0, \mathtt{rs}_0, \mathtt{rl}_0, \mathtt{rp}_0\rangle *$

$\texttt{x}{::}\texttt{Node}\langle \texttt{xv}_0, \texttt{xp}_0\rangle \wedge \texttt{rn}_0 \geq 1 \wedge (\texttt{xv}_0 \leq \texttt{rl}_0 \vee \texttt{xv}_0 > \texttt{rl}_1 \wedge \texttt{rp}_0 = \texttt{null})$. Applying similar process to postcondition, the result of this iteration with sortedness property is calculated as

$$
\begin{aligned}
&(\mathsf{Pre}_2, \mathsf{Post}_2) := \\
&(\texttt{r}{::}\texttt{sls}\langle \texttt{rn}_0, \texttt{rs}_0, \texttt{rl}_0, \texttt{rp}_0\rangle * \texttt{x}{::}\texttt{Node}\langle \texttt{xv}_0, \texttt{xp}_0\rangle \wedge \texttt{rn}_0 \geq 1 \\
&\quad \wedge (\texttt{xv}_0 \leq \texttt{rl}_0 \vee \texttt{xv}_0 > \texttt{rl}_0 \wedge \texttt{rp}_0 = \texttt{null}), \\
&\quad \texttt{res}{::}\texttt{Node}\langle \texttt{xv}_0, \texttt{r}\rangle * \texttt{r}{::}\texttt{sls}\langle \texttt{rn}_0, \texttt{rs}_0, \texttt{rl}_0, \texttt{rp}_0\rangle \wedge \texttt{res} = \texttt{x} \wedge \texttt{xv}_0 \leq \texttt{rs}_0 \wedge \texttt{rn}_0 \geq 1 \\
&\quad \vee \texttt{res}{::}\texttt{sls}\langle \texttt{rn}_1, \texttt{rs}_0, \texttt{rl}_1, \texttt{x}\rangle * \texttt{x}{::}\texttt{sls}\langle \texttt{xn}_1, \texttt{xv}_0, \texttt{xl}_0, \texttt{xp}_1\rangle \\
&\quad\quad \wedge \texttt{xv}_0 > \texttt{rl}_1 \wedge \texttt{res} = \texttt{r} \wedge \texttt{rn}_1 \geq 1 \wedge \texttt{xn}_1 \geq 1 \wedge \texttt{rn}_1 + \texttt{xn}_1 = \texttt{rn}_0 + 1 \\
&\quad\quad \wedge (\texttt{xv}_0 \leq \texttt{rl}_0 \wedge \texttt{xl}_0 = \texttt{rl}_0 \vee \texttt{xv}_0 > \texttt{rl}_0 \wedge \texttt{xl}_0 = \texttt{xv}_0 \wedge \texttt{xp}_1 = \texttt{null})) \qquad (5.19)
\end{aligned}
$$

Third Iteration The analysis of following iterations are similar to what we described above. With join and widening operators, the fixed point of the method summary is reached. The finial result with sortedness property of the method is the same as the specification (5.19).

Similarly, but without bi-abduction for abstracting precondition, another summary of `insert` with non-sortedness property is discovered with `ls` predicate:

$$
\begin{aligned}
&(\mathsf{Pre}, \mathsf{Post}) := \\
&(\texttt{r}{::}\texttt{ls}\langle \texttt{rn}_0, \texttt{null}\rangle * \texttt{x}{::}\texttt{Node}\langle \texttt{xv}_0, \texttt{xp}_0\rangle \wedge \texttt{rn}_0 \geq 1, \\
&\quad \texttt{res}{::}\texttt{Node}\langle \texttt{xv}_0, \texttt{r}\rangle * \texttt{r}{::}\texttt{ls}\langle \texttt{rn}_0, \texttt{null}\rangle \wedge \texttt{res} = \texttt{x} \wedge \texttt{rn}_0 \geq 1 \\
&\quad\quad \vee \texttt{res}{::}\texttt{ls}\langle \texttt{rn}_1, \texttt{x}\rangle * \texttt{x}{::}\texttt{ls}\langle \texttt{xn}_1, \texttt{null}\rangle \\
&\quad\quad\quad \wedge \texttt{res} = \texttt{r} \wedge \texttt{rn}_0 \geq 1 \wedge \texttt{xn}_1 \geq 1 \wedge \texttt{rn}_0 + \texttt{xn}_0 = \texttt{rn}_0 + 1) \qquad (5.20)
\end{aligned}
$$

With the discovered summaries of method `sort`, the specifications of method `insert_sort` are three pairs:

$(\texttt{x::ls}\langle \texttt{xn}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1,$ $\texttt{res::ls}\langle \texttt{xn}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1);$

$(\texttt{x::sls}\langle \texttt{xn}_0, \texttt{xs}_0, \texttt{xl}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1,$ $\texttt{res::sls}\langle \texttt{xn}_0, \texttt{xs}_0, \texttt{xl}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1);$

$(\texttt{x::ls}\langle \texttt{xn}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1,$ $\texttt{res::sls}\langle \texttt{xn}_0, \texttt{xs}_0, \texttt{xl}_0, \texttt{null}\rangle \wedge \texttt{xn}_0 \geq 1);$

5.3 Bi-Abduction

In this section, we introduce our bi-abduction algorithm for discovering missing information in precondition. The bi-abduction extends previous works (Calcagno et al., 2009; Giacobazzi, 1994; Qin et al., 2010b) with more power to work over our combined domain.

Given σ_1 and σ_2, bi-abduction aims to find the anti-frame σ_M and frame part σ_R so that

$$\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R$$

where σ_1 and σ_2 can be considered as the current program state and the requirement of next instruction respectively, σ_M is the missing part which will be propagated back to the precondition and make the analysis continue, and frame part σ_R is the residue from σ_1. The bi-abduction rules are exhibited in Figure 5.4.

The first rule `Residue` triggers when the LHS (σ_1) does not imply the RHS (σ_2) but the RHS implies the LHS with some formula (σ_M) as the residue.

$$\frac{\sigma_1 \not\vdash \sigma_2 * \texttt{true} \quad \sigma_2 \vdash \sigma_1 * \sigma_M \quad \sigma_1 * \sigma_M \vdash \sigma_2 * \sigma_R}{\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R}\ \textbf{Residue}$$

$$\frac{\begin{array}{c}\sigma_1 \not\vdash \sigma_2 * \texttt{true} \quad \sigma_2 \not\vdash \sigma_1 * \texttt{true} \\ \sigma_0 \in \mathsf{unroll}(\sigma_1) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_2) \\ \sigma_0 \vdash \sigma_2 * \sigma_M \ \text{ or } \ \sigma_0 * [\sigma'_M] \rhd \sigma_2 * \sigma_M \qquad \sigma_1 * \sigma_M \vdash \sigma_2 * \sigma_R\end{array}}{\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R}\ \textbf{Unroll}$$

$$\frac{\begin{array}{c}\sigma_1 \not\vdash \sigma_2 * \texttt{true} \quad \sigma_2 \not\vdash \sigma_1 * \texttt{true} \\ \sigma_2 * [\sigma'_M] \rhd \sigma_1 * \sigma_M \quad \sigma_1 * \sigma_M \vdash \sigma_2 * \sigma_R\end{array}}{\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R}\ \textbf{Reverse}$$

$$\frac{\sigma_1 \not\vdash \sigma_2 * \texttt{true} \quad \sigma_2 \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 * \sigma_2 \vdash \sigma_2 * \sigma_R}{\sigma_1 * [\sigma_2] \rhd \sigma_2 * \sigma_R}\ \textbf{Missing}$$

$$\frac{\begin{array}{c}\sigma_1 \not\vdash \sigma_2 * \sigma'_2 * \texttt{true} \quad \sigma_2 * \sigma'_2 \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma'_2 * \texttt{true} \\ \sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma'_R \quad \sigma_1 * \sigma_M \vdash \sigma_2 * \sigma'_2 * \sigma_R\end{array}}{\sigma_1 * [\sigma_M] \rhd (\sigma_2 * \sigma'_2) * \sigma_R}\ \textbf{Remove}$$

Figure 5.4: Bi-Abduction rules.

This rule is quite general and applies in many cases. For the example $\texttt{emp} \not\vdash \texttt{x::Node}\langle\texttt{xv}, \texttt{xp}\rangle$, the RHS can entail the LHS with residue $\texttt{x::Node}\langle\texttt{xv}, \texttt{xp}\rangle$. The abduction then checks whether σ_1 plus the residue implies $\sigma_2 * \sigma_R$ for some σ_R (emp in this example), and returns $\texttt{x::Node}\langle\texttt{xv}, \texttt{xp}\rangle$ as the anti-frame.

The second rule `Unroll` deals with the cast that neither side implies the other. For example, $\texttt{x::sls}\langle\texttt{xn}, \texttt{xs}, \texttt{xl}, \texttt{xp}\rangle$ is LHS (σ_1) and $\texttt{x::node}\langle\texttt{u}, \texttt{p}\rangle *$

$\mathtt{p}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle$ is RHS (σ_2). As the shape predicates in the antecedent σ_1 are formed by disjunctions according to their definitions (like `sls`), its certain disjunctive branches may imply σ_2. As the rule suggests, to accomplish bi-abduction $\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R$, we firstly unfold σ_1 ($\sigma_0 \in \mathsf{unroll}(\sigma_1)$) and try entailment or further abduction with the results (σ_0) against σ_2. If it succeeds with a frame σ_M, then we confirm the abduction by ensuring $\sigma_1 \wedge \sigma_M \vdash \sigma_2 * \sigma_R$. For the example above, the abduction returns $\mathtt{xn}{=}2$ as the anti-frame σ_M and discovers the nontrivial frame $\mathtt{u}{=}\mathtt{xs} \wedge \mathtt{v}{=}\mathtt{xl} \wedge \mathtt{u}{\leq}\mathtt{v} \wedge \mathtt{xp} = \mathtt{q}$ (σ_R). Note that the function $\mathsf{data\_no}$ returns the number of data nodes in a state, e.g. it returns one for $\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{llB}\langle \mathtt{n}, \mathtt{T}\rangle$. (This syntactic check is important for the termination of the abduction.) The unroll unfolds all shape predicates once in σ_1, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n} \sigma_i$), and returns the result as a set of formulae ($\{\sigma_2, ..., \sigma_n\}$).An instance is that it expands $\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{llB}\langle \mathtt{T}\rangle$ to be $\{\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{p}\rangle \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{T}{=}\emptyset, \exists \mathtt{u}, \mathtt{q}, \mathtt{T}_1, \mathtt{k} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{node}\langle \mathtt{u}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{llB}\langle \mathtt{T}_1\rangle \wedge \mathtt{T}{=}\mathtt{T}_1 \cup \{\mathtt{k}\}\}$.

In the third rule `Reverse`, neither side entails the other, and the second rule does not apply. This happens frequently during the abstraction stage in the verification when we need to fold up a "concrete" state of nodes against an abstract shape predicate. For example, $\cdot\mathtt{x}{::}\mathtt{node}\langle \mathtt{u}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle$ is LHS (σ_1) and $\mathtt{x}{::}\mathtt{sls}\langle \mathtt{xn}, \mathtt{xs}, \mathtt{xl}, \mathtt{xp}\rangle$ is RHS (σ_2). In this case the antecedent (σ_1) cannot be unfolded as they are already data nodes. As the rule suggests, it reverses two sides of the entailment and applies the second rule to uncover the constraints σ'_M and σ_M. Then it checks that $\sigma_1 * \sigma_M \vdash \sigma_2 * \sigma_R$ before it returns σ_M. For the example above, the anti-frame $\mathtt{u} \leq \mathtt{v}$ is needed to establish a sorted list.

When an abduction is conducted, the first three rules should be tried firstly; if they do not succeed in finding a solution, then the rule `Missing` is invoked to add the consequence to the antecedent, provided that they are consistent. It is effective for situations like $\texttt{x::node}\langle \_, \_\rangle \not\vdash \texttt{y::node}\langle \_, \_\rangle$, where we should add $\texttt{y::node}\langle \_, \_\rangle$ to the LHS directly (as the other three rules do not apply here). Note that we do not consider `x` and `y` point to the same node unless the aliasing is suggested by the program code.

If the `Missing` rule also fails, we will try our last rule `Remove` which tries to entail a part of consequent by antecedent (σ_1), and remove this part from the consequent if the entailment check succeeds. The anti-frame (σ_M) is continuously calculated with the rest of the consequent.

Example 16 *Considering the abduction question:*

$$(\texttt{x::sls}\langle \texttt{xn}, \texttt{xs}, \texttt{xl}, \texttt{xp}\rangle \wedge \texttt{xn} > 2) * [\sigma_M] \;\; \rhd \;\; \texttt{x::Node}\langle \texttt{v}_0, \texttt{p}_0\rangle * \texttt{y::Node}\langle \texttt{v}_1, \texttt{p}_1\rangle * \sigma_R$$

Our abduction system firstly applies `Remove` *rule to this case, and remove* $\texttt{x::Node}\langle \texttt{v}_0, \texttt{p}_0\rangle$ *from LHS since* $\texttt{x::sls}\langle \texttt{xn}, \texttt{xs}, \texttt{xl}, \texttt{xp}\rangle \wedge \texttt{xn} > 2 \vdash \texttt{x::Node}\langle \texttt{v}_0, \texttt{p}_0\rangle * \texttt{true}$. *Then* `Missing` *rule can be applied to find out* $\texttt{y::Node}\langle \texttt{v}_1, \texttt{p}_1\rangle$ *is missed in antecedent as* σ_M. *The frame* σ_R *is computed by entailment check as* $\texttt{p}_0\texttt{::sls}\langle \texttt{xn}_1, \texttt{xs}_1, \texttt{xl}, \texttt{xp}\rangle \wedge \texttt{v}_0 \leq \texttt{xn}_1 \wedge \texttt{xn} = \texttt{xn}_1 + 1 \wedge \texttt{xn}_1 > 1$. □

The abduction procedures presented in earlier work (Qin et al., 2010b, 2011) have mainly focused on discovering pure information with the assumption that either complete or partial shape information is available. Our bi-abduction algorithm presented in this paper generalises them to cater for

full specification discovery scenarios, whereby we do not have the hints to guide the analysis anymore due to the absence of shape information; but at the same time we can have more freedom as to discover what the missing information is. One observation on abduction is that there can be too many solutions of the anti-frame σ_M for the entailment $\sigma_2 * \sigma_M \vdash \sigma_R * \texttt{true}$ to succeed. Therefore, we define "quality" of anti-frame solutions with the partial order $\preceq$ defined in the last section, i.e. the smaller (weaker) one in two abduction solutions is regarded as the better one. We prefer to find solutions that are (potentially locally) minimal with respect to $\preceq$ and consistent. However, such solutions are generally not easy to compute and could incur excess cost (with additional disjunction in the analysis). Therefore, our abductive inference is designed more from a practical perspective to discover anti-frames that should be suitable as preconditions for programs, and the partial order $\preceq$ sounds more like a guidance of the decision choices of our abduction implementation, rather than a guarantee to find the best theoretically solution.

5.4 Analysis Algorithm

Our proposed analysis algorithm is given in Figure 5.5. The algorithm takes three input parameters: $\mathcal{T}$ as the program environment possibly with some of the method specifications in the program and be ready for newly inferred specifications to be added in, the procedure $t\ mn\ ((t\ \vec{x}); (t\ \vec{y}))\ \{e\}$ to be analysed, and the upper bound n of shared logical variables that we keep during

the analysis.

Our analysis is based on the abstract interpretation framework. It has two distinct features: the abduction exploited in the abstract semantics, and the specifically designed abstraction, join and widening operations over the combined domain.[1] At the beginning, we initialise the iteration variable (i) and the states to record the computed pre- and postconditions (Pre_i and Post_i). The `emp` and `false` here act as initial values that denote the starting point of the fixpoint iteration.

Each iteration starts at line 1. Firstly, we calculate the pre- and post-conditions for the program based on the result from last iteration, with a forward analysis using the abduction-based semantics (line 3). We perform abstraction on both pre- and postconditions obtained to preserve shape domain's finiteness. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 5). Otherwise the obtained results are joined with the results from last iteration (line 6 and 7), and a widening is conducted over both to ensure termination of the analysis (line 8 and 9). Then the new summary will be updated to the program environment (line 10). Finally, we judge whether a fixed-point is already reached or not (line 11). The last few lines (from line 12) are for soundness purposes. We will run a forward analysis over the method body with the discovered specifications to see whether

[1]Note that our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details later.

Fixpoint Computation in Combined Domain

Input: $\mathcal{T},\ t\ mn\ ((t\ \vec{x});(t\ \vec{y}))\ \{e\},\ n$

Local: $i := 0;\ \mathsf{Pre}_i := \mathtt{emp};\ \mathsf{Post}_i := \mathtt{false};$

1 **repeat**

2 $i := i + 1;$

3 $(\mathsf{Pre}_i, \mathsf{Post}_i) := [\![e]\!]^{\mathsf{A}}_{\mathcal{T}}(\mathtt{emp}, \mathtt{emp});$

4 $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{abs_a}^{\dagger}(\mathsf{Pre}_i), \mathsf{abs}^{\dagger}\mathsf{Post}_i);$

5 **if** $\mathsf{Pre}_i \wedge \mathsf{Post}_i = \mathtt{false}$ **or** $\mathsf{cp\_no}(\mathsf{Pre}_i) > n$ **or** $\mathsf{cp\_no}(\mathsf{Post}_i) > n$ **then return** fail **end if**

6 $\mathsf{Pre}_i := \mathsf{join}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i);$

7 $\mathsf{Post}_i := \mathsf{join}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i);$

8 $\mathsf{Pre}_i := \mathsf{widen}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i);$

9 $\mathsf{Post}_i := \mathsf{widen}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i);$

10 $\mathcal{T} := \mathcal{T} \cup \{t\ mn\ ((t\ \vec{x});(t\ \vec{y}))\ \mathit{requires}\ \mathsf{Pre}_i\ \mathit{ensures}\ \mathsf{Post}_i\ \{e\}\};$

11 **until** $\mathcal{T}$ does not changed;

12 $\mathsf{Post} = [\![e]\!]_{\mathcal{T}'}\mathsf{Pre}_i;$

13 **if** $\mathsf{Post} = \mathtt{false}$ **or** $\mathsf{Post} \not\vdash \mathsf{Post}_i * \mathtt{true}$ **then return** fail

14 **else return** $\mathcal{T}$

15 **end if**

Figure 5.5: Main analysis algorithm.

they are sound. If so, then the analysis succeeds; otherwise, fail is returned.

As mentioned previously, the kernel of our analysis include the abstract semantics with abduction, and the abstraction with abduction, join and widening operators. Some of them like join and widening operators are introduced in previous chapter. The new abstract semantics and abstraction are elaborated respectively in the following parts.

5.4.1 Abstract semantics with abduction

As shown in the algorithms, our analysis utilises two semantics: an underlying semantics and an abstract semantics with abduction. They are used to conduct the forward analysis over program body. The type of our underlying semantics is defined as

$$[\![e]\!] \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where AllSpec contains procedure specifications (extracted from *Prog*). For some expression e, given its precondition, the semantics calculates the post-condition. The underlying semantics is built on two transition functions:

$$\begin{array}{lll} \mathsf{unfold}(x) \ : \ \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & & \text{Rearrangement} \\ \mathsf{exec}(ds) \ : \ \mathsf{AllSpec} \rightarrow \mathsf{SH} \rightarrow \mathsf{SH} & & \text{Execution} \end{array}$$

where ds can be either $d[x]$ or d for heap-sensitive command or heap-insensitive command respectively.

whose definitions have been introduced in subsection 4.4.1.

The abstract semantics with abduction is of the type:

$$[\![e]\!]^{\mathsf{A}} \ : \ \mathsf{AllSpec} \to \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \to \mathcal{P}(\mathsf{SH} \times \mathsf{SH})$$

It takes a piece of program and a specification table, to map a (disjunctive) set of pair of symbolic heaps to another such set, where the first in the pair is the accumulated precondition and the second is the current state.

This semantics also consists of the basic transition functions which compose the atomic instructions' semantics and then the program constructors' semantics. Here the basic transition functions are lifted as

$$\begin{array}{l}
\mathsf{Unfold}(x)(\sigma, \sigma_M) =_{df} \\
\quad \textbf{let } \Delta{=}\mathsf{unfold}(x)\sigma \text{ and } S{=}\{(\sigma_1, \sigma_M) \mid \sigma_1 \in \Delta\} \\
\quad \textbf{in if } (\mathtt{false} \notin \Delta) \textbf{ then } S \\
\quad\quad \textbf{else if } (\Delta \vdash x{=}a \text{ for some } a{\in}\mathsf{Reach}(\sigma_1)\,) \text{ and} \\
\quad\quad\quad\quad (\sigma_M \not\vdash \mathtt{a}{::}\mathtt{c}\langle\vec{y}\rangle{*}\mathtt{true} \text{ for fresh } \{\vec{y}\}{\subseteq}\mathsf{LVar}) \\
\quad\quad \textbf{then } S \cup \{(\sigma_1{*}\mathtt{x}{::}\mathtt{c}\langle\vec{y}\rangle, \sigma_M{*}\mathtt{x}{::}\mathtt{c}\langle\vec{y}\rangle) \mid \sigma_1 \in \Delta\} \\
\quad\quad \textbf{else} \quad S \cup \{(\mathtt{false}, \sigma_M)\}
\end{array}$$

$$\begin{array}{l}
\mathsf{Exec}(ds)(\sigma, \sigma_M) =_{df} \textbf{let } \sigma_1{=}\mathsf{exec}(ds)\sigma \textbf{ in } \{(\sigma_1, \sigma_M)\} \\
\quad \text{where } ds \text{ is either } d[x] \text{ or } d, \text{ except procedure call}
\end{array}$$

In the definition of Unfold, we view Δ as a disjunctive set of conjunctive formulae. $\sigma \in \Delta$ denotes that σ is one branch of Δ. The function Reach is defined in subsection 4.4.2 which returns all variables that are reachable from free variables in the abstract state. In the definition of Exec, we need special

treatment for instructions that may alter variable values, like procedure call. As can be seen in the rule below, when a call-by-reference variable y is assigned to a new value after the call, the original value is still preserved with a substitution $\rho = [y_0/y]$ where y_0 is fresh. Doing this allows us to keep the connection among the history values of a variable and its latest value, which may be essential as a link from the procedure's precondition to its postcondition.

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^m \circ [y'_i/v_i]_{i=1}^n \\ \sigma \vdash \rho\Phi_{pr} * \sigma_1 \text{ and } \sigma'_1 = \texttt{emp} \text{ or } \sigma * [\sigma'_1] \rhd \rho\Phi_{pr} * \sigma_1 \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x'_i/u'_i]_{i=1}^m \circ [y'_i/v'_i]_{i=1}^n \quad \rho_l = [r_i/y'_i]_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\begin{array}{l} \mathsf{Exec}(mn(x_1, .., x_m; y_1, .., y_n))(\mathcal{T})(\sigma, \sigma') =_{df} \\ \qquad \{(\sigma_2, \rho_o(\sigma' * \sigma'_1)) \mid \sigma_2 \in (\rho_l\sigma_1) * (\rho_o\Phi_{po})\} \end{array}}$$

A similar lifting function † is defined to lift Unfold's and Exec's domains:

$$\begin{array}{lll} \mathsf{Unfold}^{\dagger}(x)\{(\sigma_i, \sigma'_i)\} & =_{df} & \bigcup(\mathsf{Unfold}(x)(\sigma_i, \sigma'_i)) \\ \mathsf{Exec}^{\dagger}(ds)(\mathcal{T})\{(\sigma_i, \sigma'_i)\} & =_{df} & \bigcup(\mathsf{Exec}(ds)(\mathcal{T})(\sigma_i, \sigma'_i)) \end{array}$$

Based on the above transition functions, the abstract semantics with abduction is defined as follows:

$$
\begin{array}{lll}
[\![d[x]]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & \mathsf{Exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{Unfold}^{\dagger}(x)\{(\sigma,\sigma')\} \\
[\![d]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & \mathsf{Exec}^{\dagger}(d)(\mathcal{T})\{(\sigma,\sigma')\} \\
[\![e_1; e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}} \circ [\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} \\
[\![\texttt{if } (v) \ e_1 \ \texttt{else } e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & ([\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}\{(v \wedge \sigma,\sigma')\}) \cup ([\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\neg v \wedge \sigma,\sigma')\})
\end{array}
$$

$$
\frac{[\![e]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} = \{(\sigma_1,{\sigma'}_1)\} \quad \rho = [x_1/x', r_1/\texttt{res}] \quad \textit{fresh logical } x_1, r_1}{[\![x := e]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} =_{df} \{((\rho\sigma_1) \wedge x'{=}r_1, \rho{\sigma'}_1)\}}
$$

The abstract semantics forms the foundation of our forward analysis for program body.

Example 17 *Here the example tries to free a node given as input.*

```
1 void freeNode(Node x)
2 { free(x); }
```

Since `free` *is a heap sensitive command, the analysis applies* Unfold *to* `x` *and abducts that* `x` *should be a node in the precondition.*

However, if the example tries to free a node twice as

```
1 void freeNode(Node x)
2 { free(x); free(x); }
```

The abduction will generate $\texttt{x::Node}\langle\_\rangle * \texttt{x::Node}\langle\_\rangle$ *that is false. This means we may discover a bug in the program code.* □

Example 18 *Considering the program which has if-else expression*

```
1 void assert_next(Node x, Node y)
2 { if (x.val == 0) {x.next = y;}
3   else {y.next = x;} }
```

During the interpretation of the if-condition, our analysis will split the precondition for case analysis. A pair of precondition can be discovered,

$$\texttt{x::Node}\langle\texttt{xv}, \_\rangle \wedge \texttt{xv} = 0\,;$$
$$\texttt{x::Node}\langle\texttt{xv}, \_\rangle * \texttt{y::Node}\langle\_, \_\rangle \wedge \texttt{xv} \neq 0$$

The if-branch does not touch any field of `y`*, hence no requirement for* `y` *is needed if* `x.val = 0`*. However, if* `x.val` $\neq$ `0`*,* `y` *is a node will be required. From this simple example, we can observe that some shape properties are related to the numerical information.* □

5.4.2 Abstraction with Abduction

The definitions of abstraction, join and widening operations for full specification discovery in this chapter are similar to the definitions of them in chapter 4. The major difference is that we now have a specifically designed abstraction with abduction for precondition (abs_a) which is used to discover

extra obligations for the abstraction of precondition. It consists of two rules:

$$\mathsf{abs_a}(\sigma) =_{df} \mathsf{abs}(\sigma)$$

$$\frac{\mathsf{c_2}\langle\vec{u_2}\rangle \equiv \Phi \qquad \mathtt{p}{::}\mathsf{c_1}\langle\vec{v_1}\rangle{*}\sigma_1{*}[\sigma'] \rhd \mathtt{p}{::}\mathsf{c_2}\langle\vec{v_2}\rangle{*}\sigma_2 \quad \mathsf{Reach}(\mathtt{p}{::}\mathsf{c_2}\langle\vec{v_2}\rangle{*}\sigma_2) \cap \{\vec{v_1}\} = \emptyset}{\mathsf{abs_a}(\mathtt{p}{::}\mathsf{c_1}\langle\vec{v_1}\rangle{*}\sigma_1) =_{df} \mathtt{p}{::}\mathsf{c_2}\langle\vec{v_2}\rangle{*}\sigma_2}$$

The first rule makes use of abs and does not find new constraints for precondition. The second rule tries to abstract the state $\mathtt{p}{::}\mathsf{c_1}\langle\vec{v_1}\rangle{*}\sigma_1$ with a stronger predicate $\mathsf{c_2}$ against it while abs failed to abstract, and discovers extra σ' to be propagated back to the precondition for the abstraction to succeed. The lifting function for $\mathsf{abs_a}$ is analogously defined as that for abs.

$$\mathsf{abs_a}^{\dagger}(\bigvee \sigma_i) =_{df} \bigvee \mathsf{abs_a}(\sigma_i)$$

Example 19 *Recalling the insertion sort example, we want to compute the following abstraction:*

$$\mathsf{abs_a}(\mathtt{x}{::}\mathtt{Node}\langle\mathtt{v_0},\mathtt{p_0}\rangle * \mathtt{p_0}{::}\mathtt{Node}\langle\mathtt{v_1},\mathtt{p_1}\rangle)$$

if we use user-defined predicate `sls` *as the abstraction strategy, the numerical relation of* $\mathtt{v_0}$ *and* $\mathtt{v_1}$ *is missing. Fortunately, the missing information can be found by abduction to make the abstraction succeed, and have the result as* $\mathtt{x}{::}\mathtt{sls}\langle 2,\mathtt{v_0},\mathtt{v_1},\mathtt{p_1}\rangle$. □

Note that, firstly, abduction is a strength process in abstracting the precondition, thus it is a safe process to add information to generate the precondition.

Secondly, although finding the weakest precondition of programs is ideal, the precondition may be too weak to guarantee its functional correctness if we only look at the footprint of the program code. Therefore, it is reasonable to make use of the user-defined predicates as the guidance to enhance the abstraction strategy.

5.4.3 Soundness and Termination

Soundness

The soundness of our analysis is ensured by the soundness of the following: the entailment prover (Nguyen et al., 2007; Chin et al., 2010), the abstract semantics (w.r.t. concrete semantics), the bi-abduction, the abstraction, and the join and widening operators. The underlying operational semantics of our language, and its concrete program state are described in chapter 3. Based on that, we have the following theorem:

Theorem 20 (Soundness) *Let (*Pre, Post*) denote the pre- and postcondition discovered by our analysis for a program* e. *The analysis is sound if* $\forall\ s, h\ \cdot\ s, h \models Post(\mathsf{Pre})$ *and* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$ *(for some* s', h'*), we have* $s', h' \models Post(\mathsf{Post})$.

where the function $Post(\Delta)$ is defined in subsection 4.4.3. Our analysis is sound following the soundness of entailment checking, abstract semantics,

and the operations of bi-abduction, abstraction, join and widening. The soundness of entailment checking is established by structural induction over the abstract domain. The soundness of abstract semantics is proven by induction over program constructors, and the soundness of bi-abduction is proven by the soundness of entailment checking. The soundness of abstraction, join and widening is proven in subsection 4.4.3. Finally, as abstraction and widening for precondition are essentially unsound, we perform in the analysis algorithm a final check to ensure soundness, which is guaranteed by the soundness of abstract semantics.

Lemma 21 (Soundness of entailment checking) *If* $\Delta \vdash \Delta'$*, we have* $\forall s, h$*, if* $s, h \models \Delta$*, then* $s, h \models \Delta'$*.*

Proof. The proof is done by structural induction over formulae constructors of the abstract domain (Nguyen et al., 2007; Chin et al., 2010). □

Lemma 22 (Soundness of abstract semantics) *If* $[\![e]\!]_{\mathcal{T}}(\Delta, 0) = (\Delta_1, 0)$*, then for all* s, h*, if* $s, h \models Post(\Delta)$ *and* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$*, then there always exists* Δ_0 *so that*

$$s_1, h_1 \models Post(\Delta_0) \qquad \textit{and} \qquad [\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$$

Proof. The proof is done by structural induction over program constructors and is left in the Appendix. □

Lemma 23 (Soundness of abduction) *If* $\sigma_1 * [\sigma_M] \rhd \sigma_2 * \sigma_R$*, then* $\forall s, h \models Post(\sigma_1 * \sigma_M)$*, we have* $s, h \models Post(\sigma_2 * \sigma_R)$*.*

Proof. This is ensured by the entailment relationship in the premise of each of the abduction rules and the soundness of the entailment checking. □

Termination

For the termination aspect, we have the following theorem:

Theorem 24 (Termination) *The analysis of full specification discovery will terminate in finite steps for finite input parameters and user-defined predicates with a given finite upper bound for the number of logical variables.*

Proof. The proof is based on two facts: 1), the finiteness over the shape domain guaranteed by our abstraction and the restriction on the number of logical variables 2), the termination over the numerical domain guaranteed by our widening operator. These two facts guarantee the convergence of our analysis. Both of them are proven in section 4.4.3.

5.5 Related Work

Dramatic advances have been made in synthesising heap-manipulating programs' specifications since the emergence of separation logic. The SpaceInvader tool (Yang et al., 2008; Calcagno et al., 2011) infers full method speci-

fications over the separation domain, as to verify pointer safety for industrial programs up to 10K lines of code. The SLAyer tool (Gotsman et al., 2006) implements an inter-procedural analysis for programs with shape information. To deal with size information (such as number of nodes in lists/trees), THOR (Magill et al., 2010) derives a numerical program from the original heap-processing one in a sound way, so that the size properties can be obtained by numerical analysis. A similar approach (Gulwani et al., 2009) combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can discover specification with stronger invariants such as sortedness and bag-related properties, which have not been addressed in the previous works. Two more works to be mentioned are relational inductive shape analysis (Chang and Rival, 2008) and our previous inference works (Qin et al., 2010a, 2011). These works can handle shape and numerical information over a combined domain. However, they still require user annotation for the program code whereas we compute the whole specification at once.

There are also other approaches that can synthesise shape-related program invariants other than those based on separation logic. The shape analysis framework TVLA (Sagiv et al., 2002) is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. (2007) reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. (2002) developed a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Hackett and Rugina (2005) can deal with

AVL-trees but is customised to handle only tree-like structures with height property. Type-based approaches (Rondon et al., 2008, 2010) are also used to infer numerical constraints for given type templates, but limited to capture flow sensitive constraints. Compared with these works, separation logic based approach benefits from the frame rule with support for local reasoning.

5.6 Summary

We have reported in this chapter a program analysis which automatically discovers program specifications over a combined separation and pure domain. The key components of our analysis include an abduction for precondition discovery, and novel operations for abstraction, join and widening in the combined domain. We have built a prototype system and the initial experimental results will be reported in chapter 6.

Chapter 6

Experiments and Evaluation

This chapter presents the experimental results from two implemented systems. One system synthesises loop invariants and the other discovers full specifications for programs that manipulate shared and mutable sophisticated data structures. The evaluations of this thesis is discussed based on the results. The advantage and limitation of the proposed frameworks are also discussed in this chapter.

6.1 Experiments and Evaluation of Loop Invariants Synthesis

We have implemented a prototype system of the loop invariants synthesis framework for the evaluation purpose. The prototype system was built in Objective Caml. We used SLEEK (Nguyen et al., 2007) as the solver for entailment checking over the heap domain, and Omega constraint solver (Pugh, 1991) and Fixcalc solver (Popeea and Chin, 2006) for join and widening operations in the numerical domain. Our test platform was an Intel Core 2 CPU 2.66GHz system with 8Gb RAM.

Example [Merge] Fig. 6.1 shows a `merge` procedure which merges two sorted lists `left` and `right`, and returns a sorted list as the result. The precondition of the while loop within the `merge` procedure (starting at line 6) is calculated as

$$\texttt{left::sll}\langle \texttt{n1}, \texttt{s1}, \texttt{l1}\rangle * \texttt{right::sll}\langle \texttt{n2}, \texttt{s2}, \texttt{l2}\rangle \wedge \texttt{r} = \texttt{null}$$

The function `append` concatenates two sorted lists, which requires that the maximal value stored in the first input list is smaller than or equal to the minimal value of the second input list, and ensures that it returns a concatenated sorted list (referred to by `res`). The following is the specification of `append`:

```
1 node append(node x, node y)
2 requires x::sll⟨n1, s1, l1⟩*y::sll⟨n2, s2, l2⟩∧l1≤s2
3 ensures res::sll⟨n1+n2, s1, l2⟩;
```

```
1  node merge(node left, node right)
2  requires  left::sll⟨n1, s1, l1⟩ ∗ right::sll⟨n2, s2, l2⟩
3  ensures   res::sll⟨n3, s3, l3⟩ ∧ n3 = n1 + n2
4              ∧ s3 = min(s1, s2) ∧ l3 = max(l1, l2);
5  {
6    node r = null;
7    while (left != null && right != null) {
8      if (left.val <= right.val) {
9        node tmp = left;
10       left = left.next;
11       tmp.next = null;
12       r = append(r,tmp);
13     }
14     else {
15       node tmp = right;
16       right = right.next;
17       tmp.next = null;
18       r = append(r,tmp);
19     }
20   }
21   if (left == null) {
22     r = append(r, right);
23   }
24   else {
25     r = append(r, left);
26   }
27   return r;
28 }
```

Figure 6.1: Loop-based Merge.

By applying our analysis system to the program code, the following loop invariant is discovered for the while loop:

$$\texttt{left}::\texttt{sll}\langle \texttt{n}_1, \texttt{s}_1, \texttt{l}_1\rangle * \texttt{right}::\texttt{sll}\langle \texttt{n}_2, \texttt{s}_2, \texttt{l}_2\rangle \wedge \texttt{r} = \texttt{null} \tag{6.1}$$

$$\vee\ \texttt{r}::\texttt{node}\langle \texttt{s}_1, \texttt{null}\rangle * \texttt{right}::\texttt{sll}\langle \texttt{n}_2, \texttt{s}_2, \texttt{l}_2\rangle \wedge \texttt{left} = \texttt{null} \wedge \texttt{s}_1 = \texttt{l}_1 \wedge \texttt{s}_1 < \texttt{s}_2 \tag{6.2}$$

$$\vee\ \texttt{r}::\texttt{node}\langle \texttt{s}_2, \texttt{null}\rangle * \texttt{left}::\texttt{sll}\langle \texttt{n}_1, \texttt{s}_1, \texttt{l}_1\rangle \wedge \texttt{right} = \texttt{null} \wedge \texttt{s}_2 = \texttt{l}_2 \wedge \texttt{s}_2 \leq \texttt{s}_1 \tag{6.3}$$

$$\begin{aligned}\vee\ & \texttt{r}::\texttt{node}\langle \texttt{s}_3, \texttt{null}\rangle * \texttt{left}::\texttt{sll}\langle \texttt{n}'_1, \texttt{s}'_1, \texttt{l}_1\rangle * \texttt{right}::\texttt{sll}\langle \texttt{n}'_2, \texttt{s}'_2, \texttt{l}_2\rangle \\ & \wedge (\texttt{s}_3 = \texttt{s}_1 \wedge \texttt{s}_1 < \texttt{s}_2 \wedge \texttt{n}_1 = \texttt{n}'_1 + 1 \wedge \texttt{n}'_2 = \texttt{n}_2 \wedge \texttt{s}_1 \leq \texttt{s}'_1 \wedge \texttt{s}'_2 = \texttt{s}_2 \\ & \quad \vee\ \texttt{s}_3 = \texttt{s}_2 \wedge \texttt{s}_2 \leq \texttt{s}_1 \wedge \texttt{n}_2 = \texttt{n}'_2 + 1 \wedge \texttt{n}'_1 = \texttt{n}_1 \wedge \texttt{s}_2 \leq \texttt{s}'_2 \wedge \texttt{s}'_1 = \texttt{s}_1)\end{aligned} \tag{6.4}$$

$$\begin{aligned}\vee\ & \texttt{r}::\texttt{sll}\langle \texttt{n}_3, \texttt{s}_3, \texttt{l}_3\rangle * \texttt{right}::\texttt{sll}\langle \texttt{n}'_2, \texttt{s}'_2, \texttt{l}_2\rangle \wedge \texttt{left} = \texttt{null} \wedge \texttt{n}_3 = \texttt{n}_1 + \texttt{n}_2 - \texttt{n}'_2 \\ & \wedge \texttt{s}_3 = \texttt{min}(\texttt{s}_1, \texttt{s}_2) \wedge \texttt{l}_3 = \texttt{l}_1 \wedge \texttt{l}_1 < \texttt{s}'_2\end{aligned} \tag{6.5}$$

$$\begin{aligned}\vee\ & \texttt{r}::\texttt{sll}\langle \texttt{n}_3, \texttt{s}_3, \texttt{l}_3\rangle * \texttt{left}::\texttt{sll}\langle \texttt{n}'_1, \texttt{s}'_1, \texttt{l}_1\rangle \wedge \texttt{right} = \texttt{null} \wedge \texttt{n}_3 = \texttt{n}_2 + \texttt{n}_1 - \texttt{n}'_1 \\ & \wedge \texttt{s}_3 = \texttt{min}(\texttt{s}_1, \texttt{s}_2) \wedge \texttt{l}_3 = \texttt{l}_2 \wedge \texttt{l}_2 \leq \texttt{s}'_1\end{aligned} \tag{6.6}$$

$$\begin{aligned}\vee\ & \texttt{r}::\texttt{sll}\langle \texttt{n}_3, \texttt{s}_3, \texttt{l}_3\rangle * \texttt{left}::\texttt{sll}\langle \texttt{n}'_1, \texttt{s}'_1, \texttt{l}_1\rangle * \texttt{right}::\texttt{sll}\langle \texttt{n}'_2, \texttt{s}'_2, \texttt{l}_2\rangle \\ & \wedge \texttt{n}_3 = \texttt{n}_1 - \texttt{n}'_1 + \texttt{n}_2 - \texttt{n}'_2 \wedge \texttt{s}_3 = \texttt{min}(\texttt{s}_1, \texttt{s}_2) \wedge \texttt{l}_3 \leq \texttt{min}(\texttt{s}'_1, \texttt{s}'_2)\end{aligned} \tag{6.7}$$

The loop invariant contains seven different disjunctive branches, with each branch describing a different situation. The branch (6.1) represents the state before any iteration. The branches (6.2), (6.3), and (6.4) denote three special scenarios after one iteration. The branch (6.2) (resp. (6.3)) denotes the case where initially the `left` (resp. `right`) list contains only one node which holds a value no bigger than any value stored in the `right` (resp. `left`) list, and after one iteration, `r` refers to the sole node in the initial `left` (resp. `right`) list and the `left` (resp. `right`) pointer becomes null. The branch (6.4) denotes the scenario where after one iteration neither the `left` list

nor the `right` list is empty but `r` still refers to the node with the smallest value. The branches (6.5) , (6.6) and (6.7) denote possible states reached after some (one or more) iterations. The branch (6.5) (resp. (6.6)) denotes the state reached after some iterations where the `left` (resp. `right`) pointer has traversed to the end of the list. The branch (6.7) denotes the case where neither the `left` pointer nor the `right` pointer has reached the end of their lists after some iterations. In all these three branches, `r` refers to the merged list obtained so far.

Note that branches (6.2), (6.3) and (6.4) are, respectively, special cases of branches (6.5), (6.6) and (6.7) (logically, the former formulae entail the latter ones respectively). Thus we can simplify the loop invariant as

$$
\begin{aligned}
&\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n}_3, \mathtt{s}_3, \mathtt{l}_3\rangle * \mathtt{right}{::}\mathtt{sll}\langle \mathtt{n}'_2, \mathtt{s}'_2, \mathtt{l}_2\rangle \wedge \mathtt{left} = \mathtt{null} \wedge \mathtt{n}_3 = \mathtt{n}_1 + \mathtt{n}_2 - \mathtt{n}'_2 \\
&\quad \wedge \mathtt{s}_3 = \mathtt{min}(\mathtt{s}_1, \mathtt{s}_2) \wedge \mathtt{l}_3 = \mathtt{l}_1 \wedge \mathtt{l}_1 {<} \mathtt{s}'_2 \\
\vee\ &\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n}_3, \mathtt{s}_3, \mathtt{l}_3\rangle * \mathtt{left}{::}\mathtt{sll}\langle \mathtt{n}'_1, \mathtt{s}'_1, \mathtt{l}_1\rangle \wedge \mathtt{right} = \mathtt{null} \wedge \mathtt{n}_3 = \mathtt{n}_2 + \mathtt{n}_1 - \mathtt{n}'_1 \\
&\quad \wedge \mathtt{s}_3 = \mathtt{min}(\mathtt{s}_1, \mathtt{s}_2) \wedge \mathtt{l}_3 = \mathtt{l}_2 \wedge \mathtt{l}_2 {\leq} \mathtt{s}'_1 \\
\vee\ &\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n}_3, \mathtt{s}_3, \mathtt{l}_3\rangle * \mathtt{left}{::}\mathtt{sll}\langle \mathtt{n}'_1, \mathtt{s}'_1, \mathtt{l}_1\rangle * \mathtt{right}{::}\mathtt{sll}\langle \mathtt{n}'_2, \mathtt{s}'_2, \mathtt{l}_2\rangle \\
&\quad \wedge \mathtt{n}_3 = \mathtt{n}_1 - \mathtt{n}'_1 + \mathtt{n}_2 - \mathtt{n}'_2 \wedge \mathtt{s}_3 = \mathtt{min}(\mathtt{s}_1, \mathtt{s}_2) \wedge \mathtt{l}_3 {\leq} \mathtt{min}(\mathtt{s}'_1, \mathtt{s}'_2)
\end{aligned}
$$

The simplification of generated loop invariant increases the scalability of our inference system, and also simplifies the verification process by using the loop invariant. □

We have four tables to depict the programs that we have used to conduct experiments with, including list processing programs (Table 6.1), sorting algorithms (Table 6.2), tree processing programs(Table 6.3), and loops from

Program	**Function**	**Time**
`create`	Creates a list with given length parameter	0.452
`delete`	Disposes a list	0.720
`traverse`	Traverses a list	0.636
`length`	Counts the length of a list	0.772
`append`	Appends two sorted lists	0.312
`take`	Takes the first n elements of a list, or itself	0.852
`drop`	Returns suffix of a list after the first n elements, or `null`	0.844
`reverse`	Reverses the elements of the list, in place	1.032
`filter`	Drops the elements bigger than k of a list	1.182
`lookup`	Returns the first node whose values equals to k, or `null`	0.876
`drop_even`	Drops all the elements whose indexes are even	1.332

(Total LOC: 232)

Table 6.1: Loop invariants synthesis Experimental Results for Lists.

Program	Function	Time
`ins_sort(inner)`	Inner loop of Fig. 4.1	0.824 .
`ins_sort(outer)`	Outer loop of Fig. 4.1	4.372
`partition`	Auxiliary operation used by Quick-sort	1.497
`merge`	Merges two sorted lists to be one sorted list	1.972
`split`	Divides a list into two sublists with length difference of at most one	0.354
`select`	Selects the smallest node of a list	0.692
`select_sort`	Outer loop of selection sort	4.892
`drop_even`	Drops all the elements whose indexes are even	1.332

(Total LOC: 178)

Table 6.2: Loop invariants synthesis Experimental Results for Sorting Algorithm.

Program	Function	Time
`tree_search`	Finds a node in a binary search tree	1.294
`tree_insert`	Inserts a node into a binary search tree	1.364
`list2tree`	Inserts nodes of a list into a binary search tree	5.176

(Total LOC: 87)

Table 6.3: Loop invariants synthesis Experimental Results for Trees.

Program	Function	Time
`list.c`	One loop in FreeRTOS *list.c* with heap manipulation	4.124
`task.c`	Six loops in FreeRTOS *task.c* with heap manipulation	32.18

(Total LOC: 331)

Table 6.4: Loop invariants synthesis Experimental Results for FreeRTOS.

FreeRTOS (Barry, 2009) (Table 6.4). Total LOC denotes the total number of lines of the code. The first column denotes the names of the programs. The second column states the programs' functionalities. The last column exhibits the time in second that is taken by our analysis. As can be seen from their functions, these programs involve recursive data structures such as (sorted) linked lists and binary (search) trees, and employ loops to manipulate these data structures (and some of them even have nested loops). Our target is to verify these programs with the help of our analysis over the loops they invoke, so that user annotations for while loops can be avoided. Our experiments confirm that HIP/SLEEK can verify all these programs successfully when supplying with loop invariants discovered by our analysis. According to our experience, these experiments just require the bound of shared cutpoints be a reasonably small number, saying no more than twice of the number of program variables. Note that it takes a longer time to analyse the procedures that have nested loops, such as `select_sort`, `list2tree`, and so on (because we need to analyse the inner loop multiple times). We transfer the source code of FreeRTOS to our language, and successfully infer the loop invariants of the loops which do not involve pointer arithmetic in *list.c* and *task.c*. We employ doubly linked sorted list predicate for the verification of FreeRTOS.

We have two main observations from our experimental results. The first is that we can handle many data structures with rich program properties they bear. To analyse these loops, we need to deal with both singly linked and doubly linked list predicates to capture the list data structure, as well as their sorted version for the sorting algorithms. We can also handle tree-like predicates such as binary trees and binary search trees. Meanwhile, these predicates also come along with many properties such as the length of the list and size/height of the tree, and the minimum/maximum value of a sorted list/binary search tree. Based on them, our analysis is capable of expressing the invariants of these properties in terms of the constraints over the predicates' parameters.

Beyond the number of predicates and properties we can process, another observation on our analysis is that we can process them *rather precisely*. For example, the list creation program creates a list with the same length as user input, list *traverse* does not change list's length, all of elements of the return list of *filter* program are smaller than or equal to the input value k, and the length of return list of *drop_even* is between `half` and `half+1` of the original list.

Besides these, some loops provide critical invariants for the method to process them to function correctly. For example, the quicksort algorithm partitions a list into three parts, where two are lists and the third just one node, whose value is exactly in the middle of that of the two other lists (`partition` in the table). We use a list bound predicate to indicate that fact which is successfully inferred by our analysis. We can also infer that the first loop of a

mergesort (`split` in the table) can divide the list into two portions whose length difference is at most one, which is unimportant for the algorithm's functional correctness but essential for its performance. For `tree_insert`, we have the result that the tree's height is increased at most one, and the minimum/maximum value of the new binary search tree will be exactly the inserted value, if that value is out of the value bounds of the original tree. For code in FreeRTOS, the invariants we inferred maintains the sortedness property for the doubly linked list used for tasks. The invariants we discovered are sufficiently precise to prove the functional correctness of all these programs with the given predicates.

We also observe that not all synthesised invariant are adequate to prove the postcondition of the method. For the insertion sort example in Figure 4.1, if we use the singly linked list predicate `ll` as the target of abstraction, the synthesised loop invariant for the outer loop will be:

$$(\mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{n_x}{=}\mathtt{n}) \vee (\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{null}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}1) \vee$$
$$(\mathtt{r}{::}\mathtt{ll}\langle \mathtt{n_r}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}\mathtt{n_r} \wedge \mathtt{n_r}{\geq}2)$$

This result is a sound invariant, but it is not sufficient to prove the given postcondition. Such invariant will be filtered out in our post checking process.

Some data structures are beyond the capability of our system. For instance, a pointer field of a class node is non-local, namely, it does not have a direct relationship with fields of surrounding objects, but rather is determined by some global constraint. For example, the skip lists (Pugh, 1990) are not captured by our approach. Another limitation of our approach is to analyse the programs with overlaid data structures. The overlaid data structure

indicates that a set of data nodes have been used by multiple data structures, and these data structures are manipulated at the same time (Lee et al., 2011). For example, the deadline IO scheduler of Linux has a queue whose nodes been used by a linked list and a tree. The linked list is used to record the order of insertion of each node, and the tree supports an efficient indexing structure of the nodes. For such data structure, we cannot define a recursive heap predicate to capture all the shape and pure properties by our specification mechanism. If we supply both list and tree shape predicates, since every node are shared by the two data structures, we cannot apply our abstraction operation over the program states.

6.2 Experiments and Evaluation of Full Specification Discovery

We have implemented a prototype system of the full specification discovery framework and evaluated it over a number of heap-manipulating programs to test its viability and precision. We used Sleek (Nguyen et al., 2007) as the solver for entailment checking over the heap domain, and Fixcalc (Popeea and Chin, 2006) and Fixbag (Pham et al., 2011) for numerical and bag domain. Our experimental results are achieved with an Intel Core 2 Quad CPU 2.66GHz with 8Gb RAM platform.

Example [Merge] We show a similar `merge` example without given precon-

dition, which has been declared as an unverified example in Calcagno et al. (2011) since their method does not keep track of values stored in the list. This example (Figure 6.2) merges two sorted lists into one sorted list. If either of the input list is empty, then the other one is returned; if not, we select the smallest element of both sorted lists, and make the next field of the smallest node points to the result of merging the tail list of this node with the other list. The shape predicate selected for this example is `sls` which keeps track on both the minimal (`sm`) and maximal (`lg`) values of a sorted list.

$$\begin{aligned}\texttt{sls}\langle \texttt{n}, \texttt{sm}, \texttt{lg}, \texttt{p}\rangle \equiv\ & \texttt{root::Node}\langle \texttt{sm}, \texttt{p}\rangle \wedge \texttt{n=1} \wedge \texttt{lg=sm} \ \vee \\ & \texttt{root::Node}\langle \texttt{sm}, \texttt{q}\rangle * \texttt{q::sls}\langle \texttt{n}_1, \texttt{s}_1, \texttt{lg}, \texttt{p}\rangle \wedge \texttt{n=n}_1\texttt{+1} \wedge \texttt{sm} \leq \texttt{s}_1 \wedge \texttt{s}_1 \leq \texttt{lg}\end{aligned}$$

Supposing after the third iteration of symbolically executing the code, we have generated a precondition as follows:

$$\begin{aligned}&\texttt{x=null} \vee \texttt{y=null} \vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle \\ &\quad \wedge (\texttt{xv}_1 \leq \texttt{yv}_1 \wedge \texttt{xp}_1\texttt{=null} \vee \texttt{xv}_1 > \texttt{yv}_1 \wedge \texttt{yp}_1\texttt{=null}) \qquad (6.8) \\ &\vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{xp}_1\texttt{::Node}\langle \texttt{xv}_2, \texttt{xp}_2\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle \\ &\quad \wedge (\texttt{xv}_1 \leq \texttt{yv}_1 \wedge (\texttt{xv}_2 \leq \texttt{yv}_1 \wedge \texttt{xp}_2\texttt{=null} \vee \texttt{xv}_2 > \texttt{yv}_1 \wedge \texttt{yp}_1\texttt{=null})) \qquad (6.9) \\ &\vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle * \texttt{yp}_1\texttt{::Node}\langle \texttt{yv}_2, \texttt{yp}_2\rangle \\ &\quad \wedge (\texttt{xv}_1 > \texttt{yv}_1 \wedge (\texttt{xv}_1 \leq \texttt{yv}_2 \wedge \texttt{xp}_1\texttt{=null} \vee \texttt{xv}_1 > \texttt{yv}_2 \wedge \texttt{yp}_2\texttt{=null})) \qquad (6.10)\end{aligned}$$

Branch (6.8) says that the program only touches the second node of `x` if

```
1 Node merge(Node x, Node y)
2 {
3   if (x == null) {
4     return y;
5   } else if (y == null) {
6     return x;
7   } else
8   if (x.val <= y.val) {
9     Node t = x.next;
10    x.next = merge(t, y);
11    return x;
12  } else {
13    Node t = y.next;
14    y.next = merge(x, t);
15    return y;
16 } }
```

Figure 6.2: Recursive-call based Merge.

$\texttt{xv}_1 \leq \texttt{yv}_1$. If $\texttt{xv}_2 \leq \texttt{yv}_1$, $\texttt{xp}_2$ should be `null`; otherwise $\texttt{yp}_1$ must be `null` to guarantee the termination of the method and memory safety. Branch (6.9) states a similar condition when touching the second node of `y`. This formula is very precise, but not scalable if the analysis continues. According to the given user-defined predicate `sls`, we could abstract the shapes of `x` and `y` to be a sorted list. However, the formula is not sufficient to do that, i.e. the sortedness information about `x` and `y` is missing. This missing information is the numerical relation between $\texttt{xv}_1$ and $\texttt{xv}_2$ in `x` list, and $\texttt{yv}_1$ and $\texttt{yv}_2$ in `y` list. The guidance for this abstraction comes from the predicate `sls`. We use such user-defined predicates to infer data structure properties that are anticipated from some program codes. By applying abstraction (equipped with an abduction mechanism) against the predicate `sls` and then joining the branches with the same shape, the precondition from two iterations becomes:

$$\texttt{x}{=}\texttt{null} \vee \texttt{y}{=}\texttt{null} \vee \texttt{x}{::}\texttt{sls}\langle \texttt{xn}_0, \texttt{xsm}_0, \texttt{xlg}_0, \texttt{xp}_0\rangle * \texttt{y}{::}\texttt{sls}\langle \texttt{yn}_0, \texttt{ysm}_0, \texttt{ylg}_0, \texttt{yp}_0\rangle$$
$$\wedge\, 1{\leq}\texttt{xn}_0{\leq}2 \wedge 1{\leq}\texttt{yn}_0{\leq}2 \wedge (\texttt{xlg}_0{\leq}\texttt{ylg}_0 \wedge \texttt{xp}_0{=}\texttt{null} \vee \texttt{xlg}_0{>}\texttt{ylg}_0 \wedge \texttt{yp}_0{=}\texttt{null})$$

Continuing the analysis, the fixed point of the program summary is calculated as

$$(\mathsf{Pre}, \mathsf{Post}) :=$$
$$(\texttt{x}{=}\texttt{null} \vee \texttt{y}{=}\texttt{null} \vee \texttt{x}{::}\texttt{sls}\langle \texttt{xn}_0, \texttt{xsm}_0, \texttt{xlg}_0, \texttt{xp}_0\rangle{*}\texttt{y}{::}\texttt{sls}\langle \texttt{yn}_0, \texttt{ysm}_0, \texttt{ylg}_0, \texttt{yp}_0\rangle$$
$$\wedge\, (\texttt{xlg}_0{\leq}\texttt{ylg}_0{\wedge}\texttt{xp}_0{=}\texttt{null}{\vee}\texttt{xlg}_0{>}\texttt{ylg}_0{\wedge}\texttt{yp}_0{=}\texttt{null}),$$
$$\texttt{x}{=}\texttt{null}{\wedge}\texttt{res}{=}\texttt{y} \vee \texttt{y}{=}\texttt{null}{\wedge}\texttt{res}{=}\texttt{x}{\vee}$$
$$\texttt{x}{::}\texttt{sls}\langle \texttt{xn}_1, \texttt{xsm}_1, \texttt{xlg}_1, \texttt{xp}_1\rangle{*}\texttt{y}{::}\texttt{sls}\langle \texttt{yn}_1, \texttt{ysm}_1, \texttt{ylg}_1, \texttt{yp}_1\rangle \wedge \texttt{xn}_1{+}\texttt{yn}_1{=}\texttt{xn}_0{+}\texttt{yn}_0$$
$$\wedge \texttt{xsm}_1{=}\texttt{xsm}_0 \wedge \texttt{ysm}_1{=}\texttt{ysm}_0{\wedge}(\texttt{xsm}_0{\leq}\texttt{ysm}_0 \wedge \texttt{res}{=}\texttt{x}{\wedge}\texttt{xp}_1{=}\texttt{y} \wedge \texttt{xlg}_1{\leq}\texttt{ysm}_1$$
$$\vee \texttt{xsm}_0{>}\texttt{ysm}_0 \wedge \texttt{res}{=}\texttt{y} \wedge \texttt{yp}_1{=}\texttt{x} \wedge \texttt{ylg}_1{\leq}\texttt{xsm}_1)\,)$$

Figure 6.3 illustrates the specification via an instance of merge. The aliasing and heap shapes properties depend on the numerical information. `res` points to the smallest element, i.e. the node that `x` points to. Both `x` and `y` point to their original nodes, but some nodes of `x` are merged into the tail list from `y`. The grey block is the untouched heap of the procedure, as the merge process stops when the end of `x` list is reached, thus it is an unknown but safe heap portion by the footprint guided analysis for this procedure.

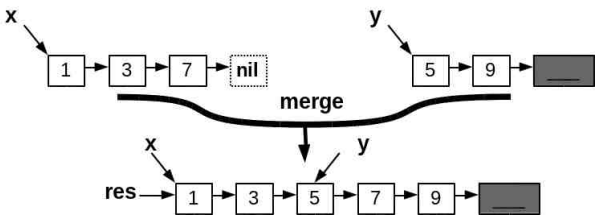

Figure 6.3: An Instance of Merge

From this example, we can observe that the memory safety of shape analysis is related to the values stored in the list. Our analysis can find that only one list is traversed to its end, i.e. until `null` is reached, and the other list is partially traversed till it reaches an element that is larger than the maximal value of the former list. As captured in the inferred precondition, the rest of the list will not be accessed by the program. Similarly, the inferred postcondition captures a fairly precise specification that represents the merged list which uses two list segments that either begins from `x` or from `y`, depending on which of the two input lists contains the smaller element. □

The experiment results are presented in Table 6.5 which shows the analysed methods, the number of code lines, and the analysis time in second respectively. We have analysed all of the method successfully, including programs processing AVL tree with its binary-search and height-balanced properties.

Porg.	LOC	Time
Singly Linked List		
create	10	1.12
delete	9	1.20
traverse	9	1.35
length	11	1.28
append	11	1.47
take	12	1.28
reverse	13	1.72
filter	15	2.37
drop_2nd	12	1.42
Sorting algorithm		
insert_sort	32	2.72
merge_sort	78	4.18
quick_sort	70	5.72
select_sort	45	3.16

Porg.	LOC	Time
Doubly Linked List		
create	15	1.47
append	24	2.53
insert	22	2.32
Binary Search Tree		
create	18	2.58
delete	48	4.76
insert	22	3.57
search	22	2.78
height	15	1.56
count	17	1.63
flatten	32	2.74
AVL Tree		
insert	114	27.57
delete	239	34.42

Table 6.5: Full specification discovery Experimental Results.

We note down two observations on the experimental results. The first is that the analysis may discover more than one specifications for some programs. For example, if we give two predicates, ordinary linked list and sorted list, for sorting algorithm, we can obtain two specifications for most of them. The reason is that a sorted list is also a linked list. When there are more than one predicate definitions supplied, the analysis can have multiple choices during the abstraction.

The other observation concerns the precision of the analysis, which is witnessed by the rich information inferred in the specifications. For precondition, our analysis discovers sufficient information to guarantee memory safety. For example, the preconditions of some programs require their input data structures are non-empty so that memory safety can be preserved. For the `take` program which traverses the list down a user-specified number `n` of nodes, we can find that the list length must be no less than `n`.

One restriction of our analysis we observed is that it requires a proper predicate to depict the requirement of a program. For example, if we only give an ordinary linked list for verifying merge sort, it will not succeed. This is because the stored value information in the list will not be preserved during the analysis.

6.3 Summary

This chapter has reported the experimental results obtained from our implemented systems. The results validate the feasibility and precision of both loop invariants and full specification discovering approaches. The advantage and disadvantage of the systems are also exhibited. Based on the outcomes, the achievement of this thesis confirms the contributions which generally meet the previous proposed objectives.

Chapter 7

Conclusions

This thesis focuses on analysis of both functional correctness and memory safety of programs that manipulates pointer-based data structures. The work has two steps by reducing the demand of user annotations. Firstly, we discover the loop invariants to assist the verification of loops. Secondly, we synthesise the full specification of given program without user annotations to free the labour of users. This chapter summaries the works presented in this thesis with its achieved results and discusses the potential future works.

7.1 Achieved Results

The main contributions of this dissertation is that I demonstrate it is possible and practical to analyse heap manipulating programs automatically and precisely by using abstract interpretation techniques in a combined shape and pure domain. To present this work, this thesis clearly defines a target programming language of the analysis. The language is simple, yet has the essential features of mainstream imperative languages that manipulate heap-based data structures. The operational semantics of the language is declared.

The domain of properties of the programming language that we are interested in is the combined shape and pure domain which can be used to capture both functional correctness and memory safety of heap manipulating programs. A specification language based on separation logic is defined in this combined domain. This specification language has an advantage feature to allow users to define their own predicates to specify the properties of data structures that they are interested in. Sophisticated numerical and shape properties can be expressed by this richly expressive language. The semantic model of it is declared.

Base on the target and specification language, the approaches are built up to discover loop invariants and full specification of the heap-manipulating language.

7.1.1 Loop Invariants Synthesis

Loop invariant is a key component of loop verification. An automated analysis system of discovery of loop invariant is proposed. The analysis system is based on abstract interpretation with fixed point computation. Novel abstraction, join and widening operators are specifically designed over the combined domain to guarantee the termination of the analysis. The soundness of the analysis is proved. The system has been implemented and integrated with Hip/Sleek verification system. The experimental results show the viability of the system.

7.1.2 Full Specification Discovery

A novel compositional analysis system is presented for generating full specification of programs without given any specification about the program codes in the combined domain. The compositional analysis is based on abstract interpretation technique with a novel bi-abduction algorithm over the combined domain. The system has implemented and the experimental results confirm the viability and precision of the framework in finding interesting properties of non-trivial programs.

7.2 Future works

In this section, I propose some possible application and improvement of this thesis as future works.

7.2.1 Resource Analysis

One future application of this work is to predict the resource requirement of heap-manipulating programs. The resource of a computing system is always limited, like the CPU power, memory size and battery capacity, particularly for embedded real-time systems. CPU time, memory usage and battery consumption are quite restricted for programs that run on such platforms. Paying insufficient attention to resource issues when developing such software may result in a corresponding software failure after deployment.

In this thesis, we have successfully analysed the shape of data structures and the related numerical properties. With the information that automatically inferred by our system, it is possible to extend the technique to predict the resource requirement of such programs. For instance, if we know the maximum sizes of recursive data structures allocated on the heap by a program, it is possible to prevent "Out of memory" error by checking the free memory before the program runs. If we know the length of a list, the timing of traversing the list is predictable.

An extension of Java language, Safety Critical Java (SCJ) (Henties et al., 2009), is based on a region-based memory management, which requires users to specify the memory usage of each mission to ensure the memory safety and avoid the delay caused by garbage collection. Equipped with our techniques, it is possible to calculate out the memory requirement of each mission statically, to eliminate such requirement from users, and to guarantee the memory safety of SCJ software.

We have had some achievements on the verification and analysis of memory usage requirement via separation logic and related numerical information (He et al., 2009; He and Luo, 2009). We will explore more and also study timing and battery issues with our analysis result.

7.2.2 Scalability

Currently, the system implementation is not focused on dealing with program code with large size. One restriction is that we still need to provide the user-defined predicates to satisfy the requirement of the program code, which will be discussed in next subsection. Another restriction is the limitation of the tools we adopt in our system. The numerical solver used in our system is Omega constraint solver (Pugh, 1991) which works in a compact convex polyhedra domain. It is a well known precise solver to eliminate existential variables and solve the Presburger Arithmetic. The advantage of Omega solver is in its precision, the disadvantage is in its cost. The worst case of

solving a Presburger formulae is exponential ($2^{2^{2^{O(n)}}}$).

To speed up the analysis, we may adopt some simple numerical domain, like interval abstract domain (Cousot and Cousot, 1976), linear equalities abstract domain (Chen et al., 2010), weighted hexagons abstract domain (Fulara et al., 2010), and octagon abstract domain (Miné, 2001). Users then have a choice between the precision and the efficiency.

Another one of the optimization aspects of the system is benefited from distributed computing system. It is also one of the advantage of compositional analysis. If some methods do not depend on each other from the call graph, they can be distributed to multiple computers and analysed in parallel to save analysis time. We hope the approaches addressed in this thesis could apply to real industry software in the future.

7.2.3 Predicates Discovery

The demand of user-defined predicates is one limitation of the analysis. For some sophisticated programs, the predicates are arduous to be defined unless the user understands the requirement very well. One dreamy idea is to discover the predicates automatically from the program code itself. It is not just a dream. A possible solution is to adopt counterexample-guided (CEG) approach (Podelski and Wies, 2010). Abduction is a kind of counterexample guided process. If the information we hold is not enough to continue the

verification of a program, i.e. a counterexample occurs, abduction analyses the counterexample and tries to add more knowledge to the precondition as the missing information to make the verification resume, and assumes that the missing information is part of precondition which should be given. CEG tries to strengthen the abstraction strategy of the analysis by analysing the counterexample, and then re-verifies the program. The work (Podelski and Wies, 2010) proposes a CEG algorithm to discover simple point-to relation in shape domain. Our analysis method employs user-defined predicates as the abstraction strategies. I believe it is possible to establish more complex predicates in the combined domain by combining both abduction and CEG techniques.

7.2.4 Arrays and Pointer Arithmetic

At the moment, we have not focused on dealing with arrays and pointer arithmetic in our analysis. To handle such features of program languages, we could exploit some techniques proposed by Calcagno et al. (2006) and Gulwani et al. (2008). The work (Calcagno et al., 2006) is founded on separation logic, where some restrictions are added over pointer arithmetics so that it is under control of the verification. Gulwani et al. (2008) construct a lifted abstract domain which is capable of representing universally quantified facts such as "$\forall i \cdot (0 \leq i < n) \rightarrow a[i]=0$". It is possible to incorporate such techniques to arrays in our verifier in order to verify a wider range of heap-manipulating programs.

7.2.5 Concurrency

Due to the rapid development of multi-core processor architectures, the verification of concurrent programs becomes a quite worthwhile research topic. Brookes (2004); O'Hearn (2007); Vafeiadis and Parkinson (2007); Feng et al. (2007); Dodds et al. (2009) propose a number of approaches to verify concurrent programs with separation logic. However, these works do not consider the numerical properties. It will be possible to extend our works to verify the concurrent programs in the combined domain in the future.

7.2.6 Program Derivation

This thesis mainly focuses on extracting formal specifications from program code to help the programmer debug and to verify the program. As a reversal process, program derivation (Burstall and Darlington, 1977; Kaldewau, 1990; Chin and Hu, 2002) starts with the formal specification of the user's requirement. The specification is then constructed to an executable implementation, and the correctness is promised by construction. To the best of my knowledge, no work based on separation logic has been done to derive programs which manipulate shared and mutable data structures till now. It will be an interesting research direction for future exploration.

7.3 Summary

This chapter summarises the contributed works presented in this thesis, discusses the limitations of the works and the potential improvements to over come the limitations. Two main contribution works are loop invariant synthesis and full specification discovery in a combined numerical and shape domain. The possible future improvements of the work include applying the approach to resource requirement calculation, improving the efficiency of the implementation, discovering predicates with counterexample-guided techniques, and extending our analysis to programs with arrays, pointer arithmetic and concurrency features. The future works depict possible directions of this dissertation in further steps. I believe, with further extensions, my works can make more contributions to the international grand challenge (Jones et al., 2006; Woodcock, 2006) in computer science.

Bibliography

Balaban, I., Pnueli, A., and Zuck, L. D. (2005). Shape analysis by predicate abstraction. In Cousot (2005b), pages 164–180.

Ball, T., Cook, B., Levin, V., and Rajamani, S. K. (2004). Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In Boiten, E. A., Derrick, J., and Smith, G., editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. (2001). Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA. ACM.

Ball, T. and Rajamani, S. K. (2001). Automatically validating temporal safety properties of interfaces. In Dwyer, M. B., editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer.

Ball, T. and Rajamani, S. K. (2002). The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA. ACM.

Barry, R. (2009). *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Ltd. http://www.freertos.org/.

Basse, S. (2009). *A gift of fire: social, legal, and ethical issues in computing*. Prentice Hall, third edition.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. (2007). Shape analysis for composite data structures. In Damm, W. and Hermanns, H., editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2004). A decidable fragment of separation logic. In Lodaya, K. and Mahajan, M., editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005a). Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005b). Symbolic execution with separation logic. In Yi, K., editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer.

Berdine, J., Cook, B., Distefano, D., and O'Hearn, P. W. (2006). Automatic termination proofs for programs with shape-shifting heaps. In Ball, T. and Jones, R. B., editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 386–400. Springer.

Bezier, B. (1990). *Software Testing Techniques*. Van Nostrand Rheinhold Company, New York, second edition.

Bidoit, M. and Mosses, P. D. (2004). CASL *User Manual*. LNCS 2900 (IFIP Series). Springer. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

Blair, M., Obenski, S., and Bridickas, P. (1992). *Patriot Missile Defense–Software Problem Led to System Failure at Dhahran, Saudi Arabia*. Tech. Rep. GAO/IMTEC-92-26, United States General Accounting Office, Washington, DC 20548.

Bozga, M., Iosif, R., and Lakhnech, Y. (2003). Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 55–65. ACM.

Brookes, S. D. (2004). A semantics for concurrent separation logic. In Gardner, P. and Yoshida, N., editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer.

Burstall, R. M. (1974). Program proving as hand simulation with a little induction. In *Proceedings of IFIP Congress 74*, pages 308–312. North-Holland.

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. (2006). Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In Yi (2006), pages 182–203.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. (2009). Compositional shape analysis by means of bi-abduction. In Shao and Pierce (2009), pages 289–300.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. (2011). Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26.

Calcagno, C., O'Hearn, P. W., and Yang, H. (2007). Local action and abstract separation logic. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378. IEEE Computer Society.

Calcagno, C., Yang, H., and O'Hearn, P. W. (2001). Computability and complexity results for a spatial assertion language for data structures. In Hariharan, R., Mukund, M., and Vinay, V., editors, *FSTTCS*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer.

CFTC and SEC (2010). Findings regarding the market events of may 6, 2010. Report of the staffs of the CFTC and SEC to the Joint Advisory Committee on Emerging Regulatory Issues. http://www.sec.gov/news/studies/2010/marketevents-report.pdf.

Chang, B.-Y. E. and Rival, X. (2008). Relational inductive shape analysis. In Necula and Wadler (2008), pages 247–260.

Chen, L., Miné, A., Wang, J., and Cousot, P. (2010). An abstract domain to discover interval linear equalities. In Barthe, G. and Hermenegildo, M. V., editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 112–128. Springer.

Chen, Y., Xia, B., Yang, L., and Zhan, N. (2007). Generating polynomial invariants with discoverer and qepcad. In Jones, C. B., Liu, Z., and Woodcock, J., editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 67–82. Springer.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2007). Automated verification of shape, size and bag properties. In *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*, pages 307–320. IEEE Computer Society.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2008). Enhancing modular OO verification with separation logic. In Necula and Wadler (2008), pages 87–99.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2010). Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *To appear in Science of Computer Programming*.

Chin, W.-N. and Hu, Z. (2002). Towards a modular program derivation via fusion and tupling. In Batory, D. S., Consel, C., and Taha, W., editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 140–155. Springer.

Clarke, E. M. (1979). Programming language constructs for which it is impossible to obtain good hoare axiom systems. *Journal of the ACM*, 26(1):129–147.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen, D.,

editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In Emerson, E. A. and Sistla, A. P., editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794.

Colón, M., Sankaranarayanan, S., and Sipma, H. (2003). Linear invariant generation using non-linear constraint solving. In Jr., W. A. H. and Somenzi, F., editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer.

Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90.

Cornhilll, D., Sha, L., and Lehoczky, J. P. (1987). Limitations of ada for real-time scheduling. In *Proceedings of the first international workshop on Real-time Ada issues*, IRTAW '87, pages 33–39, New York, NY, USA. ACM.

Cousot, P. (1981). Semantic foundations of program analysis. In Muchnick, S. and Jones, N., editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Cousot, P. (2005a). Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Cousot (2005b), pages 1–24.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France.

Cousot, P. and Cousot, R. (1977a). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM.

Cousot, P. and Cousot, R. (1977b). Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94. ACM.

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL 1979)*, pages 269–282. ACM.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM.

Cousot, R., editor (2005b). *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France,*

January 17-19, 2005, Proceedings, volume 3385 of *Lecture Notes in Computer Science*. Springer.

Deutsch, A. (1994). Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI 1994), SIGPLAN Notices 29(6)*, pages 230–241. ACM.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Distefano, D., O'Hearn, P. W., and Yang, H. (2006). A local shape analysis based on separation logic. In Hermanns, H. and Palsberg, J., editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer.

Distefano, D. and Parkinson, M. J. (2008). jstar: towards practical verification for java. In Harris, G. E., editor, *OOPSLA*, pages 213–226. ACM.

Dodds, M., Feng, X., Parkinson, M. J., and Vafeiadis, V. (2009). Deny-guarantee reasoning. In Castagna, G., editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer.

Dong, J. S. and Zhu, H., editors (2010). *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*. Springer.

Dowson, M. (1997). The ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22.

Feng, X., Ferreira, R., and Shao, Z. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. In Nicola (2007), pages 173–188.

Floyd, R. W. (1967). Assigning meanings to programs. In *Mathematical Aspects of Computer Science: Symposium in Applied Mathematics*, volume 19. American Mathematical Society.

Fulara, J., Durnoga, K., Jakubczyk, K., and Schubert, A. (2010). Relational abstract domain of weighted hexagons. *Electronic Notes in Theoretical Computer Science*, 267(1):59–72.

Giacobazzi, R. (1994). Abductive analysis of modular logic programs. In Bruynooghe, M., editor, *Logic Programming, Proceedings of the 1994 International Symposium*, pages 377–391. The MIT Press.

Gopalakrishnan, G. and Qadeer, S., editors (2011). *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*. Springer.

Gotsman, A., Berdine, J., and Cook, B. (2006). Interprocedural shape analysis with separated heap abstractions. In Yi (2006), pages 240–260.

Gulwani, S., Lev-Ami, T., and Sagiv, M. (2009). A combination framework for tracking partition sizes. In Shao and Pierce (2009), pages 239–251.

Gulwani, S., McCloskey, B., and Tiwari, A. (2008). Lifting abstract interpreters to quantified logical domains. In Necula and Wadler (2008), pages 235–246.

Gulwani, S. and Tiwari, A. (2007). Computing procedure summaries for interprocedural analysis. In Nicola (2007), pages 253–267.

Guo, B., Vachharajani, N., and August, D. I. (2007). Shape analysis with inductive recursion synthesis. In Ferrante, J. and McKinley, K. S., editors, *PLDI*, pages 256–265. ACM.

Gupta, A. and Malik, S., editors (2008). *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer.

Habermehl, P., Iosif, R., and Vojnar, T. (2010). Automata-based verification of programs with tree updates. *Acta Informatica*, 47(1):1–31.

Hackett, B. and Rugina, R. (2005). Region-based shape analysis with tracked locations. In Palsberg, J. and Abadi, M., editors, *POPL*, pages 310–323. ACM.

He, G. and Luo, C. (2009). Heap memory requirements analysis via separation logic. In Chin, W.-N. and Qin, S., editors, *TASE*, pages 321–322. IEEE Computer Society.

He, G., Qin, S., Luo, C., and Chin, W.-N. (2009). Memory usage verification using hip/sleek. In Liu, Z. and Ravn, A. P., editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 166–181. Springer.

Henties, T., Hunt, J. J., Locke, D., Nilsen, K., Schoeberl, M., and Vitek, J. (2009). Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2003). Software verification with BLAST. In Ball, T. and Rajamani, S. K., editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer.

Hermenegildo, M. V. and Palsberg, J., editors (2010). *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM.

Hetzel, W. C. and Hetzel, B. (1991). *The Complete Guide to Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, second edition.

Hoare, C. A. R. (1969). An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Hoare, C. A. R. and He, J. (1999). A trace model for pointers and objects. In Guerraoui, R., editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer.

Hoare, C. A. R. and Milner, R. (2004). *Grand challenges in computing research*. The British Computer Society, Swindon.

Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Ishtiaq, S. S. and O'Hearn, P. W. (2001). Bi as an assertion language for mutable data structures. In *Conference Record of the 28th ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages (POPL 2001), pages 14–26.

Jacobson, I., Booch, G., and Rumbaugh, J. E. (1999). *The unified software development process - the complete guide to the unified process from the original designers*. Addison-Wesley object technology series. Addison-Wesley.

Jones, C. B., O'Hearn, P. W., and Woodcock, J. (2006). Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95.

Jones, N. D. and Leroy, X., editors (2004). *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM.

Jones, S. P., editor (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, England.

Jonkers, H. M. B. (1981). Abstract storage structures. *Algorithmic languages*.

Kaldewau, A. (1990). *Programming: the derivation of algorithms*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

Kapur, D. (2005). Automatically generating loop invariants using quantifier elimination. In Baader, F., Baumgartner, P., Nieuwenhuis, R., and Voronkov, A., editors, *Deduction and Applications*, volume 05431 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Kildall, G. A. (1973). A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL 1973)*, pages 194–206.

Kuncak, V., Lam, P., and Rinard, M. C. (2002). Role analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 17–32.

Lee, O., Yang, H., and Petersen, R. (2011). Program analysis for overlaid data structures. In Gopalakrishnan and Qadeer (2011), pages 592–608.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In Clarke, E. M. and Voronkov, A., editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer.

Leroy, X., Doligez, D., Garrigue, J., Rmy, D., and Vouillon, J. (2010). *The Objective Caml system, documentation and user's manual – release 3.12*. INRIA.

Leveson, N. G. and Turner, C. S. (1993). Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41.

Lipton, R. J. (1977). A necessary and sufficient condition for the existence of hoare logics. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 1–6. IEEE Computer Society.

Magill, S., Berdine, J., Clarke, E. M., and Cook, B. (2007). Arithmetic strengthening for shape analysis. In Nielson, H. R. and Filé, G., editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 419–436. Springer.

Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. (2008). Thor: A tool for reasoning about shape and arithmetic. In Gupta and Malik (2008), pages 428–432.

Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. (2010). Automatic numeric abstractions for heap-manipulating programs. In Hermenegildo and Palsberg (2010), pages 211–222.

Manna, Z. and Pnueli, A. (1974). Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243–263.

McMillan, K. L. (1992). *Symbolic model checking – an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University.

Miné, A. (2001). The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 310–319. IEEE Computer Society.

Müller-Olm, M. and Seidl, H. (2004). Precise interprocedural analysis through linear algebra. In Jones and Leroy (2004), pages 330–341.

Necula, G. C. and Wadler, P., editors (2008). *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM.

Nguyen, H. H. and Chin, W.-N. (2008). Enhancing program verification with lemmas. In Gupta and Malik (2008), pages 355–369.

Nguyen, H. H., David, C., Qin, S., and Chin, W.-N. (2007). Automated verification of shape and size properties via separation logic. In Cook,

B. and Podelski, A., editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer.

Nicola, R. D., editor (2007). *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*. Springer.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of program analysis*. Springer.

O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307.

O'Hearn, P. W. and Pym, D. J. (1999). The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In Fribourg, L., editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer.

Parkinson, M. J. and Bierman, G. M. (2008). Separation logic, abstraction and inheritance. In Necula and Wadler (2008), pages 75–86.

Petit-Bianco, A. (1998). Java garbage collection for real-time systems. *Dr Dobb's Journal of Software Tools and Professional Programmers*, 23(10):8.

Pham, T.-H., Trinh, M.-T., Truong, A.-H., and Chin, W.-N. (2011). Fixbag: A fixpoint calculator for quantified bag constraints. In Gopalakrishnan and Qadeer (2011), pages 656–662.

Podelski, A. and Wies, T. (2010). Counterexample-guided focus. In Hermenegildo and Palsberg (2010), pages 249–260.

Popeea, C. and Chin, W.-N. (2006). Inferring disjunctive postconditions. In Okada, M. and Satoh, I., editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 331–345. Springer.

Pugh, W. (1990). Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.

Pugh, W. (1991). The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. IEEE Computer Society / ACM.

Qin, S., He, G., Luo, C., and Chin, W.-N. (2010a). Loop invariant synthesis in a combined domain. In Dong and Zhu (2010), pages 468–484.

Qin, S., Luo, C., Chin, W.-N., and He, G. (2011). Automatically refining partial specifications for program verification. In Butler, M. and Schulte, W., editors, *FM*, volume 6664 of *Lecture Notes in Computer Science*, pages 369–385. Springer.

Qin, S., Luo, C., He, G., Craciun, F., and Chin, W.-N. (2010b). Verifying heap-manipulating programs with unknown procedure calls. In Dong and Zhu (2010), pages 171–187.

Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini, M. and Montanari, U., editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer.

Reynolds, J. C. (2000). Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society.

Reynolds, J. C. (2005). An overview of separation logic. In Meyer, B. and Woodcock, J., editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer.

Rodríguez-Carbonell, E. and Kapur, D. (2007). Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75.

Rogers, M. W., editor (1984). *ADA: Language, compilers and bibliography*. Ada companion series. Cambridge University Press, New York, NY, USA.

Rondon, P. M., Kawaguchi, M., and Jhala, R. (2008). Liquid types. In Gupta, R. and Amarasinghe, S. P., editors, *PLDI*, pages 159–169. ACM.

Rondon, P. M., Kawaguchi, M., and Jhala, R. (2010). Low-level liquid types. In Hermenegildo and Palsberg (2010), pages 131–144.

Rugina, R. (2004). Shape analysis quantitative shape analysis. In Giacobazzi, R., editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 228–245. Springer.

Sagiv, S., Reps, T. W., and Wilhelm, R. (2002). Parametric shape analysis

via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298.

Sankaranarayanan, S., Sipma, H., and Manna, Z. (2004). Non-linear loop invariant generation using gröbner bases. In Jones and Leroy (2004), pages 318–329.

Shao, Z. and Pierce, B. C., editors (2009). *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM.

Skeel, R. (1992). Roundoff error and the patriot missile. *Society for Industrial and Applied Mathematics (SIAM) News*, 25(4).

Spivey, J. M. (1989). *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Vafeiadis, V. and Parkinson, M. J. (2007). A marriage of rely/guarantee and separation logic. In Caires, L. and Vasconcelos, V. T., editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer.

Venet, A. (1996). Abstract cofibered domains: Application to the alias analysis of untyped programs. In Cousot, R. and Schmidt, D. A., editors, *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer.

Venners, B. (1999). *Inside the Java Virtual Machine*. McGraw-Hill Professional, 1st edition.

Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F. (2003). Model checking programs. *Automated Software Engineering*, 10(2):203–232.

Woodcock, J. (2006). Verified software grand challenge. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 617–617. Springer.

Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. (2008). Scalable shape analysis for systems code. In *20th CAV*, volume 5123 of *LNCS*, pages 385–398. Springer.

Yang, H. and O'Hearn, P. W. (2002). A semantic basis for local reasoning. In Nielsen, M. and Engberg, U., editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer.

Yi, K., editor (2006). *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*. Springer.

Appendix A

Soundness of Abstract Semantics

We show the soundness proof of the abstract semantics for our analysis as below.

Lemma (Soundness of Abstract Semantics) If $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_1$, then for all s, h, if $s, h \models \mathit{Post}(\Delta)$ and $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, then there always exists Δ_0 such that

$$s_1, h_1 \models \mathit{Post}(\Delta_0) \qquad \text{and} \qquad [\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof. The proof is done by structural induction over program construc-

tors:

- Case $\texttt{null} \mid k \mid v \mid v.f$. Straightforward.
- Case $v = e$. There are two cases according to the operational semantics:
  - e is not a value. From operational semantics, there is e_1 s.t. $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, and $\langle s, h, v{=}e\rangle \hookrightarrow \langle s_1, h_1, v{=}e_1\rangle$. From abstract semantics for assignment, if $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_2$, and $\Delta_1 = [v_1/v', r_1/\texttt{res}]\ (\Delta_2) \wedge v' = r_1$. By induction hypothesis, there exists Δ_0, $s_1, h_1 \models \Delta_0$ and $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2$. It concludes from the assignment rule that $[\![v = e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.
  - e is a value. Trivial.
- Case $\texttt{new}\ c(\vec{v})$. From abstract semantics for $\texttt{new}$, we have $[\![\texttt{new}\ c(\vec{v})]\!]_{\mathcal{T}}\Delta = \Delta_1$, where $\Delta_1 = \Delta{*}\texttt{res::c}\langle \texttt{v}'_1, \ldots, \texttt{v}'_\texttt{n}\rangle$. Let $\Delta_0 = \Delta_1$. From the operational semantics, we have $\langle s, h, \texttt{new}\ c(\vec{v})\rangle \hookrightarrow \langle s, h{+}[\iota \mapsto r], \iota\rangle$, where $\iota \notin dom(h)$. From $s, h \models \Delta$, we have $s, h{+}[\iota \mapsto r] \models \Delta_0$. Moreover, $[\![\iota]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.
- Case $v_1.f = v_2$. Take $\Delta_0 = \Delta$. It concludes immediately from the exec rule for field update and the underlying operational semantics.
- Case $\texttt{free}(x)$. Denote Δ as $\bigvee_i(\texttt{x::c}\langle\tilde{\texttt{y}}_\texttt{i}\rangle{*}\sigma_i)$ and Δ_0 as $\bigvee_i \sigma_i$, then from free's operational semantics we know that if $s, h \models \mathit{Post}(\Delta)$ and $\langle s, h, \texttt{free}(x)\rangle \hookrightarrow \langle s_1, h_1, \text{-}\rangle$, then $s_1, h_1 \models \mathit{Post}(\Delta_0)$ and $\Delta_0 = \Delta_1$.
- Case $e_1; e_2$. We consider the case where e_1 is not a value (otherwise it is straightforward). From the operational semantics, we have $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. From the abstract semantics rule for sequence,

we have $\vdash \{\Delta\}e_1\{\Delta_2\}$. By induction hypothesis, there exists Δ_0 s.t. $s_1, h_1 \models \mathit{Post}(\Delta_0)$, and $\vdash \{\Delta_0\}e_3\{\Delta_2\}$. By the sequential rule we have $[\![e_3; e_2]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.

- Case `if` (v) e_1 `else` e_2. There are two possibilities in the operational semantics:

  - $s(v)$=`true`. We have $\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle$. Let $\Delta_0 = (\Delta \wedge v')$. It is obvious that $s, h \models \Delta_0$. From the if-conditional rule of abstract semantics, we have:

    $$[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2$$
    $$[\![e_2]\!]_{\mathcal{T}}\Delta \wedge \neg v' = \Delta_3$$

    And we also have (due to sound weakening of postcondition)

    $$[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2 \vee \Delta_3$$

    That is, $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.

  - $s(v) =$ `false`. Analogous.

- Case $mn(v_{1\ldots n})$. For the method invocation rule, we know $\Delta \vdash [v'_j/v_j]_{j=1}^{n}$ $\Phi^i_{pr} * \Delta^i$, for $i = 1, \ldots, p$. Take $\Delta_0 = \bigvee_{i=1}^{p} [v'_j/v_j]_{j=1}^{n} \Phi^i_{pr} * \Delta^i$. From the operational semantics and the above heap entailment, we have $s_1, h_1 \models \Delta_0$. Then the method invocation rule implies $\forall i \in 1 \ldots p \cdot [\![e_1]\!]_{\mathcal{T}}[v'_j/v_j]_{j=1}^{n} \Phi^i_{pr} * \Delta^i = \Delta^i * \Phi^i_{po}$. Therefore we have $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$ which concludes.

Appendix B

Collection of Shape Predicates

We list a collection of the definition of shape predicates used by the thesis and the experiments.

$$\texttt{list}\langle\rangle \;\equiv\; (\texttt{root=null}) \vee (\texttt{root::Node}\langle \texttt{i}, \texttt{q}\rangle * \texttt{q::list}\langle\rangle)$$

$$\texttt{ll}\langle \texttt{n}\rangle \equiv (\texttt{root=null} \wedge \texttt{n=0}) \vee (\texttt{root::Node}\langle \_, \texttt{q}\rangle * \texttt{q::ll}\langle \texttt{m}\rangle \wedge \texttt{n=m+1})$$

$$\texttt{ls}\langle \texttt{n}, \texttt{p}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node}\langle \_, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{ls}\langle \texttt{m}, \texttt{p}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

$$\begin{aligned}\texttt{llB}\langle \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \\ & \vee (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{llB}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1)\end{aligned}$$

$$\begin{aligned}\texttt{sllB}\langle \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee \\ & (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{sllB}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x} {\in} \texttt{S}_1 {\cdot} \texttt{v} {\leq} \texttt{x}))\end{aligned}$$

$$\begin{aligned}\texttt{dll}\langle \texttt{p}, \texttt{n}\rangle \equiv\ & (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee \\ & (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{dll}\langle \texttt{root}, \texttt{n}_1\rangle \wedge \texttt{n}{=}\texttt{n}_1{+}1)\end{aligned}$$

$$\begin{aligned}\texttt{dllB}\langle \texttt{p}, \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{p} \wedge \texttt{S}{=}\emptyset) \vee \\ & (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle {*} \texttt{q}{::}\texttt{dllB}\langle \texttt{root}, \texttt{S}_1\rangle \wedge \texttt{S}{=}\texttt{S}_1 \sqcup \{\texttt{v}\})\end{aligned}$$

$$\begin{aligned}\texttt{dlls}\langle \texttt{pr}, \texttt{bo}, \texttt{bi}, \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{bo} \wedge \texttt{pr}{=}\texttt{bi} \wedge \texttt{S}{=}\emptyset) \vee \\ & (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{pr}, \texttt{nx}\rangle * \texttt{nx}{::}\texttt{dlls}\langle \texttt{root}, \texttt{bo}, \texttt{bi}, \texttt{S1}\rangle \\ & \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x} {\in} \texttt{S}_1 {\cdot} \texttt{v} {\leq} \texttt{x}))\end{aligned}$$

$$\texttt{sll}\langle \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root::node}\langle \texttt{mn}, \texttt{null}\rangle \wedge \texttt{n}{=}1 \wedge \texttt{mn}{=}\texttt{mx}) \vee$$
$$(\texttt{root::node}\langle \texttt{mn}, \texttt{q}\rangle * \texttt{q::sll}\langle \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{mn} \leq \texttt{k} \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

$$\texttt{slsB}\langle \texttt{p}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{S}{=}\emptyset) \vee$$
$$(\texttt{root::Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q::slsB}\langle \texttt{p}, \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{x}))$$

$$\texttt{sls}\langle \texttt{n}, \texttt{s}, \texttt{l}, \texttt{p}\rangle \equiv (\texttt{root::Node}\langle \texttt{s}, \texttt{p}\rangle \wedge \texttt{n}{=}1 \wedge \texttt{l} = \texttt{s}) \vee$$
$$\texttt{root::Node}\langle \texttt{s}, \texttt{q}\rangle * \texttt{q::sls}\langle \texttt{m}, \texttt{s1}, \texttt{l}, \texttt{p}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1 \wedge \texttt{s} \leq \texttt{s1} \wedge \texttt{s1} \leq \texttt{l}$$

$$\texttt{bnd}\langle \texttt{s}, \texttt{l}, \texttt{p}\rangle \equiv (\texttt{root::Node}\langle \texttt{v}, \texttt{p}\rangle) \wedge \texttt{s}{=}\texttt{v} \wedge \texttt{l}{=}\texttt{v} \vee$$
$$\texttt{root::Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q::bnd}\langle \texttt{s}_1, \texttt{l}_1, \texttt{p}\rangle \wedge \texttt{s} = \min(\texttt{s}_1, \texttt{v}) \wedge \texttt{l} = \max(\texttt{l}_1, \texttt{v})$$

$$\texttt{bt}\langle \texttt{S}, \texttt{h}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset \wedge \texttt{h}{=}0) \vee (\texttt{root::Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle *$$
$$\texttt{p::bt}\langle \texttt{S}_\texttt{p}, \texttt{h}_\texttt{p}\rangle * \texttt{q::bt}\langle \texttt{S}_\texttt{q}, \texttt{h}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{p} \sqcup \texttt{S}_\texttt{q} \wedge \texttt{h}{=}1{+}\mathsf{max}(\texttt{h}_\texttt{p}, \texttt{h}_\texttt{q}))$$

$$\texttt{treep}\langle \texttt{p}\rangle \equiv \texttt{root}{=}\texttt{null} \vee$$
$$\texttt{root::Node3}\langle \_, \texttt{l}, \texttt{r}, \texttt{p}\rangle * \texttt{l::treep}\langle \texttt{root}\rangle * \texttt{r::treep}\langle \texttt{root}\rangle$$

$$\mathtt{bst}\langle \mathtt{sm}, \mathtt{lg}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{sm}{=}\mathtt{lg}) \vee$$
$$(\mathtt{root}{::}\mathtt{Node2}\langle \mathtt{v}, \mathtt{p}, \mathtt{q}\rangle {*} \mathtt{p}{::}\mathtt{bst}\langle \mathtt{sm}, \mathtt{mn}\rangle {*} \mathtt{q}{::}\mathtt{bst}\langle \mathtt{mx}, \mathtt{lg}\rangle \wedge \mathtt{mn}{<}\mathtt{v}{<}\mathtt{mx})$$

$$\mathtt{avl}_1\langle \mathtt{S}, \mathtt{h}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{h}{=}0) \vee (\mathtt{root}{::}\mathtt{Node2}\langle \mathtt{v}, \mathtt{p}, \mathtt{q}\rangle {*} \mathtt{p}{::}\mathtt{bt}\langle \mathtt{S_p}, \mathtt{h_p}\rangle \ *$$
$$\mathtt{q}{::}\mathtt{bt}\langle \mathtt{S_q}, \mathtt{h_q}\rangle \wedge \mathtt{S}{=}\mathtt{S_p}{\sqcup}\mathtt{S_q} \wedge \mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h_p}, \mathtt{h_q}) \wedge -1{\leq}\mathtt{h_p}{-}\mathtt{h_q}{\leq}1)$$

$$\mathtt{avl}_2\langle \mathtt{h}, \mathtt{S}\rangle \equiv \mathtt{root}{=}\mathtt{null}{\wedge}\mathtt{h}{=}0 \wedge \mathtt{S}{=}\emptyset$$
$$\vee \mathtt{root}{::}\mathtt{Node2}\langle \mathtt{v}, \mathtt{l}, \mathtt{r}\rangle * \mathtt{l}{::}\mathtt{avl}\langle \mathtt{h_l}, \mathtt{S_l}\rangle * \mathtt{r}{::}\mathtt{avl}\langle \mathtt{h_r}, \mathtt{S_r}\rangle$$
$$\wedge \mathtt{h}{=}1{+}\mathtt{max}(\mathtt{h_l}, \mathtt{h_r}) \wedge -1{\leq}\mathtt{h_l}{-}\mathtt{h_r}{\leq}1$$
$$\wedge \mathtt{S}{=}\mathtt{S_l}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_l}{\cdot}\mathtt{x}{\leq}\mathtt{v}) \wedge (\forall \mathtt{x}{\in}\mathtt{S_r}{\cdot}\mathtt{v}{<}\mathtt{x})$$